GateXAIML
AIML | Data Science | CSE | Mathematics

GATE
Data Science & Artificial Intelligence

Data Structures & Algorithms

GateXAIML

Happy Learning!

Contents

About the Book **5**

1 Algorithm Analysis **7**
1.1 Introduction . . . 7
1.2 Core Algorithmic Paradigms . . . 8
1.3 Analysis Framework . . . 11
1.3.1 Time and Space Efficiency . . . 11
1.3.2 Input Size and Efficiency Analysis . . . 11
1.3.3 Measuring Running Time of Algorithms . . . 12
1.3.4 Order of Growth . . . 14
1.4 Asymptotic Notations . . . 16
1.4.1 Asymptotic Efficiency of Algorithms . . . 16
1.4.2 O-Notation (Big-Oh Notation) . . . 18
1.4.3 Ω-Notation (Big-Omega Notation) . . . 21
1.4.4 Θ-Notation (Big-Theta Notation) . . . 23
1.4.5 Relationship Between O, Ω, and Θ . . . 25
1.4.6 Little-o Notation . . . 27
1.4.7 Little-ω Notation . . . 28
1.4.8 Relationship Between O, Ω, Θ, o, and ω . . . 30
1.5 Best Case, Worst Case and Average Case . . . 35
1.6 Relation Between O, Ω, Θ and Best, Worst, Average Case . . . 37
1.7 Analysis of Non Recursive Algorithms . . . 39
1.7.1 Useful Maths Formulas . . . 39
1.7.2 Mathematical Analysis of Nonrecursive Algorithms . . . 42
1.8 Recurrence Relations . . . 48
1.8.1 Substitution Method . . . 49
1.8.2 Recursion Tree Method . . . 50
1.8.3 Extended Master Theorem . . . 51
1.9 Analysis of Recursive Algorithms . . . 53
1.10 Useful Formulas . . . 59
1.11 Practice Problems . . . 63
1.12 GATE PYQs . . . 75
1.13 YouTube Links and QR Codes . . . 84

2 Stacks **87**
2.1 Core Operations . . . 87
2.1.1 Array Implementation . . . 88
2.2 Applications of Stacks . . . 91
2.2.1 Expression Notations: Infix, Postfix, Prefix . . . 91
2.2.2 Infix to Postfix Conversion (Shunting-Yard Algorithm) . . . 92
2.2.3 Postfix Expression Evaluation . . . 94
2.2.4 Infix to Prefix Conversion . . . 96
2.2.5 Prefix Expression Evaluation . . . 98
2.2.6 Prefix to Infix Conversion . . . 99
2.2.7 Postfix to Prefix Conversion . . . 99

2.2.8 Prefix to Postfix Conversion . . . 100
2.2.9 Balanced Parentheses / Brackets Check . . . 101
2.2.10 Recursion and Call Stack Frames . . . 103
2.3 Practice Problems . . . 106
2.4 GATE PYQs . . . 113
2.5 YouTube Links and QR Codes . . . 116

3 Queues 117
3.1 Core Operations . . . 117
3.1.1 Array Implementation of Queue . . . 118
3.1.2 Circular Queue . . . 121
3.1.3 Deque (Double-Ended Queue) . . . 123
3.2 Applications of Queues . . . 126
3.3 Practice Problems . . . 128
3.4 GATE PYQs . . . 133
3.5 YouTube Links and QR Codes . . . 136

4 Linked Lists 137
4.1 Singly Linked List . . . 138
4.1.1 Singly LL — Insertion Operations . . . 138
4.1.2 Singly LL — Deletion Operations . . . 141
4.1.3 Singly LL — Search Operation . . . 143
4.1.4 Singly LL — Reversal . . . 144
4.2 Doubly Linked List . . . 145
4.2.1 Doubly LL — Insertion Operations . . . 146
4.2.2 Doubly LL — Deletion Operations . . . 149
4.2.3 Doubly LL — Search Operation . . . 153
4.3 Circular Linked List . . . 154
4.3.1 Circular LL — Insertion Operations . . . 155
4.3.2 Circular LL — Deletion Operations . . . 158
4.3.3 Circular LL — Search Operation . . . 163
4.4 Classic Problems . . . 164
4.4.1 Floyd's Cycle Detection Algorithm . . . 164
4.4.2 Finding the Start of a Cycle . . . 165
4.4.3 Finding the Middle Node . . . 166
4.4.4 Merge Two Sorted Linked Lists . . . 167
4.4.5 Nth Node from the End . . . 169
4.4.6 Intersection of Two Linked Lists . . . 170
4.5 Linked List Implementation of Stack and Queue . . . 174
4.5.1 Stack Using Linked List . . . 174
4.5.2 Queue Using Linked List . . . 176
4.6 Practice Problems . . . 178
4.7 GATE PYQs . . . 186
4.8 YouTube Links and QR Codes . . . 191

5 Trees 193
5.1 Terminology and Structure . . . 193
5.1.1 Full, Complete, and Perfect Binary Tree . . . 195
5.1.2 Array Representation of a Binary Tree . . . 196
5.2 Binary Tree Formulas . . . 197
5.3 Theorems and Structural Properties of Binary Trees . . . 202
5.3.1 Full Binary Tree . . . 202
5.3.2 Perfect Binary Tree . . . 204
5.3.3 Complete Binary Tree . . . 206
5.4 Binary Tree Traversals . . . 210
5.4.1 Inorder Traversal (Left, Root, Right) . . . 210

5.4.2 Preorder Traversal (Root, Left, Right) . . . 211
5.4.3 Postorder Traversal (Left, Right, Root) . . . 211
5.4.4 Reconstructing a Tree from Two Traversals . . . 211
5.5 Binary Search Tree (BST) . . . 216
5.5.1 Construct a BST from a Given Insertion Sequence . . . 217
5.5.2 Search in BST . . . 219
5.5.3 Insert in BST . . . 219
5.5.4 Inorder Successor and Predecessor . . . 220
5.5.5 Delete in BST: Three Cases . . . 224
5.5.6 Inorder Traversal of BST Gives Sorted Order . . . 226
5.5.7 Height of BST: Best and Worst Case . . . 226
5.5.8 Number of Distinct BSTs: Catalan Numbers . . . 226
5.6 Practice Problems . . . 228
5.7 GATE PYQs . . . 235
5.8 YouTube Links and QR Codes . . . 247

6 Hash Tables 249
6.1 Hash Functions and Load Factor . . . 249
6.1.1 Division Method . . . 250
6.1.2 Multiplication Method . . . 250
6.1.3 Load Factor . . . 251
6.2 Collision Resolution . . . 252
6.2.1 Chaining (Separate Chaining) . . . 252
6.2.2 Open Addressing . . . 253
6.3 Performance Analysis . . . 257
6.4 Practice Problems . . . 260
6.5 GATE PYQs . . . 266
6.6 YouTube Links and QR Codes . . . 271

7 Graph Theory — Introduction 272
7.1 Basic Definitions . . . 272
7.1.1 Vertex, Edge, and Graph $G = (V, E)$ . . . 272
7.1.2 Directed vs. Undirected Graph . . . 273
7.1.3 Weighted vs. Unweighted Graph . . . 273
7.1.4 Degree, In-Degree, and Out-Degree . . . 273
7.1.5 Handshaking Lemma . . . 274
7.1.6 Complete Graph K_n . . . 274
7.1.7 Directed Acyclic Graph (DAG) . . . 275
7.1.8 Tree as a Special Graph . . . 275
7.1.9 Connected Graph and Connected Components . . . 276
7.1.10 Strongly Connected and Weakly Connected (Directed Graphs) . . . 277
7.1.11 Walk, Trail, Circuit, Path, and Cycle . . . 278
7.1.12 Havel–Hakimi Theorem . . . 280
7.2 Graph Representations . . . 283
7.2.1 Adjacency Matrix . . . 283
7.2.2 Adjacency List . . . 284
7.2.3 Dense vs. Sparse Graphs . . . 284
7.3 Practice Problems . . . 286
7.4 GATE PYQs . . . 292
7.5 YouTube Links and QR Codes . . . 296

8 Graph Traversals 297
8.1 Breadth-First Search (BFS) . . . 297
8.1.1 Introduction to BFS . . . 297
8.1.2 BFS Algorithm . . . 297
8.1.3 BFS Trace: Example Graph . . . 298

8.1.4 BFS on Directed Graphs . . . 299
8.1.5 BFS on Disconnected Graph — Connected Components . . . 299
8.1.6 Properties of BFS . . . 300
8.2 Depth-First Search (DFS) . . . 305
8.2.1 DFS Discovery and Finish Timestamps . . . 305
8.2.2 DFS Step-by-Step Trace with Timestamps . . . 305
8.2.3 DFS Edge Classification . . . 306
8.3 Topological Sort . . . 308
8.3.1 Method 1: DFS-Based Topological Sort . . . 309
8.3.2 Method 2: Kahn's Algorithm (BFS with In-Degree) . . . 309
8.4 Practice Problems . . . 313
8.5 GATE PYQs . . . 316
8.6 YouTube Links and QR Codes . . . 324

9 Shortest Path Algorithms 325
9.1 Single-Source: Dijkstra's Algorithm . . . 325
9.1.1 Dijkstra: Step-by-Step Trace (Standard Example) . . . 327
9.1.2 Dijkstra: Complex Trace (Directed Graph with Reordering) . . . 328
9.2 Single-Source: Bellman-Ford Algorithm . . . 329
9.2.1 Bellman-Ford: Step-by-Step Trace . . . 330
9.3 All-Pairs: Floyd-Warshall Algorithm . . . 332
9.3.1 Floyd-Warshall: Step-by-Step Matrix Trace . . . 333
9.4 Choosing the Right Algorithm . . . 335
9.5 Practice Problems . . . 336
9.6 GATE PYQs . . . 340
9.7 YouTube Links and QR Codes . . . 344

10 Searching and Sorting 345
10.1 Search Algorithms . . . 346
10.1.1 Linear Search . . . 346
10.1.2 Binary Search . . . 347
10.1.3 Binary Search on the Answer . . . 351
10.2 Basic (Quadratic) Sorting . . . 352
10.2.1 Selection Sort . . . 352
10.2.2 Bubble Sort . . . 353
10.2.3 Insertion Sort . . . 355
10.2.4 Comparison of the Three Quadratic Sorts . . . 357
10.3 Divide and Conquer Sorting . . . 357
10.3.1 Merge Sort . . . 357
10.3.2 Quicksort . . . 360
10.3.3 Lower Bound for Comparison-Based Sorting . . . 365
10.4 Randomised Quicksort . . . 366
10.4.1 Examples . . . 367
10.4.2 Performance . . . 367
10.5 Master Summary . . . 369
10.6 Practice Problems . . . 370
10.7 GATE PYQs . . . 373
10.8 YouTube Links and QR Codes . . . 379

11 Solutions to Practice Problems 380

12 Solutions to GATE PYQs 382

About the Book

Artificial Intelligence and Machine Learning (AI/ML) are transforming industries across the globe — from healthcare and finance to transportation and education. From medical diagnosis systems and fraud detection to personalized recommendations and autonomous vehicles, AI/ML is shaping the way we live, work, and interact with technology.

To support this rapidly growing field, the GATE Data Science and Artificial Intelligence (DA) exam was introduced as a national-level gateway to higher studies, research, and employment opportunities in top institutions and organizations. The exam tests a candidate's proficiency in mathematics, programming, data handling, machine learning, and AI fundamentals.

This book is a compact and comprehensive guide for GATE DA aspirants. It is designed to help learners build a strong conceptual foundation while developing the problem-solving skills required for the exam. Many solved examples are included to illustrate key concepts, and each chapter features carefully crafted problems for practice.

Solutions to selected problems and topic-wise lectures will be discussed in detail on my YouTube channel (@GATEXAiml). All the concepts covered in the book will also be taught step-by-step through video tutorials, making this a complete learning resource for GATE DA preparation.

This book is designed for aspirants of the GATE DA exam focusing on **Data Structures and Algorithms**. It systematically covers theory, solved examples, and practice problems aligned with the official syllabus.

Dedicated to all my Gurus and Students.
"Knowledge grows only when shared — and it must remain free, for that is how it thrives."

gatexaiml

Data Structures and Algorithms – Syllabus

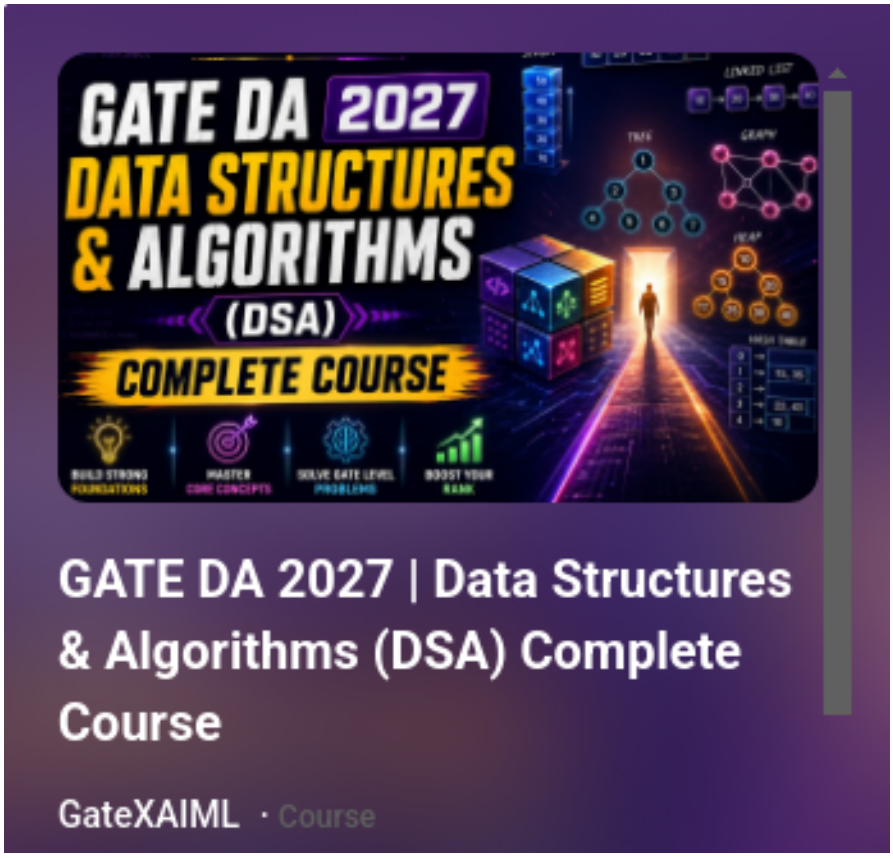
Basic data structures: stacks, queues, linked lists, trees, hash tables; Search algorithms: linear search and binary search, basic sorting algorithms: selection sort, bubble sort and insertion sort; divide and conquer: mergesort, quicksort; introduction to graph theory; basic graph algorithms: traversals and shortest path.

Attention!

Some examples solved in video lectures are different from those given in this book.

The procedure to solve problems and examples is well explained in the video lectures, and it is highly recommended to go through the video lectures for complete understanding.

Official Video Playlist



Watch on YouTube

Chapter 1

Algorithm Analysis

1.1 Introduction

Data Structures

A data structure is a way to store data in a computer so that we can use it easily. Just like we keep clothes in a cupboard with shelves, so that we find them fast, a data structure keeps data in an organised way. Different data structures store data in different ways. We pick the right one based on what work we want to do with the data.

Example 1: Train and Bogies

Think of a train. Each bogie is connected to the next bogie in a line. This is like a Linked List data structure, where one item points to the next item.

Example 2: Choosing a Data Structure

Suppose we want to check if a webpage URL is already visited or not, very fast, again and again. Which data structure suits best, a List or a Hash Table? A Hash Table suits best here, because checking an item in a Hash Table takes less time on average than checking in a List.

Algorithms

An algorithm is a step by step method to solve a problem. It is like a recipe to cook food. The recipe tells what to do first, what to do next, and what to do last. An algorithm takes some input, does some steps, and gives an output. A good algorithm should be clear, correct, and should finish in finite steps.

Example 3: Making Tea

To make tea, we follow steps. Boil water, add tea powder, add milk, add sugar, boil again, and pour in a cup. This is an algorithm, because it has clear steps in a fixed order.

Example 4: Finding Maximum Number

Given an array $A = [4, 9, 2, 7, 1]$, an algorithm to find the maximum number will compare each element one by one with a variable holding the current maximum, and update it when a bigger number is found. At the end, the variable holds the maximum value, which is 9.

Algorithm: Find-Max(A)

```
1: max ← A[0]
2: for i = 1 to A.length − 1 do
3:     if A[i] > max then
4:         max ← A[i]
5:     end if
6: end for
7: return max
```

Pseudocode

Pseudocode is a simple way to write the steps of an algorithm using plain words and simple structure, similar to a programming language but not exact. It is not a real programming language, so we cannot run it on a computer. It helps us to plan the logic first, before writing real code. We write it using simple words like if, for, while, so anyone can understand it easily.

Example 5: Pseudocode for Finding Maximum

Find-Max(A)

1. max = A[0]
2. for i = 1 to length(A) - 1
3. if A[i] > max
4. max = A[i]
5. return max

Code (Python)

Code is the actual set of instructions written in a programming language, which a computer can run directly. We take the logic from pseudocode and convert it into Python code, following Python rules like indentation and syntax. Once written, this code can be run on a computer to get the real output.

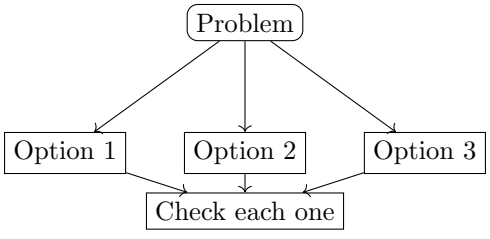
Python implementation: find-max.py

```python
def find_max(arr):
    max_val = arr[0]
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
    return max_val
```

1.2 Core Algorithmic Paradigms

Brute Force (Exhaustive Search)

Brute force means we check every possible option one by one, until we find the correct answer. We do not use any trick or shortcut. It is simple to think and write, but it can take a lot of time when the number of options is large.

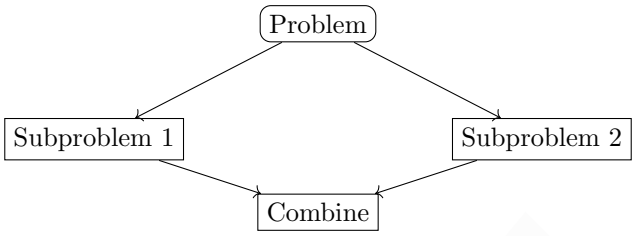


Example 6: Applying Brute Force

Suppose we want to find two numbers in a list that add up to a target value. Brute force will pick every pair of numbers, one pair at a time, and check if their sum matches the target. It keeps doing this for all possible pairs.

Divide and Conquer

This method breaks a big problem into smaller parts, called subproblems. These subproblems are independent of each other, which means one subproblem does not depend on another. We solve each small part separately, and then we combine all the small answers to get the final answer.

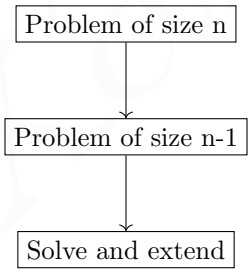


Example 7: Applying Divide and Conquer

Suppose we want to sort a big list of numbers. Divide and conquer will split the list into two halves, sort each half on its own, and then join the two sorted halves into one sorted list.

Decrease and Conquer

This method reduces the problem to one smaller problem of the same type, not many. We solve that one smaller problem, and then we use its answer to build the answer for the original problem.

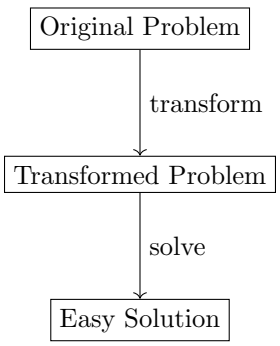


Example 8: Applying Decrease and Conquer

Suppose we want to find the maximum number in a list. Decrease and conquer will find the maximum in the list without the last element, which is a smaller problem, and then compare that result with the last element to get the final maximum.

Transform and Conquer

This method changes the problem into a different, simpler form, which is easier to solve. After solving the new simpler form, we use that result for our original problem.

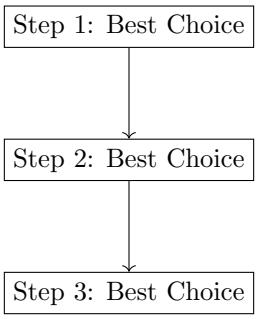


Example 9: Applying Transform and Conquer

Suppose we want to check if a list has any duplicate numbers. Transform and conquer will first sort the list, which is a transformation, and then simply check neighbouring elements, which becomes a much easier check.

Greedy Method

This method picks the best option available right now, at every step, without thinking about the future steps. It hopes that picking the best choice every time will give the best overall answer at the end.

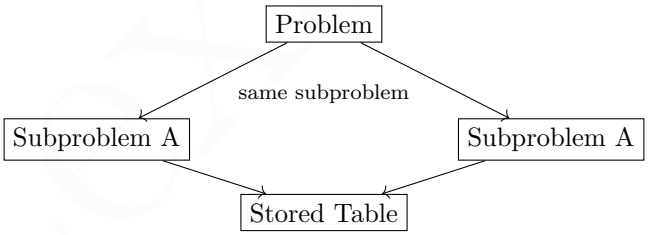


Example 10: Applying Greedy Method

Suppose we want to give change to a customer using minimum number of coins. Greedy method will always pick the largest coin value possible at each step, until the full change amount is given.

Dynamic Programming

This method also breaks a problem into smaller subproblems, like divide and conquer. But here, the same subproblem can come again and again, so we store the answer of each subproblem once, and reuse it later instead of solving it again.

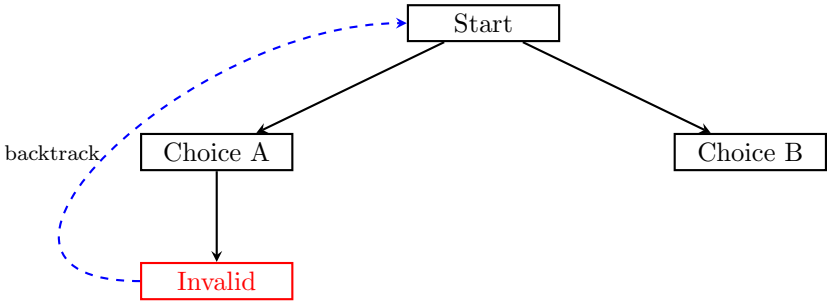


Example 11: Applying Dynamic Programming

Suppose we want to find the number of ways to climb a staircase, taking one or two steps at a time. Dynamic programming will store the number of ways for each smaller staircase height, and reuse those stored values to build the answer for the full staircase.

Backtracking

This method builds a solution step by step. At each step, if we find that the current path breaks a rule of the problem, we go back to the previous step and try a different option. This way, we avoid wrong paths early.

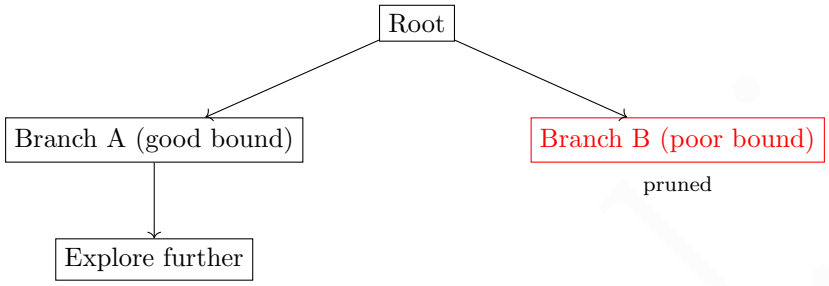


Example 12: Applying Backtracking

Suppose we want to place numbers in a sudoku grid. Backtracking will place a number in a cell, and if it breaks a sudoku rule, it removes that number and tries the next possible number in that cell.

Branch and Bound

This method is used for optimization problems, where we want the best possible answer, like minimum cost or maximum profit. It builds a tree of choices, and at each branch, it calculates a bound, which tells if that branch can possibly give a better answer or not. If a branch cannot give a better answer, we cut it off and do not explore it further.



Example 13: Applying Branch and Bound

Suppose a delivery person wants the shortest route to visit many cities. Branch and bound will build partial routes, calculate a lower bound cost for each partial route, and skip any partial route whose bound is already worse than the best route found so far.

1.3 Analysis Framework

1.3.1 Time and Space Efficiency

Time and Space Efficiency

We check how good an algorithm is using two things, time and space. Time efficiency means how long the algorithm takes to run. Space efficiency means how much extra memory the algorithm needs, apart from the input and output.

Example 14: Cooking Time vs Kitchen Space

Think of cooking food. Time efficiency is like how many minutes the cooking takes. Space efficiency is like how many extra vessels and counter space we need while cooking, apart from the raw ingredients and the final dish.

Primary Focus on Time

Nowadays, computers have a lot of memory. So memory is not a big problem most of the time. But running time is still important, because we want answers fast. So, in most cases, we focus more on time efficiency than space efficiency.

1.3.2 Input Size and Efficiency Analysis

Input Size and Algorithm Efficiency

The work done by an algorithm depends on how big the input is. A bigger input usually means more work. So, we write the running time as a function of a number n, which represents the size of the input.

Example 15: Sweeping a Room

Sweeping a small room takes less time. Sweeping a big hall takes more time. Here, the size of the room is like input size n, and the sweeping time grows as the room size grows.

Selecting an Appropriate Size Measure

For most problems, we pick the input size directly from the data we are given, like the number of elements in a list, the number of rows in a table, or the number of terms in an expression. Small differences in counting do not change our final conclusion about efficiency.

Example 16: Checking a Student ID in a File

Suppose we want to check if a student ID is present in a record file. The algorithm reads each record one by one, compares the ID, and stops when found or when the file ends. If the file has n records, the algorithm may check up to n records in the worst case. If the file size doubles, the work also doubles. So here, input size is the number of records.

Algorithm: Check-Student-ID(records, target)

```
for i = 1 to records.length do
    if records[i].ID = target then
        return true
    end if
end for
return false
```

Multiple Valid Size Metrics

Some problems can have more than one way to measure input size. The algorithm itself does not change, but the way we write its growth function changes, depending on which size measure we choose.

Example 17: Matrix Size Measure

For a matrix, we can measure size as the number of rows, or as the total number of elements (rows times columns). Both are valid, but the growth function looks different depending on which one we use.

Operation-Dependent Measurement

We should choose the size measure based on what unit the algorithm actually works on. If the algorithm works on characters, we count characters. If it works on records or rows, we count those instead.

Example 18: Counting Letters vs Counting Words

An algorithm that checks spelling works on each character of a word, so its input size is the number of characters. An algorithm that counts how many words are in a sentence works on each word, so its input size is the number of words.

1.3.3 Measuring Running Time of Algorithms

Why Clock Time Is Not a Good Measure

We should not measure an algorithm using seconds or milliseconds. This is because clock time depends on many outside things, like processor speed, programming language used, coding style, and how busy the system is at that moment. These things are not part of the algorithm. So, clock time does not give a fair measure of how good an algorithm is.

Example 19: Same Recipe, Different Kitchens

If two cooks use the same recipe, but one has a fast gas stove and the other has a slow stove, their cooking time will differ. But the recipe, that is the method, is the same for both. So, we should judge the recipe by its steps, not by the stove used.

Operation Count as a Machine-Independent Measure

Instead of checking how many seconds an algorithm takes, we count how many times an important operation runs. This count does not depend on the computer used. We do not count every single operation, we only count the operation that takes the most work.

Basic Operation

The basic operation is the one operation that takes the most time in an algorithm. It is usually present inside the innermost loop, and it is the heaviest step in the algorithm. For example, comparing two elements in sorting, comparing two records in searching, or doing a division in a number based algorithm.

Example 20: Checking an Employee Registration Number

Suppose we want to check if a registration number is present in a list of employee records. The algorithm compares the target number with each record one by one, and stops when it matches or the list ends. Here, the basic operation is the comparison between two numbers. If the list has n records, the algorithm does at most n comparisons.

Algorithm: Check-Registration(records, target)

```
1: for i = 1 to records.length do
2:     if records[i] = target then
3:         return true
4:     end if
5: end for
6: return false
```

Estimating Running Time from Operation Count

We can write the number of basic operations needed for input size n as $C(n)$. We also have c_{op}, which is the time taken for one basic operation. Using these, we can estimate the running time as

$$T(n) \approx c_{op} \cdot C(n)$$

This gives us a rough idea of running time, without depending on the actual machine timing.

Example 21: Estimating Time for Record Check

If checking one record takes c_{op} time, and we have n records, then in the worst case, the total time is approximately $c_{op} \cdot n$. If n becomes double, the total time also becomes roughly double.

Ignoring Constant Factors

When we study efficiency, we ignore constant multipliers. This is because these constants depend on the hardware and the way the program is written. Also, when input size becomes very large, these constants do not matter much, because the growth pattern of $C(n)$ matters more than the fixed constant.

Example 22: Doubling Input Size

If an algorithm takes $5n$ steps and another takes $2n$ steps, both grow in the same straight line pattern as n increases. So, we say both algorithms have the same growth pattern, and we ignore the constants 5 and 2 while comparing their efficiency.

1.3.4 Order of Growth

When we compare algorithms, the difference does not show much when the input is small, because even a slow algorithm finishes fast for small input. The real difference is seen when the input size becomes very large. So, we study how the running time grows as the input size grows.

Order of Growth

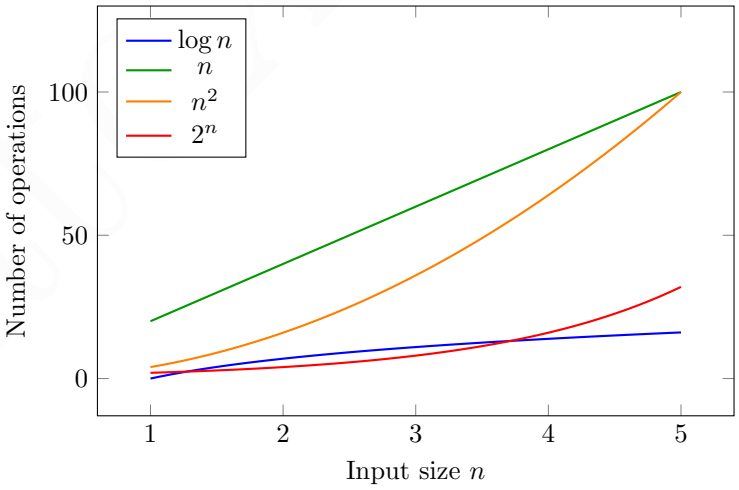
If an algorithm's operation count grows slowly as input size increases, the algorithm stays usable even for large problems. But if the operation count grows very fast, the algorithm becomes useless for large input. Because of this, we classify algorithms by their order of growth, which means how fast their cost increases with input size.

Example 23: Walking vs Running on a Long Road

For a short distance, walking and running may take almost the same time. But for a very long road, running clearly takes much less time than walking. In the same way, for small input, slow and fast algorithms look similar, but for large input, the difference becomes clear.

We compare algorithms based on how the number of basic operations grows with input size n, not based on actual running time for small inputs. Different growth rates behave very differently:

- Logarithmic growth ($\log n$) increases very slowly.
- Linear growth (n) increases in direct proportion to input size.
- Polynomial growth (n^2, n^3) increases fast.
- Exponential (2^n) and factorial ($n!$) growth increase extremely fast, becoming unusable very quickly.



Base of Logarithm Does Not Matter

When we write $\log n$, the base of the log does not matter in efficiency analysis. This is because changing the base only changes the value by a constant factor, as shown below

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Since we ignore constant factors in this kind of analysis, we simply write $\log n$ without mentioning the base.

Exponential and Factorial Growth

Algorithms with exponential growth (2^n) or factorial growth ($n!$) are practical only for very small input sizes. As n increases even a little, their running time increases extremely fast, making them unusable for large input.

Example 24: Doubling Input Size Effect

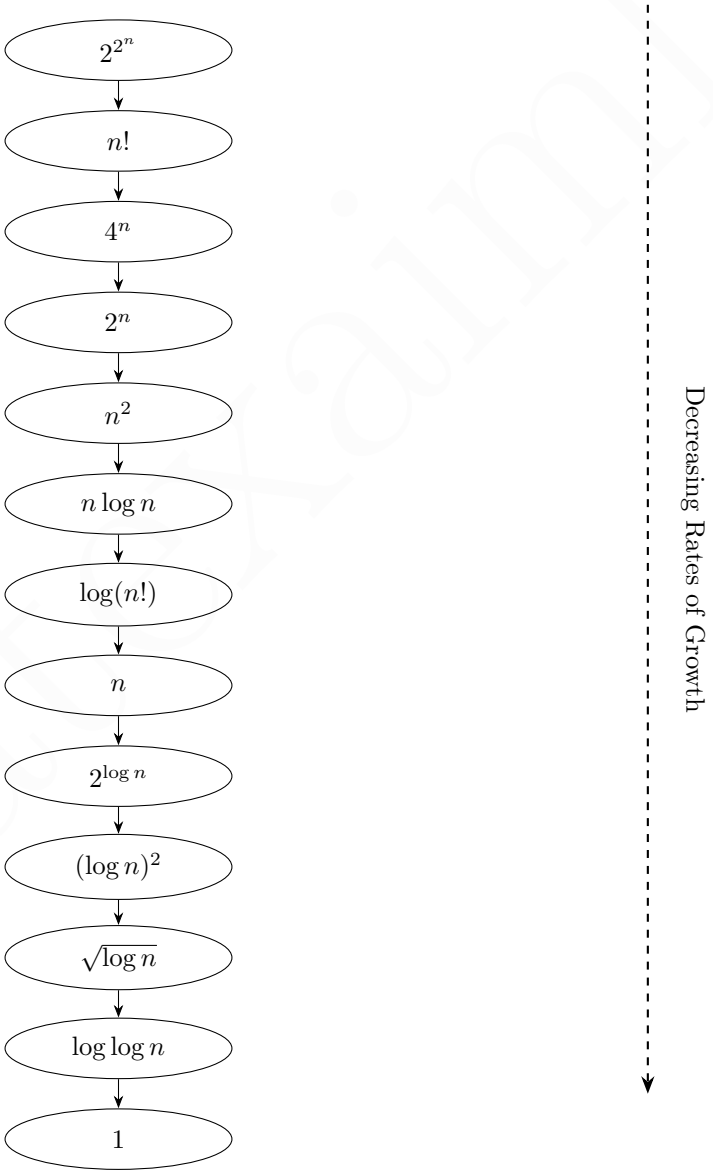
Suppose an algorithm needs $f(n) = n^2$ operations. If input size becomes double, from n to $2n$, then

$$f(2n) = (2n)^2 = 4n^2$$

So the work becomes four times more.
Now suppose another algorithm needs $g(n) = n$ operations. If input size becomes double, then

$$g(2n) = 2n$$

So the work becomes only double.
This shows that even if both algorithms look similar for small input, the algorithm with n^2 becomes much costlier than the algorithm with n, as input size grows large.



Example 25: Order of Growth Practice 1

Arrange the following functions in increasing order of growth:

$$n, \quad \log n, \quad n^2, \quad 1$$

Order: $1 < \log n < n < n^2$

Example 26: Order of Growth Practice 2

Arrange the following functions in increasing order of growth:

$$n!, \quad 2^n, \quad n^3, \quad n \log n$$

Order: $n \log n < n^3 < 2^n < n!$

Example 27: Order of Growth Practice 3

Arrange the following functions in increasing order of growth:

$$\sqrt{n}, \quad n, \quad \log n, \quad n^2 \log n$$

Order: $\log n < \sqrt{n} < n < n^2 \log n$

Example 28: Order of Growth Practice 4

Arrange the following functions in increasing order of growth:

$$2^n, \quad n^{10}, \quad 10^n, \quad \log_2 n$$

Order: $\log_2 n < n^{10} < 2^n < 10^n$

Example 29: Order of Growth Practice 5

Arrange the following functions in increasing order of growth:

$$n \log n, \quad n^2, \quad n, \quad 2^{\log n}$$

Hint: simplify $2^{\log n}$ first using log identity, base 2.
Order: $n = 2^{\log n} < n \log n < n^2$

1.4 Asymptotic Notations

1.4.1 Asymptotic Efficiency of Algorithms

Asymptotic Efficiency Analysis

When input size becomes very large, small parts of the running time formula stop mattering. Constant numbers and small extra terms become tiny compared to the main growing part. So, we only look at the main part that grows with n. This way of studying growth is called asymptotic efficiency analysis.

Example 30: Buying in Bulk

If we buy 2 apples, paying 5 rupees extra for a bag does matter. But if we buy 1000 apples, that same 5 rupees extra does not matter much. In the same way, small constants in running time do not matter when input size is very large.

What Asymptotic Analysis Tells Us

Asymptotic analysis tells us how well an algorithm scales, that means how it behaves as input grows bigger and bigger. It does not give us the exact running time in seconds. It only tells us the pattern of growth, which is the

main thing we need while choosing an algorithm for large input.

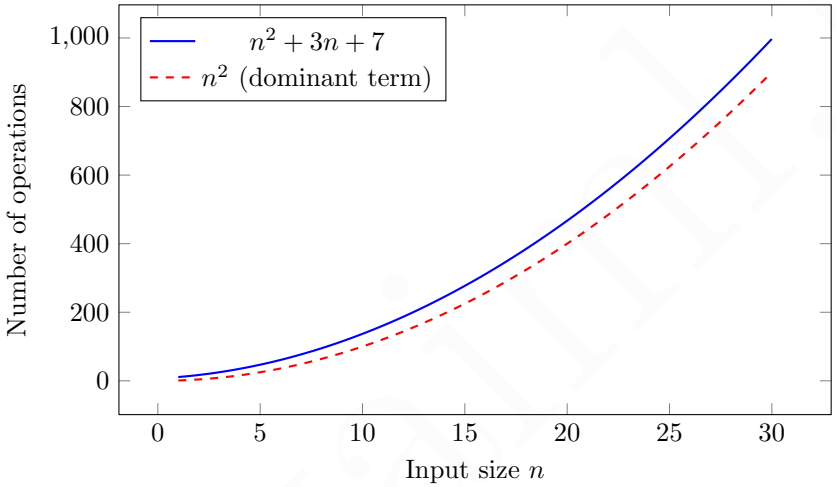
Why We Look at the Dominant Term

A running time formula can have many terms, like $n^2 + 3n + 7$. As n becomes very large, the term n^2 becomes much bigger than $3n$ and 7. So, $3n$ and 7 become useless in comparison. We only keep the dominant term, which is the term that grows the fastest, and we ignore the rest.

Mathematical view: Consider $f(n) = n^2 + 3n + 7$. We can divide every term by n^2 to see their relative size as n grows:

$$\frac{f(n)}{n^2} = 1 + \frac{3}{n} + \frac{7}{n^2}$$

As $n \to \infty$, the terms $\frac{3}{n}$ and $\frac{7}{n^2}$ become almost 0. So, $f(n)$ behaves almost exactly like n^2 for large n. This is why we say $f(n)$ has order of growth n^2.



Notice in the graph that the solid curve ($n^2 + 3n + 7$) and the dashed curve (n^2) come very close together as n becomes large. This shows visually why we can ignore the smaller terms and constants.

Comparing Algorithms Using Order of Growth

When we compare two algorithms, we do not check their exact step count. We check their order of growth. An algorithm with a slower growing order of growth is better for large input, even if it looks worse for small input.

Example 31: Linear vs Quadratic Growth

Suppose Algorithm A takes $5n + 20$ steps, and Algorithm B takes $n^2 + 3n$ steps. Algorithm A grows linearly, and Algorithm B grows quadratically. For small n, both may take similar steps. But for large n, Algorithm A becomes much faster than Algorithm B, because linear growth is slower than quadratic growth.

Example 32: Numerical Comparison: Algorithm A vs Algorithm B

Suppose Algorithm A needs $4n + 10$ operations, and Algorithm B needs n^2 operations.
Case 1: Small input, $n = 5$

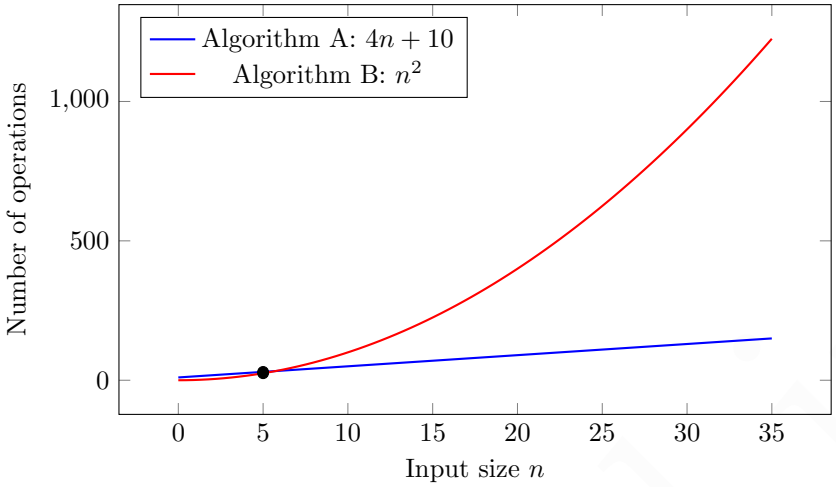
$$A = 4(5) + 10 = 30, \qquad B = 5^2 = 25$$

Here, B is actually doing fewer operations than A.
Case 2: Large input, $n = 1000$

$$A = 4(1000) + 10 = 4010, \qquad B = 1000^2 = 1{,}000{,}000$$

Now, A is doing far fewer operations than B.

This shows that for small input, the algorithm with a worse order of growth can still look better. But as input size grows large, the algorithm with the better order of growth always wins. So, asymptotically, Algorithm A is the better choice.



The graph shows the crossover point, where Algorithm B's curve crosses above Algorithm A's curve. Before this point, B looks better. After this point, A is always better, and the gap keeps increasing as n grows.

Example 33: Practice: Identify the Dominant Term

For each running time expression, identify the dominant term, that means the order of growth.

1. $T(n) = 7n^3 + 50n^2 + 100$ $\rightarrow$ dominant term is n^3
2. $T(n) = 2^n + n^{10}$ $\rightarrow$ dominant term is 2^n
3. $T(n) = 20n \log n + 5n$ $\rightarrow$ dominant term is $n \log n$
4. $T(n) = n! + 2^n + n^5$ $\rightarrow$ dominant term is $n!$

1.4.2 O-Notation (Big-Oh Notation)

Big-Oh Notation

Big-Oh notation gives us an upper bound on how fast a function grows. It tells us that a function $f(n)$ does not grow faster than another function $g(n)$, once n becomes large enough, except for a constant multiplying factor. Formally, we say $f(n) = O(g(n))$ if we can find two positive numbers c and n_0, such that

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

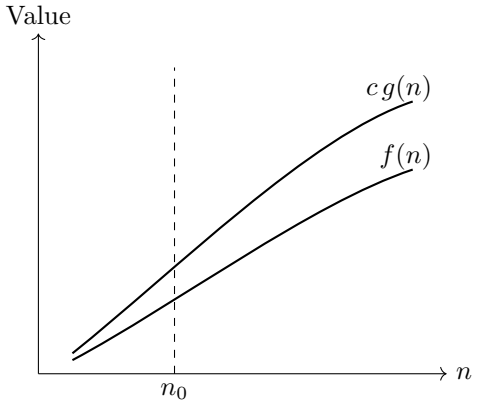
In simple words, after some point n_0, the function $f(n)$ always stays below or on the curve $c \cdot g(n)$. We do not care what happens before n_0, we only care about large n.

Example 34: Speed Limit Board

Think of a speed limit board on a highway that says maximum speed is 80. A car may go slow sometimes, fast sometimes, but it never crosses 80, once it is on the highway. Big-Oh is similar, $c \cdot g(n)$ acts like a speed limit, and $f(n)$ never crosses it, after point n_0.

Maths intuition: Why do we allow a constant c and a starting point n_0?

- The constant c is needed because we do not care about exact values, we only care about growth pattern. So even if $f(n)$ is a bit bigger than $g(n)$, like $f(n) = 5g(n)$, we can still say $f(n) = O(g(n))$, by simply choosing $c = 5$ or more.
- The point n_0 is needed because for small n, a function may behave randomly, or one function may be bigger than other by chance. We only care about behavior when n becomes large, so we allow the inequality to fail for small n, before n_0.



In this picture, before n_0, the curve $f(n)$ might cross above $c \cdot g(n)$, that is fine. But after n_0, $f(n)$ always stays below $c \cdot g(n)$. This is exactly what the Big-Oh definition demands.

Properties of Big-Oh Notation

- $O(g(n))$ is actually a set of functions, not a single function. Writing $f(n) = O(g(n))$ really means $f(n)$ belongs to this set.
- The constants c and n_0 are not fixed or unique. Many valid choices of c and n_0 can work for the same proof.
- Big-Oh ignores small input behavior. Only what happens as n becomes very large matters.
- A slower growing function can still belong to a faster growing function's class. For example, $n \in O(n^2)$, because n never grows faster than n^2. This does not mean they grow at the same rate, it only means n does not cross n^2 beyond a constant factor.
- We assume $f(n)$ is non-negative for large n, so that the comparison $f(n) \leq c \cdot g(n)$ makes proper sense.

Example 35: Finding Upper Bound: $f(n) = 3n + 8$

We want to show $f(n) = O(n)$.
We need to find $c > 0$ and n_0 such that

$$0 \leq 3n + 8 \leq c \cdot n \quad \text{for all } n \geq n_0$$

For $n \geq 8$, we know $8 \leq n$. So,

$$3n + 8 \leq 3n + n = 4n$$

This means $3n + 8 \leq 4n$ for all $n \geq 8$.
Choosing $c = 4$ and $n_0 = 8$, the condition holds. So,

$$3n + 8 = O(n)$$

Example 36: Finding Upper Bound: $f(n) = n^2 + 1$

We want to show $f(n) = O(n^2)$.
We need $c > 0$ and n_0 such that

$$0 \leq n^2 + 1 \leq c \cdot n^2 \quad \text{for all } n \geq n_0$$

For $n \geq 1$, we know $1 \leq n^2$. So,

$$n^2 + 1 \leq n^2 + n^2 = 2n^2$$

This means $n^2 + 1 \leq 2n^2$ for all $n \geq 1$.
Choosing $c = 2$ and $n_0 = 1$, the condition holds. So,

$$n^2 + 1 = O(n^2)$$

Example 37: Finding Upper Bound: $f(n) = 100n + 5$

We want to show $f(n) = O(n^2)$.
For $n \geq 5$, we know $5 \leq n$. So,

$$100n + 5 \leq 100n + n = 101n$$

Also, for $n \geq 1$, we know $n \leq n^2$, so $101n \leq 101n^2$.
Combining, for $n \geq 5$,

$$100n + 5 \leq 101n^2$$

Choosing $c = 101$ and $n_0 = 5$, the condition holds. So,

$$100n + 5 = O(n^2)$$

Example 38: Practice 1: $f(n) = 5n^2 + 3n$

Prove that $f(n) = O(n^2)$.
Solution: For $n \geq 1$, we know $3n \leq 3n^2$. So,

$$5n^2 + 3n \leq 5n^2 + 3n^2 = 8n^2$$

Choosing $c = 8$ and $n_0 = 1$, we get $5n^2 + 3n = O(n^2)$.

Example 39: Practice 2: $f(n) = n^3 + 50n^2$

Prove that $f(n) = O(n^3)$.
Solution: For $n \geq 50$, we know $50n^2 \leq n \cdot n^2 = n^3$. So,

$$n^3 + 50n^2 \leq n^3 + n^3 = 2n^3$$

Choosing $c = 2$ and $n_0 = 50$, we get $n^3 + 50n^2 = O(n^3)$.

Example 40: Practice 3: $f(n) = 7$

Prove that $f(n) = O(1)$.
Solution: Here $f(n) = 7$ stays fixed, it does not grow with n. So,

$$7 \leq 7 \cdot 1$$

Choosing $c = 7$ and $n_0 = 1$, we get $7 = O(1)$. This shows constant functions are written as $O(1)$.

Example 41: Practice 4: $f(n) = n \log n + n$

Prove that $f(n) = O(n \log n)$.
Solution: For $n \geq 2$, we know $\log n \geq 1$, so $n \leq n \log n$. So,

$$n \log n + n \leq n \log n + n \log n = 2n \log n$$

Choosing $c = 2$ and $n_0 = 2$, we get $n \log n + n = O(n \log n)$.

Example 42: Practice 5: $f(n) = 2n^2 + 3n \log n + 10$

Prove that $f(n) = O(n^2)$.
Solution: For $n \geq 2$, we know $\log n \leq n$, so $3n \log n \leq 3n^2$. Also for $n \geq 1$, $10 \leq 10n^2$. So,

$$2n^2 + 3n \log n + 10 \leq 2n^2 + 3n^2 + 10n^2 = 15n^2$$

Choosing $c = 15$ and $n_0 = 2$, we get $2n^2 + 3n \log n + 10 = O(n^2)$.

1.4.3 Ω-Notation (Big-Omega Notation)

Big-Omega Notation

Big-Omega notation gives us a lower bound on how fast a function grows. It tells us that a function $f(n)$ grows at least as fast as another function $g(n)$, once n becomes large enough, except for a constant multiplying factor. Formally, we say $f(n) = \Omega(g(n))$ if we can find two positive numbers c and n_0, such that

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

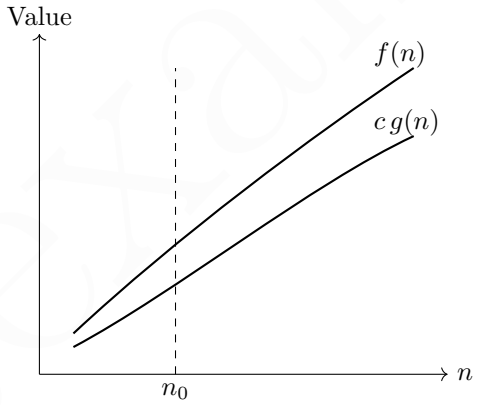
In simple words, after some point n_0, the function $f(n)$ always stays above or on the curve $c \cdot g(n)$. We do not care what happens before n_0, we only care about large n.

Example 43: Minimum Wage Floor

Think of a rule that says a worker must be paid at least 500 rupees per day. The actual pay can be more, some days much more, but it never falls below this floor amount. Big-Omega is similar, $c \cdot g(n)$ acts like a floor, and $f(n)$ never falls below it, after point n_0.

Maths intuition: Big-Oh and Big-Omega are like opposite directions of the same idea.

- In Big-Oh, $f(n)$ stays *below* $c \cdot g(n)$, so $g(n)$ acts as a ceiling.
- In Big-Omega, $f(n)$ stays *above* $c \cdot g(n)$, so $g(n)$ acts as a floor.
- The constant c lets us shrink $g(n)$ a little, so even if $f(n)$ is somewhat smaller than $g(n)$, we can still find a smaller version of $g(n)$ that $f(n)$ stays above.
- The point n_0 lets us ignore small n, where the inequality might not hold by chance.



In this picture, after n_0, the curve $f(n)$ always stays above $c \cdot g(n)$. Before n_0, this may or may not be true, and we do not worry about that part.

Properties of Big-Omega Notation

- $\Omega(g(n))$ is a set of functions, not a single function. Writing $f(n) = \Omega(g(n))$ means $f(n) \in \Omega(g(n))$.
- The constants c and n_0 are not fixed or unique. Many valid pairs can work for the same proof.
- Only behavior as $n \to \infty$ matters. Small input values are ignored.
- If $f(n)$ grows faster than $g(n)$, then $f(n)$ can still belong to $\Omega(g(n))$. For example, $n^2 \in \Omega(n)$, because n^2 eventually grows at least as fast as n. This does not mean they grow at the same rate, it only means n^2 never falls below n, up to a constant factor.
- We assume $f(n)$ and $g(n)$ are non-negative for large n, so the comparison makes proper sense.

Example 44: Showing Lower Bound: $f(n) = 3n^2 + 2n$

We want to show $f(n) = \Omega(n^2)$.
We need $c > 0$ and n_0 such that

$$0 \leq c \cdot n^2 \leq 3n^2 + 2n \quad \text{for all } n \geq n_0$$

Since $2n \geq 0$ for all $n \geq 0$,

$$3n^2 + 2n \geq 3n^2$$

This means $3n^2 + 2n \geq 3n^2$ for all $n \geq 0$.
Choosing $c = 3$ and $n_0 = 1$, the condition holds. So,

$$3n^2 + 2n = \Omega(n^2)$$

Example 45: Practice 1: $f(n) = n^2 - 3n$

Prove that $f(n) = \Omega(n^2)$.
Solution: For $n \geq 6$, we know $3n \leq \frac{1}{2}n^2$. So,

$$n^2 - 3n \geq n^2 - \frac{1}{2}n^2 = \frac{1}{2}n^2$$

Choosing $c = \frac{1}{2}$ and $n_0 = 6$, we get $n^2 - 3n = \Omega(n^2)$.

Example 46: Practice 2: $f(n) = 7n + 2$

Prove that $f(n) = \Omega(n)$.
Solution: Since $2 \geq 0$ for all n,

$$7n + 2 \geq 7n$$

Choosing $c = 7$ and $n_0 = 1$, we get $7n + 2 = \Omega(n)$.

Example 47: Practice 3: $f(n) = n^3 + 4n$

Prove that $f(n) = \Omega(n^3)$.
Solution: Since $4n \geq 0$ for all $n \geq 0$,

$$n^3 + 4n \geq n^3$$

Choosing $c = 1$ and $n_0 = 1$, we get $n^3 + 4n = \Omega(n^3)$.

Example 48: Practice 4: $f(n) = n \log n$

Prove that $f(n) = \Omega(n)$.
Solution: For $n \geq 2$, we know $\log n \geq 1$. So,

$$n \log n \geq n \cdot 1 = n$$

Choosing $c = 1$ and $n_0 = 2$, we get $n \log n = \Omega(n)$.

Example 49: Practice 5: $f(n) = 2^n$

Prove that $f(n) = \Omega(n^2)$.
Solution: It is known that 2^n grows much faster than any polynomial like n^2, for large n. We can check small

values to find a safe starting point. At $n = 6$, $2^n = 64$ and $n^2 = 36$, so $2^n \geq n^2$ here. As n increases further, 2^n grows much faster than n^2, so the inequality keeps holding.

$$2^n \geq n^2 \quad \text{for all } n \geq 6$$

Choosing $c = 1$ and $n_0 = 6$, we get $2^n = \Omega(n^2)$.

1.4.4 Θ-Notation (Big-Theta Notation)

Big-Theta Notation

Big-Theta notation gives us a tight bound on how fast a function grows. It tells us that a function $f(n)$ grows neither faster nor slower than $g(n)$, once n becomes large enough, except for constant factors. This means $f(n)$ and $g(n)$ grow at the same rate.
Formally, we say $f(n) = \Theta(g(n))$ if we can find three positive numbers c_1, c_2, and n_0, such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$

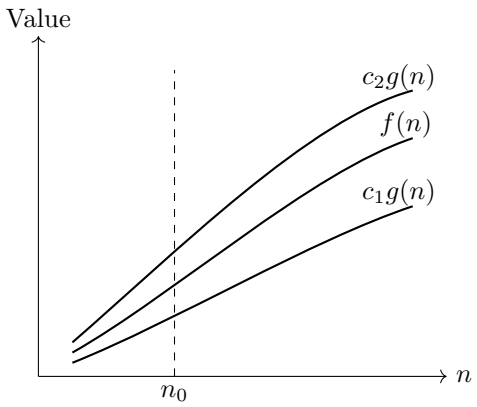
In simple words, after some point n_0, the function $f(n)$ is always sandwiched between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$. It never goes too high above $g(n)$, and never goes too low below $g(n)$.

Example 50: Road Between Two Walls

Think of a narrow road with a wall on the left side and a wall on the right side. A car driving on this road can never go beyond the left wall, and can never go beyond the right wall, it always stays between them. Big-Theta is similar, $f(n)$ always stays between the lower wall $c_1 \cdot g(n)$ and the upper wall $c_2 \cdot g(n)$.

Maths intuition: Big-Theta is simply Big-Oh and Big-Omega combined together.

- Big-Oh alone tells us $f(n)$ does not grow faster than $g(n)$, that is an upper bound only.
- Big-Omega alone tells us $f(n)$ does not grow slower than $g(n)$, that is a lower bound only.
- Big-Theta needs both conditions to be true at the same time. So if we already know $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then automatically $f(n) = \Theta(g(n))$.
- This is the strongest and most informative notation, because it tells us the exact growth class of $f(n)$, not just an upper limit or just a lower limit.



In this picture, after n_0, the function $f(n)$ is trapped between the lower curve $c_1g(n)$ and the upper curve $c_2g(n)$. This trapped region is the visual meaning of Big-Theta.

Properties of Big-Theta Notation

- $\Theta(g(n))$ is a set of functions, not a single function. Writing $f(n) = \Theta(g(n))$ means $f(n) \in \Theta(g(n))$.
- Big-Theta needs two inequalities together, $c_1g(n) \leq f(n)$ and $f(n) \leq c_2g(n)$. Both must be true at the same

time, for the same n_0.

- Only behavior for large n matters. Small n does not affect membership in $\Theta(g(n))$.
- Constant multiplying factors do not change the Theta class. For example, $an^2 \in \Theta(n^2)$, for any positive constant a.
- If $f(n) = \Theta(g(n))$, then $f(n)$ and $g(n)$ grow at the same speed, neither one beats the other by more than a constant factor, as n becomes large.

Example 51: Showing Tight Bound: $f(n) = 2n^2 + 3n + 4$

We want to show $f(n) = \Theta(n^2)$.
Step 1: Upper Bound. For $n \geq 1$, we know $3n \leq 3n^2$ and $4 \leq 4n^2$. So,

$$f(n) \leq 2n^2 + 3n^2 + 4n^2 = 9n^2$$

This gives us $c_2 = 9$.
Step 2: Lower Bound. Since all terms in $f(n)$ are non-negative for $n \geq 1$,

$$f(n) \geq 2n^2$$

This gives us $c_1 = 2$.
Step 3: Combine. For all $n \geq 1$,

$$2n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$$

Choosing $c_1 = 2$, $c_2 = 9$, and $n_0 = 1$, both conditions hold together. So,

$$2n^2 + 3n + 4 = \Theta(n^2)$$

Example 52: Practice 1: $f(n) = 5n + 2$

Prove that $f(n) = \Theta(n)$.
Upper bound: For $n \geq 1$, $5n + 2 \leq 5n + 2n = 7n$, so $c_2 = 7$.
Lower bound: Since $2 \geq 0$, $5n + 2 \geq 5n$, so $c_1 = 5$.
Choosing $c_1 = 5$, $c_2 = 7$, $n_0 = 1$, we get $5n + 2 = \Theta(n)$.

Example 53: Practice 2: $f(n) = n^2 + n$

Prove that $f(n) = \Theta(n^2)$.
Upper bound: For $n \geq 1$, $n \leq n^2$, so $n^2 + n \leq n^2 + n^2 = 2n^2$, giving $c_2 = 2$.
Lower bound: Since $n \geq 0$, $n^2 + n \geq n^2$, giving $c_1 = 1$.
Choosing $c_1 = 1$, $c_2 = 2$, $n_0 = 1$, we get $n^2 + n = \Theta(n^2)$.

Example 54: Practice 3: $f(n) = 3n^3 + 2n^2 + n$

Prove that $f(n) = \Theta(n^3)$.
Upper bound: For $n \geq 1$, $2n^2 \leq 2n^3$ and $n \leq n^3$, so

$$f(n) \leq 3n^3 + 2n^3 + n^3 = 6n^3, \quad c_2 = 6$$

Lower bound: Since all terms are non-negative, $f(n) \geq 3n^3$, so $c_1 = 3$.
Choosing $c_1 = 3$, $c_2 = 6$, $n_0 = 1$, we get $3n^3 + 2n^2 + n = \Theta(n^3)$.

Example 55: Practice 4: $f(n) = \frac{1}{2}n^2 - 3n$

Prove that $f(n) = \Theta(n^2)$.
Lower bound: For $n \geq 12$, $3n \leq \frac{1}{4}n^2$, so

$$f(n) \geq \frac{1}{2}n^2 - \frac{1}{4}n^2 = \frac{1}{4}n^2, \quad c_1 = \frac{1}{4}$$

Upper bound: Since $3n \geq 0$ does not add value here, simply $f(n) \leq \frac{1}{2}n^2$, so $c_2 = \frac{1}{2}$.
Choosing $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, $n_0 = 12$, we get $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Example 56: Practice 5: $f(n) = n \log n + n$

Prove that $f(n) = \Theta(n \log n)$.
Upper bound: For $n \geq 2$, $\log n \geq 1$, so $n \leq n \log n$, giving

$$f(n) \leq n \log n + n \log n = 2n \log n, \quad c_2 = 2$$

Lower bound: Since $n \geq 0$, $f(n) \geq n \log n$, giving $c_1 = 1$.
Choosing $c_1 = 1$, $c_2 = 2$, $n_0 = 2$, we get $n \log n + n = \Theta(n \log n)$.

1.4.5 Relationship Between O, Ω, and Θ

How the Three Notations Connect

Big-Oh, Big-Omega, and Big-Theta are not separate, unrelated ideas. They are connected to each other through some simple rules. These rules help us prove one notation using another, and help us understand them better as a single family.

Transitivity

If $f(n) = \Theta(g(n))$, and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$. This same rule also works for O and Ω. In simple words, if A matches B in growth, and B matches C in growth, then A also matches C in growth.

Example 57: Chain of Friends

If Ravi is as tall as Suresh, and Suresh is as tall as Mahesh, then Ravi is also as tall as Mahesh. Transitivity in notations works the same way, just with growth rate instead of height.

Reflexivity

Any function is always related to itself in the same notation. That means $f(n) = O(f(n))$. This same rule works for Θ and Ω also. In simple words, a function always grows at the same rate as itself, which is obviously true.

Symmetry

This rule is only for Big-Theta. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$. In simple words, if $f(n)$ grows at the same rate as $g(n)$, then $g(n)$ also grows at the same rate as $f(n)$. This makes sense because Big-Theta means equal growth, and equal growth works both ways.

Transpose Symmetry

This rule connects Big-Oh and Big-Omega together. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$. In simple words, if $f(n)$ does not grow faster than $g(n)$, then automatically $g(n)$ does not grow slower than $f(n)$. These are just two ways of saying the same thing, from different sides.

Theta as Combination of Oh and Omega

For any two functions $f(n)$ and $g(n)$ that stay non-negative for large n,

$$f(n) = \Theta(g(n)) \quad \text{if and only if} \quad f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

This means, to prove Big-Theta, we do not need a brand new method. We simply prove Big-Oh first, then prove Big-Omega separately, and if both are true, Big-Theta is automatically true.

Example 58: Upper Wall and Lower Wall Together

Think back to the road between two walls example used for Big-Theta. The left wall is the lower bound, which is Big-Omega. The right wall is the upper bound, which is Big-Oh. If we show the car never crosses the left wall, and never crosses the right wall, we have shown the car always stays on this road, which is exactly Big-Theta.

Example 59: Proving Theta Using Oh and Omega Together: $f(n) = an^2 + bn + c$

We want to show that for $a > 0$, $f(n) = an^2 + bn + c = \Theta(n^2)$.
Step 1: Prove Big-Oh. For $n \geq 1$, we know $|bn| \leq |b|n^2$ and $|c| \leq |c|n^2$. So,

$$f(n) \leq an^2 + |b|n^2 + |c|n^2 = (a + |b| + |c|)n^2$$

This shows $f(n) = O(n^2)$.
Step 2: Prove Big-Omega. Since $a > 0$, the term an^2 is the strongest term, and it grows faster than bn and c. So, for large enough n, the smaller terms cannot pull the value below half of an^2. There exists some n_0 such that

$$an^2 + bn + c \geq \frac{a}{2}n^2 \quad \text{for all } n \geq n_0$$

This shows $f(n) = \Omega(n^2)$.
Step 3: Combine. Since $f(n)$ is both $O(n^2)$ and $\Omega(n^2)$, by the rule connecting the three notations,

$$an^2 + bn + c = \Theta(n^2)$$

Example 60: Practice 1: Using Transitivity

Given $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$, with $h(n) = n^2$, show $f(n) = \Theta(h(n))$.
Solution: Since $f(n) = \Theta(g(n))$, and $g(n) = \Theta(h(n))$ (because both $g(n)$ and $h(n)$ are $\Theta(n^2)$), by transitivity, $f(n) = \Theta(h(n))$.

Example 61: Practice 2: Using Transpose Symmetry

Given $n = O(n^2)$, what can we say about n^2 using transpose symmetry?
Solution: By transpose symmetry, since $n = O(n^2)$, it directly follows that $n^2 = \Omega(n)$.

Example 62: Practice 3: Proving Theta Using Both Bounds

Show that $f(n) = 4n^2 - 5n$ is $\Theta(n^2)$.
Upper bound: For $n \geq 1$, $f(n) \leq 4n^2$, so $f(n) = O(n^2)$.
Lower bound: For $n \geq 3$, $5n \leq 2n^2$, so $f(n) \geq 4n^2 - 2n^2 = 2n^2$, giving $f(n) = \Omega(n^2)$.
Since both hold, $f(n) = \Theta(n^2)$.

Example 63: Practice 4: Symmetry Check

If $5n^2 = \Theta(n^2)$, what can we say about n^2 in relation to $5n^2$?
Solution: By symmetry of Big-Theta, since $5n^2 = \Theta(n^2)$, it directly follows that $n^2 = \Theta(5n^2)$.

Example 64: Practice 5: Chain Using Transitivity

Given $n \log n = \Theta(n \log n)$, and $n \log n = \Theta(2n \log n)$, and $2n \log n = \Theta(3n \log n)$, what can we conclude about $n \log n$ and $3n \log n$?
Solution: Using transitivity twice, since $n \log n = \Theta(2n \log n)$ and $2n \log n = \Theta(3n \log n)$, we get $n \log n = \Theta(3n \log n)$.

1.4.6 Little-o Notation

Little-o Notation

Little-o notation also gives an upper bound, like Big-Oh. But there is a big difference. In Big-Oh, $f(n)$ can grow at the same rate as $g(n)$. In little-o, $f(n)$ must grow strictly slower than $g(n)$, not at the same rate.
Formally, we say $f(n) = o(g(n))$ if, for every positive constant c, no matter how small, we can find a value n_0 such that

$$0 \leq f(n) < c \cdot g(n) \quad \text{for all } n \geq n_0$$

The key difference from Big-Oh is this, in little-o, the inequality must hold for *every* possible $c > 0$, even very tiny values of c, not just for some one fixed c.

Example 65: Younger Brother and Older Brother

Think of two brothers walking. Big-Oh is like saying the younger brother never walks faster than the older brother, but they could be walking at the exact same speed. Little-o is like saying the younger brother always walks strictly slower, he can never catch up to the same speed, no matter how slow we let the older brother go.

Properties of Little-o Notation

- Little-o gives a strict upper bound, meaning $f(n)$ grows strictly slower than $g(n)$.
- Unlike Big-Oh, here the condition must hold for every $c > 0$, not just one specific c. This makes little-o a stronger and more demanding condition than Big-Oh.
- If $f(n) = o(g(n))$, then automatically $f(n) = O(g(n))$ also holds. But the opposite is not always true.
- A function is never little-o of itself. That means $f(n) \neq o(f(n))$, because $f(n)$ cannot grow strictly slower than itself.
- A simple way to check little-o is using limits,

  $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

  If this limit becomes exactly 0, then $f(n) = o(g(n))$.

Example 66: Checking Little-o: $f(n) = n$, $g(n) = n^2$

We check whether $n = o(n^2)$, using the limit method.

$$\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0$$

Since the limit is 0, we conclude

$$n = o(n^2)$$

This makes sense, because n grows much slower than n^2, as n becomes large.

Example 67: Checking Little-o: $f(n) = n^2$, $g(n) = n^2$

We check whether $n^2 = o(n^2)$.

$$\lim_{n\to\infty} \frac{n^2}{n^2} = 1$$

Since the limit is 1, not 0, we conclude

$$n^2 \neq o(n^2)$$

This confirms that little-o does not allow same speed growth, only strictly slower growth is allowed.

1.4.7 Little-ω Notation

Little-ω Notation

Little-ω notation gives a lower bound, like Big-Omega. But just like little-o, there is a strictness added here. In Big-Omega, $f(n)$ can grow at the same rate as $g(n)$. In little-ω, $f(n)$ must grow strictly faster than $g(n)$, not at the same rate.
Formally, we say $f(n) = \omega(g(n))$ if, for every positive constant c, no matter how large, we can find a value n_0 such that

$$0 \leq c \cdot g(n) < f(n) \quad \text{for all } n \geq n_0$$

Here also, the condition must hold for *every* possible $c > 0$, not just one fixed c, which makes little-ω a stronger condition than Big-Omega.

Example 68: Faster Runner

Think of a runner who is always strictly faster than another runner, no matter how much we slow down the comparison runner. This is little-ω, a strict, always faster relationship. Big-Omega only says our runner is never slower, they could even run at the same speed sometimes.

Properties of Little-ω Notation

- Little-ω gives a strict lower bound, meaning $f(n)$ grows strictly faster than $g(n)$.
- The condition must hold for every $c > 0$, not just one specific c, making it stronger than Big-Omega.
- If $f(n) = \omega(g(n))$, then automatically $f(n) = \Omega(g(n))$ also holds. But the opposite is not always true.
- A function is never little-ω of itself, since $f(n)$ cannot grow strictly faster than itself.
- A simple way to check little-ω is using limits,

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

  If this limit becomes infinity, then $f(n) = \omega(g(n))$.
- Little-o and little-ω are reverse of each other, just like Big-Oh and Big-Omega. If $f(n) = o(g(n))$, then $g(n) = \omega(f(n))$.

Example 69: Checking Little-ω: $f(n) = n^2$, $g(n) = n$

We check whether $n^2 = \omega(n)$, using the limit method.

$$\lim_{n\to\infty} \frac{n^2}{n} = \lim_{n\to\infty} n = \infty$$

Since the limit is infinity, we conclude

$$n^2 = \omega(n)$$

This makes sense, because n^2 grows much faster than n, as n becomes large.

Example 70: Checking Little-ω: $f(n) = 2^n$, $g(n) = n^{10}$

We check whether $2^n = \omega(n^{10})$.

$$\lim_{n\to\infty} \frac{2^n}{n^{10}} = \infty$$

This is true because exponential growth always beats polynomial growth, eventually, no matter how large the power is. So,

$$2^n = \omega(n^{10})$$

Example 71: Practice 1

Check whether $\log n = o(n)$.
Solution:

$$\lim_{n\to\infty} \frac{\log n}{n} = 0$$

Since the limit is 0, we get $\log n = o(n)$.

Example 72: Practice 2

Check whether $n^3 = o(n^2)$.
Solution:

$$\lim_{n\to\infty} \frac{n^3}{n^2} = \lim_{n\to\infty} n = \infty$$

Since the limit is not 0, we get $n^3 \neq o(n^2)$. In fact, $n^3 = \omega(n^2)$ instead.

Example 73: Practice 3

Check whether $n! = \omega(2^n)$.
Solution: As n becomes large, factorial growth becomes much faster than exponential growth.

$$\lim_{n\to\infty} \frac{n!}{2^n} = \infty$$

So, $n! = \omega(2^n)$.

Example 74: Practice 4

Check whether $5n^2 = o(n^2)$.
Solution:

$$\lim_{n\to\infty} \frac{5n^2}{n^2} = 5$$

Since the limit is a fixed constant, not 0, we get $5n^2 \neq o(n^2)$. Instead, $5n^2 = \Theta(n^2)$.

Example 75: Practice 5

Check whether $n \log n = \omega(n)$.
Solution:

$$\lim_{n\to\infty} \frac{n \log n}{n} = \lim_{n\to\infty} \log n = \infty$$

Since the limit is infinity, we get $n \log n = \omega(n)$.

1.4.8 Relationship Between O, Ω, Θ, o, and ω

Two Families of Notations

We can group all five notations into two families. One family compares growth allowing equal growth also, this is the non-strict family, containing O, Ω, and Θ. The other family compares growth not allowing equal growth, this is the strict family, containing o and ω. Little-o is the strict version of Big-Oh, and little-ω is the strict version of Big-Omega. There is no strict version of Big-Theta, because Theta already means exact equal growth.

Example 76: Less Than vs Strictly Less Than

Think of two ways to compare numbers. We can say $a \leq b$, which allows a to be equal to b. Or we can say $a < b$, which does not allow them to be equal. Big-Oh is like $\leq$, and little-o is like $<$. The same pattern goes for Big-Omega and little-ω.

Limit Based Summary

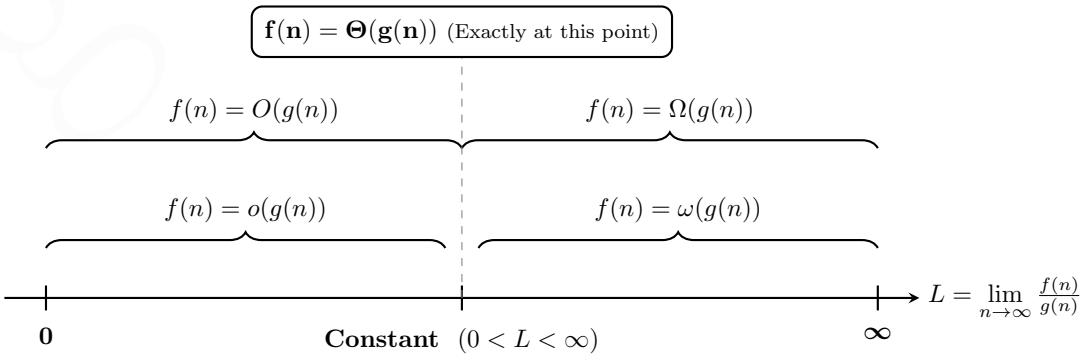
All five notations can be checked using the limit

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Depending on the value of L, we know which notation applies.

- If $L = 0$, then $f(n) = o(g(n))$, and also $f(n) = O(g(n))$.
- If L is a positive finite constant, then $f(n) = \Theta(g(n))$, and also $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- If $L = \infty$, then $f(n) = \omega(g(n))$, and also $f(n) = \Omega(g(n))$.

This single limit test tells us exactly which notations are true and which are false, for a given pair $f(n)$ and $g(n)$.



How the Notations Are Related

- $f(n) = o(g(n))$ always implies $f(n) = O(g(n))$, because strictly slower also means not faster. But O does not imply o, since O allows equal growth also.
- $f(n) = \omega(g(n))$ always implies $f(n) = \Omega(g(n))$, because strictly faster also means not slower. But Ω does not imply ω, since Ω allows equal growth also.
- $f(n) = \Theta(g(n))$ if and only if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ are true together.
- $f(n) = o(g(n))$ and $f(n) = \omega(g(n))$ can never both be true for the same pair $f(n)$ and $g(n)$, because one says strictly slower and other says strictly faster, they cannot both happen.
- $f(n) = o(g(n))$ and $f(n) = \Theta(g(n))$ can never both be true together, because o means strictly slower, but Θ means equal growth.

Example 77: Connection Through Transpose Symmetry

Just like $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$, the same flipping rule works for the strict family also. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$. For example, since $n = o(n^2)$, by this flipping rule, we directly get $n^2 = \omega(n)$, without checking the limit again.

Relation Between the Big Family and the Little Family

The big notations (O, Ω, Θ) and the little notations (o, ω) are not independent, they sit inside each other in a strict-inside-non-strict pattern.

- Every function pair classified by o is automatically also classified by O. So $o(g(n))$ is a smaller, stricter set sitting entirely inside $O(g(n))$.
- Every function pair classified by ω is automatically also classified by Ω. So $\omega(g(n))$ sits entirely inside $\Omega(g(n))$.
- $O(g(n))$ can be split into exactly two non-overlapping parts: the part where growth is strictly slower, which is $o(g(n))$, and the part where growth is exactly equal, which is $\Theta(g(n))$. That is,

$$O(g(n)) = o(g(n)) \cup \Theta(g(n))$$

- Similarly, $\Omega(g(n))$ splits into $\omega(g(n))$ and $\Theta(g(n))$:

$$\Omega(g(n)) = \omega(g(n)) \cup \Theta(g(n))$$

- Because of this split, if we already know $f(n) = O(g(n))$ but we can rule out $f(n) = \Theta(g(n))$, we can directly conclude the stronger fact $f(n) = o(g(n))$. The same logic applies to Ω and ω.

Example 78: Using the Big-Little Split: $f(n) = n^2$, $g(n) = n^3$

We already know from Practice 1 that $f(n) = O(n^3)$. Now check whether $f(n) = \Theta(n^3)$ is true. Since

$$\lim_{n\to\infty} \frac{n^2}{n^3} = 0 \neq \text{positive constant}$$

Θ is ruled out. Since $O(g(n)) = o(g(n)) \cup \Theta(g(n))$, and Θ is ruled out, we directly conclude

$$f(n) = o(n^3)$$

without needing to separately confirm the strict limit condition.

Transitivity of Little-o and Little-ω

The strict notations are transitive. If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then

$$f(n) = o(h(n))$$

Similarly, if $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then

$$f(n) = \omega(h(n))$$

Think of it as a chain of strictly-slower-than relationships. If A is strictly slower than B, and B is strictly slower than C, then A must be strictly slower than C as well, the strictness carries through the whole chain.

Example 79: Using Transitivity of Little-o

We know $\log n = o(n)$, and we know $n = o(n^2)$. By transitivity of o, without computing any new limit, we can directly conclude

$$\log n = o(n^2)$$

By the transpose symmetry rule, this also tells us $n^2 = \omega(\log n)$.

Example 80: Full Classification: $f(n) = 3n^2$, $g(n) = n^2$

$$\lim_{n\to\infty} \frac{3n^2}{n^2} = 3$$

The limit is a positive constant, so $f(n) = \Theta(n^2)$. This also means $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$ are both true. But $f(n) \neq o(n^2)$ and $f(n) \neq \omega(n^2)$, since the limit is not 0 and not ∞.

Example 81: Full Classification: $f(n) = n \log n$, $g(n) = n^2$

$$\lim_{n\to\infty} \frac{n \log n}{n^2} = \lim_{n\to\infty} \frac{\log n}{n} = 0$$

The limit is 0, so $f(n) = o(n^2)$. This also means $f(n) = O(n^2)$ is true. But $f(n) \neq \Theta(n^2)$, $f(n) \neq \Omega(n^2)$, and $f(n) \neq \omega(n^2)$.

Example 82: Practice 1: $f(n) = n^2$, $g(n) = n^3$

Find the limit and classify the relationship.
Solution:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = 0$$

So, $f(n) = o(n^3)$, which also means $f(n) = O(n^3)$.

Example 83: Practice 2: $f(n) = 2^n$, $g(n) = 3^n$

Find the limit and classify the relationship.
Solution:

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$$

So, $f(n) = o(3^n)$, which also means $f(n) = O(3^n)$.

Example 84: Practice 3: $f(n) = n!$, $g(n) = n^n$

Find the limit and classify the relationship.
Solution: Since n^n multiplies n copies of n, but $n!$ multiplies decreasing numbers below n, n^n grows faster.

$$\lim_{n\to\infty} \frac{n!}{n^n} = 0$$

So, $f(n) = o(n^n)$, which also means $f(n) = O(n^n)$.

Example 85: Practice 4: $f(n) = 7n^2 + 2n$, $g(n) = n^2$

Find the limit and classify the relationship.
Solution:

$$\lim_{n\to\infty} \frac{7n^2 + 2n}{n^2} = 7$$

Since the limit is a positive constant, $f(n) = \Theta(n^2)$, which also means $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$.

Example 86: Practice 5: $f(n) = n^3$, $g(n) = n^2 \log n$

Find the limit and classify the relationship.
Solution:

$$\lim_{n\to\infty} \frac{n^3}{n^2 \log n} = \lim_{n\to\infty} \frac{n}{\log n} = \infty$$

So, $f(n) = \omega(n^2 \log n)$, which also means $f(n) = \Omega(n^2 \log n)$.

Additive Property (Sum Rule)

When two functions are added together, the sum is bounded by whichever growth rate is faster. Formally, if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) + f_2(n) = O\big(\max(g_1(n), g_2(n))\big)$$

The same idea holds for Ω: if $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) + f_2(n) = \Omega\big(\max(g_1(n), g_2(n))\big)$. A useful special case, often called the **Max Rule**, applies when $f(n)$ and $g(n)$ are both asymptotically non-negative:

$$f(n) + g(n) = \Theta\big(\max(f(n), g(n))\big)$$

The intuition is simple: when you add a slow-growing term to a fast-growing term, the slow term eventually becomes insignificant, so the sum behaves like the faster term alone.

Example 87: Applying the Sum Rule: $f(n) = n^3 + n^2 \log n$

Here the two terms are n^3 and $n^2 \log n$. Since n^3 grows faster than $n^2 \log n$, the max rule tells us

$$n^3 + n^2 \log n = \Theta(n^3)$$

We do not need to analyze both terms separately in later steps of an algorithm's analysis; only the dominant term n^3 matters for the overall growth rate.

Multiplicative Property (Product Rule)

When two functions are multiplied, their bounds multiply together as well. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) \cdot f_2(n) = O\big(g_1(n) \cdot g_2(n)\big)$$

The same pattern holds for Ω and Θ:

- If $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$, then $f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$.
- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then $f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$.

This property is especially useful when analyzing nested loops, since the work done by an outer loop and an inner loop multiplies together.

Example 88: Applying the Product Rule: Nested Loop Cost

Suppose an outer loop runs $\Theta(n)$ times, and for each outer iteration, an inner loop runs $\Theta(\log n)$ times. Treating these as $f_1(n) = \Theta(n)$ and $f_2(n) = \Theta(\log n)$, the product rule gives the total work as

$$f_1(n) \cdot f_2(n) = \Theta(n \log n)$$

This is exactly why we multiply the bounds of nested loops instead of adding them.

Constant Multiple Property

Multiplying a function by a fixed positive constant does not change its asymptotic classification. If $f(n) = O(g(n))$, then for any constant $c > 0$,

$$c \cdot f(n) = O(g(n))$$

The same holds for Ω, Θ, o, and ω. This is why constant factors are always dropped when we state a growth rate, since $2n$, $50n$, and $0.001n$ are all $\Theta(n)$.

Example 89: Dropping Constant Factors

Suppose an algorithm performs $5n^2 + 3n$ operations. Using the sum rule, this is $\Theta(n^2)$. Using the constant multiple property, we also see that a related algorithm performing $500n^2$ operations, or even $0.5n^2$ operations, is still $\Theta(n^2)$. Only the shape of growth matters, not the constant in front of it.

Example 90: Practice 6: Big-Little Split

Given $f(n) = n \log n$ and $g(n) = n^2$, we already know $f(n) = O(n^2)$. Use the big-little split to decide whether $f(n) = o(n^2)$.
Solution: Check if Θ holds:

$$\lim_{n \to \infty} \frac{n \log n}{n^2} = 0 \neq \text{positive constant}$$

Θ is ruled out, so since $O(g(n)) = o(g(n)) \cup \Theta(g(n))$, we conclude $f(n) = o(n^2)$.

Example 91: Practice 7: Transitivity of Little-ω

Given $n^2 = \omega(n \log n)$ and $n \log n = \omega(n)$, what can we conclude?
Solution: By transitivity of ω,

$$n^2 = \omega(n)$$

Example 92: Practice 8: Sum Rule

Classify $f(n) = 2^n + n^{100}$ using the max rule.
Solution: Since 2^n grows faster than any polynomial, including n^{100},

$$2^n + n^{100} = \Theta(2^n)$$

Example 93: Practice 9: Product Rule

An algorithm has an outer loop of $\Theta(n^2)$ iterations, and inside each iteration it calls a subroutine that costs $\Theta(n)$. Find the total cost.
Solution: By the product rule,

$$\Theta(n^2) \cdot \Theta(n) = \Theta(n^3)$$

1.5 Best Case, Worst Case and Average Case

Why We Need These Three Cases

For many algorithms, the running time does not stay fixed, it changes depending on the actual input given, even if the input size n is the same. So, we study three different cases to fully understand the algorithm's behavior, best case, worst case, and average case.

Example 94: Searching for a Friend in a Queue

Suppose we are searching for our friend in a queue of people, by checking one by one from the front. If our friend is the first person, we find them fast. If our friend is the last person, we take a long time. If our friend can be anywhere with equal chance, the time taken on average lies somewhere in between.

Worst Case

Worst case means the input that makes the algorithm take the maximum possible time, among all inputs of the same size n. We study worst case because it gives us a guarantee, the algorithm will never take more time than this, no matter what input comes.

Example 95: Worst Case in Linear Search

Suppose we search for a target value in an array of n elements, using linear search, checking one by one. The worst case happens when the target is at the last position, or the target is not present at all in the array. In this case, the algorithm has to check all n elements, so the worst case time is $\Theta(n)$.

Best Case

Best case means the input that makes the algorithm take the minimum possible time, among all inputs of the same size n. Best case is less useful for guarantees, because we cannot expect lucky input every time, but it tells us the fastest the algorithm can ever be.

Example 96: Best Case in Linear Search

In the same linear search example, the best case happens when the target is found at the very first position. Here, the algorithm only checks one element, so the best case time is $\Theta(1)$.

Average Case

Average case means the expected time taken by the algorithm, averaged over all possible inputs of size n, assuming some probability distribution on the inputs. Average case is usually harder to calculate than best case or worst case, because it needs assumptions about how likely each input is.

Example 97: Average Case in Linear Search

In linear search, assume the target is equally likely to be at any position from 1 to n, or not present at all. On average, the target is found around the middle of the array. So, the average case time works out to be around $\frac{n+1}{2}$ comparisons, which is still $\Theta(n)$.

Which Case Matters Most

In most algorithm analysis, we focus mainly on worst case, because it gives a safe upper limit, useful for real world guarantees. Average case is used when we have good knowledge about the typical input pattern. Best case is rarely used alone, because it can be misleading, an algorithm may look fast in best case but actually be slow in general.

Example 98: Best, Worst, Average Together: Searching in an Array of Size 10

Suppose we have an array of 10 elements, and we search for a target using linear search.

- Best case, target is at position 1, only 1 comparison needed.
- Worst case, target is at position 10, or not present, 10 comparisons needed.
- Average case, assuming target is equally likely anywhere, around 5 to 6 comparisons needed on average.

Example 99: Practice 1: Searching in Sorted Array

For a sorted array, where we stop searching as soon as we cross the target value (since the array is sorted), identify the best and worst case for linear search.
Solution: Best case, target found at position 1, only 1 comparison. Worst case, target is bigger than all elements, or array is fully scanned, requiring n comparisons.

Example 100: Practice 2: Bubble Sort on Already Sorted Array

Identify the best case for bubble sort, when the input array is already sorted, and the algorithm can detect this and stop early.
Solution: Best case happens when the array is already sorted, requiring only 1 pass with $n-1$ comparisons to confirm no swaps are needed, giving best case $\Theta(n)$.

Example 101: Practice 3: Bubble Sort on Reverse Sorted Array

Identify the worst case for bubble sort, when the input array is sorted in reverse order.
Solution: Worst case happens when every pair of adjacent elements needs swapping, in every pass, requiring around $\frac{n(n-1)}{2}$ comparisons, giving worst case $\Theta(n^2)$.

Example 102: Practice 4: Searching in a Hash Table

Identify a situation that represents the worst case for searching in a hash table.
Solution: Worst case happens when many keys collide and land in the same bucket, forming a long chain. Searching then becomes like a linear search inside that chain, giving worst case $\Theta(n)$, even though average case is usually $\Theta(1)$.

Example 103: Practice 5: Quicksort Best vs Worst Case

Identify the best case and worst case input pattern for quicksort, based on how the pivot splits the array.
Solution: Best case happens when the pivot always splits the array into two equal halves, giving $\Theta(n \log n)$. Worst case happens when the pivot always picks the smallest or largest element, splitting the array very unevenly, giving $\Theta(n^2)$.

1.6 Relation Between O, Ω, Θ and Best, Worst, Average Case

These Are Two Completely Different Ideas

Many students mix up these two ideas, but they are not the same thing, and they do not replace each other. Best case, worst case, and average case tell us **which input** we are talking about. Big-Oh, Big-Omega, and Big-Theta tell us **how we are measuring the growth** of the running time, for whichever case we picked. These two ideas work on different axes, one is about input type, the other is about bound type.

Example 104: Two Different Questions

Think of two separate questions about a runner. Question one is, "Which race are we talking about, his fastest race, his slowest race, or his typical race?" This is like best case, worst case, average case. Question two is, "Are we giving his exact time, just an upper limit, or just a lower limit?" This is like Theta, Big-Oh, Big-Omega. Both questions are needed together to fully describe his performance.

Why Confusion Happens

The confusion happens because people are taught a shortcut, "worst case is Big-Oh, best case is Big-Omega". This shortcut is wrong, and it only works in some examples by coincidence. The truth is, we can use O, Ω, or Θ on the worst case. We can also use O, Ω, or Θ on the best case. We can also use them on the average case. All nine combinations are mathematically possible.

Example 105: All Nine Combinations Are Valid Statements

- Worst case can be written as $O(n^2)$, or $\Omega(n^2)$, or $\Theta(n^2)$.
- Best case can be written as $O(n)$, or $\Omega(n)$, or $\Theta(n)$.
- Average case can be written as $O(n \log n)$, or $\Omega(n \log n)$, or $\Theta(n \log n)$.

None of these combinations are wrong by definition. O, Ω, Θ are just tools to express a bound, and we can apply this tool to any one of the three cases.

Why the Shortcut Seems to Work Sometimes

In many simple textbook examples, people only state the tightest possible bound, which is Θ, for both best case and worst case. When they write the worst case using O instead of Θ, by habit, it looks like "worst case equals Big-Oh". But this is just a common writing style, it is not a rule. We could just as easily write the worst case using Θ, since Θ gives more information than O alone.

Example 106: Linear Search Explained Properly

Consider linear search on an array of size n.

- Worst case happens when the target is not found, requiring exactly n comparisons. Since this is an exact bound, we should correctly say worst case is $\Theta(n)$, not just $O(n)$. Saying $O(n)$ is also technically true, but it is a weaker, less informative statement.
- Best case happens when the target is found at the first position, requiring exactly 1 comparison. So best case is $\Theta(1)$.
- Average case, assuming equal probability of target at any position, requires around $\frac{n+1}{2}$ comparisons, so average case is also $\Theta(n)$.

Notice here, Θ was used for all three cases, because in each case, we know the exact tight bound. O and Ω were not needed here, but they could have been used too, just with less precision.

When We Actually Need O or Ω Alone

Sometimes, we do not know the exact tight bound for a case, we only know a limit from one side. In such situations, we are forced to use only O or only Ω, because Θ is not available to us yet.

Example 107: A Case Where Only O Is Known

Suppose we have a complicated algorithm, and we have proved that in the worst case, it never takes more than n^2 steps, but we have not yet proved the exact lower limit for worst case. Here, we can only correctly say worst case is $O(n^2)$. We cannot yet say worst case is $\Theta(n^2)$, until someone also proves a matching $\Omega(n^2)$ lower bound for the worst case.

Clear Rule to Always Remember

- First, decide which case you are analyzing, best, worst, or average. This depends only on the input pattern, nothing to do with notation.
- Second, once the case is fixed, find a formula for the number of operations, for that case only.
- Third, apply O, Ω, or Θ on that formula, exactly the same way we apply it on any normal function. The case you picked earlier has no special rule for which notation to use.
- If you know the exact tight formula for that case, always prefer writing Θ, since it carries the most information. Use O alone only when you genuinely have just an upper limit, and use Ω alone only when you genuinely have just a lower limit.

Example 108: Common Wrong Statement vs Correct Statement

Wrong belief: "Worst case is always Big-Oh, and best case is always Big-Omega."
Correct understanding: Worst case, best case, and average case are simply different inputs we are studying. For each of these, once we get the exact step count, we can describe it using O, Ω, or Θ, based on how much we have proved, not based on which case it is.

Example 109: Practice 1

For linear search, the worst case requires exactly n comparisons. Write this using all three notations correctly.
Solution: Worst case is $\Theta(n)$. It is also correct, though less precise, to say worst case is $O(n)$ and worst case is $\Omega(n)$, since $\Theta(n)$ already implies both.

Example 110: Practice 2

For insertion sort, the best case (already sorted array) requires exactly $n - 1$ comparisons. Write this using the most informative notation.
Solution: Best case is $\Theta(n)$, since we know the exact count, $n - 1$, which is tightly bounded both above and below by constant multiples of n.

Example 111: Practice 3

Suppose for some algorithm, only an upper limit of n^3 has been proved for the average case, and no lower limit is known yet. How should this be written correctly?
Solution: Average case is $O(n^3)$ only. It would be wrong to write $\Theta(n^3)$, because we have not proved a matching lower bound yet.

Example 112: Practice 4

A student says, "Best case of bubble sort is $\Omega(n)$, because best case always uses Omega." Explain why this reasoning is incorrect, even if the final answer is correct.

Solution: The reasoning is wrong, even though $\Omega(n)$ is true for the best case of bubble sort. The correct logic is, best case of bubble sort needs exactly $n-1$ comparisons, so it is $\Theta(n)$, which automatically makes $\Omega(n)$ true also. It is not true simply because it is a best case, it is true because of the actual exact analysis.

Example 113: Practice 5

For binary search on a sorted array, the worst case requires exactly $\log n$ comparisons. Write the best, worst, and average case using the most precise notation.
Solution: Best case is $\Theta(1)$, when the target is found at the middle on the first check. Worst case is $\Theta(\log n)$. Average case is also $\Theta(\log n)$, since on average, the search still takes a number of steps proportional to $\log n$.

1.7 Analysis of Non Recursive Algorithms

1.7.1 Useful Maths Formulas

Why We Need These Formulas

When we analyze a loop in an algorithm, we often get a sum of many terms, like $1+2+3+\cdots+n$. To find the exact order of growth, we need to convert this long sum into a simple formula. These maths formulas help us do that conversion quickly and correctly.

Sum of First n Natural Numbers

This formula gives the sum of all numbers from 1 to n.

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

This sum grows as $\Theta(n^2)$, even though we are adding n terms, because the terms themselves keep increasing.

Example 114: Applying Sum of Natural Numbers

Consider a loop where, in the first round we do 1 unit of work, in the second round 2 units, in the third round 3 units, and so on, till n rounds. The total work is

$$1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

which is $\Theta(n^2)$.

Sum of First n Squares

This formula gives the sum of squares of numbers from 1 to n.

$$\sum_{i=1}^{n} i^2 = 1^2+2^2+3^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$$

This sum grows as $\Theta(n^3)$.

Example 115: Applying Sum of Squares

Consider a loop where, in round i, we do i^2 units of work, for $i=1$ to n. The total work is

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

which is $\Theta(n^3)$.

Sum of a Geometric Series

This formula gives the sum of terms where each term is a fixed ratio r times the previous term.

$$\sum_{i=0}^{n} r^i = 1 + r + r^2 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}, \quad r \neq 1$$

If $r > 1$, this sum grows as $\Theta(r^n)$, because the last term dominates all the earlier terms combined. If $0 < r < 1$, the sum stays bounded by a constant, no matter how large n becomes.

Example 116: Applying Geometric Series with $r = 2$

Consider a loop where, in round i, we do 2^i units of work, for $i = 0$ to n. The total work is

$$1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

which is $\Theta(2^n)$. This matches the idea that the last term in a growing geometric series dominates the entire sum.

Sum of Harmonic Series

This formula gives the sum of reciprocals of numbers from 1 to n.

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n$$

This sum grows as $\Theta(\log n)$, even though we are adding n terms, because each term becomes smaller and smaller.

Example 117: Applying Harmonic Series

Consider an algorithm where, in round i, the work depends on $\frac{n}{i}$, for $i = 1$ to n. The total work is

$$\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} \approx n \ln n$$

which is $\Theta(n \log n)$.

Sum of Logarithms

This formula gives the sum of logarithm values from 1 to n, using the property that sum of logs equals log of product.

$$\sum_{i=1}^{n} \log i = \log(1 \cdot 2 \cdot 3 \cdots n) = \log(n!)$$

Using Stirling's approximation, $\log(n!)$ grows as $\Theta(n \log n)$.

Example 118: Applying Sum of Logarithms

Consider a loop where, in round i, we do $\log i$ units of work, for $i = 1$ to n. The total work is

$$\sum_{i=1}^{n} \log i = \log(n!) = \Theta(n \log n)$$

Sum Splitting Rule

A sum of two different parts can be split and solved separately, and then added together.

$$\sum_{i=1}^{n}(a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Also, a constant multiplying factor can be taken outside the sum.

$$\sum_{i=1}^{n} c \cdot a_i = c \sum_{i=1}^{n} a_i$$

Example 119: Applying Sum Splitting

Consider the sum $\sum_{i=1}^{n}(3i+5)$. We split this into two parts.

$$\sum_{i=1}^{n} 3i + \sum_{i=1}^{n} 5 = 3 \cdot \frac{n(n+1)}{2} + 5n$$

The dominant term here is $3 \cdot \frac{n(n+1)}{2}$, so the overall sum is $\Theta(n^2)$.

Example 120: Practice 1

Find the order of growth of $\sum_{i=1}^{n} 4i$.
Solution:

$$\sum_{i=1}^{n} 4i = 4 \cdot \frac{n(n+1)}{2} = 2n(n+1)$$

This is $\Theta(n^2)$.

Example 121: Practice 2

Find the order of growth of $\sum_{i=1}^{n} 3i^2$.
Solution:

$$\sum_{i=1}^{n} 3i^2 = 3 \cdot \frac{n(n+1)(2n+1)}{6} = \frac{n(n+1)(2n+1)}{2}$$

This is $\Theta(n^3)$.

Example 122: Practice 3

Find the order of growth of $\sum_{i=0}^{n} 3^i$.
Solution:

$$\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{3-1}$$

This is $\Theta(3^n)$.

Example 123: Practice 4

Find the order of growth of $\sum_{i=1}^{n}\left(i + \frac{1}{i}\right)$.

Solution: We split the sum into two parts.

$$\sum_{i=1}^{n} i + \sum_{i=1}^{n} \frac{1}{i} = \frac{n(n+1)}{2} + \ln n$$

The term $\frac{n(n+1)}{2}$ dominates $\ln n$, so the overall sum is $\Theta(n^2)$.

Example 124: Practice 5

Find the order of growth of $\sum_{i=1}^{n} \log(i^2)$.
Solution: Using log property, $\log(i^2) = 2\log i$.

$$\sum_{i=1}^{n} 2\log i = 2\sum_{i=1}^{n} \log i = 2\log(n!)$$

This is $\Theta(n \log n)$.

1.7.2 Mathematical Analysis of Nonrecursive Algorithms

Mathematical Analysis of Nonrecursive Algorithms

To find how good a nonrecursive algorithm is, we count how many times its basic operation runs, as a function of input size. This count is not guessed, it is built step by step, using a fixed plan, so that we do not miss anything and do not make mistakes.

General Plan

1. Decide on a parameter that represents the input size.
2. Find the basic operation of the algorithm, usually inside the innermost loop.
3. Check if the operation count depends only on input size, or also on the actual input values, that means check if we need best case, worst case, or average case.
4. Write a summation that counts how many times the basic operation runs.
5. Solve the summation exactly, or at least find its order of growth.

Example 125: Step by Step Walkthrough: Sum of First n Numbers

Consider this code, which adds numbers from 1 to n into a variable `sum`.

```
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum + i;
```

We now apply the general plan, one step at a time.
Step 1: Decide input size parameter. Here, the input is the number n itself. So, our size parameter is simply n.
Step 2: Identify the basic operation. The line `sum = sum + i;` is inside the loop, and it is the operation that repeats. This addition is our basic operation.
Step 3: Check dependency on input configuration. The loop always runs from $i = 1$ to $i = n$, no matter what value n has, there is no condition like searching for a target, where the count could change based on where something is found. So, here, the count depends only on n, we do not need separate best case, worst case, average case, the count is fixed for a given n.
Step 4: Set up the summation. The basic operation runs once for every value of i from 1 to n. So the count is

$$C(n) = \sum_{i=1}^{n} 1$$

Step 5: Solve the summation. Adding 1, exactly n times, gives

$$C(n) = n$$

So, the running time is $\Theta(n)$.

How We Decide the Bounds

In the above example, we got an exact value, $C(n) = n$, not just an upper or lower limit. Since we have the exact formula, we can confidently write $\Theta(n)$, because Θ needs both an upper bound and a lower bound, and here both bounds are the same exact function, n itself. If a problem only allows us to estimate an upper limit, like "the loop runs at most n times" without knowing the exact count, then we should only write $O(n)$. Bounds are decided based on how much we actually know about the exact behavior of the count, not by guesswork.

Example 126: Loops

The running time of a loop is exactly proportional to the running time of the statements inside the loop, multiplied by the number of times the loop runs.

```
for (i = 1; i <= n; i++)
    m = m + 2;   // constant time
```

Here, the loop runs exactly n times, and each time, a constant amount of work happens. Since this represents both an upper and a lower bound for the execution, the tight complexity is:

$$c \cdot n = \Theta(n)$$

Algorithm: Single-Loop-Count(n)

1: $m \leftarrow 0$
2: **for** $i = 1$ to n **do**
3: $\quad m \leftarrow m + 2$
4: **end for**

Example 127: Nested Loops

For nested loops, we analyze from inside out, and the total running time is the product of the sizes of all the loops.

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++)
        k = k + 1;   // constant time
}
```

The inner loop runs exactly n times for each of the n runs of the outer loop. Because this behavior is deterministic and constant across all inputs of size n, the tight running time is:

$$c \cdot n \cdot n = \Theta(n^2)$$

Example 128: Consecutive Statements

When statements come one after another, not nested, we simply add their time complexities.

```
x = x + 1;   // constant time
for (i = 1; i <= n; i++)
    m = m + 2;   // constant time
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++)
        k = k + 1;   // constant time
}
```

Total time is:

$$c_0 + c_1 n + c_2 n^2 = \Theta(n^2)$$

Here, the largest term, $c_2 n^2$, decides the final order of growth. The smaller terms do not matter for large n. Since the execution path always runs both loops, the runtime is tightly bounded by $\Theta(n^2)$.

Example 129: If-Then-Else Statements

For an if-else block, the worst-case time is the time for checking the condition, plus the larger of the two branches (the then-part or the else-part).

```
if (length() == 0)
    return false;      // constant
else {
    for (int n = 0; n < length(); n++) {
        if (list[n].equals(otherList.list[n]))
            return false;   // constant
    }
}
```

Here, the `then` part is constant time, but the worst-case scenario for the `else` part is that the loop runs to completion (taking time proportional to n). So, the worst-case complexity is tightly bounded by the `else` branch:

$$c_0 + c_1 + (c_2 + c_3)n = \Theta(n)$$

Example 130: Logarithmic Complexity

An algorithm takes $\Theta(\log n)$ time if its problem size gets divided or multiplied by a constant fraction each step, usually by 2.

```
for (i = 1; i <= n; i = i * 2)
    ;   // constant time
```

Here, i doubles every round, so after k rounds, $i = 2^k$. The loop stops precisely when 2^k becomes bigger than n, which means:

$$2^k = n \Rightarrow k = \log_2 n$$

Since the loop always executes exactly $\lfloor \log_2 n \rfloor + 1$ iterations, its tight bound is $\Theta(\log n)$.
Similarly, dividing by 2 every round also gives a tight bound of $\Theta(\log n)$:

```
for (i = n; i >= 1; i = i / 2)
    ;   // constant time
```

Example 131: Three Nested Loops with Different Patterns

Find the time complexity of this code.

```
void function(int n) {
    int i, j, k, count = 0;
    for (i = n/2; i <= n; i++)
        for (j = 1; j + n/2 <= n; j++)
            for (k = 1; k <= n; k = k * 2)
                count++;
}
```

Outer loop: i runs from $n/2$ to n, so the number of rounds is exactly $n - n/2 + 1 \approx n/2$, which is $\Theta(n)$.
Middle loop: The condition is $j + n/2 \leq n$, which simplifies to $j \leq n/2$. So this loop always runs exactly $n/2$ times, which is $\Theta(n)$.
Inner loop: k doubles each round $(1, 2, 4, 8, \dots)$ until it exceeds n. This always yields exactly $\lfloor \log_2 n \rfloor + 1$ rounds, which is $\Theta(\log n)$.

Combining all three loops: Since these nested loops execute unconditionally, we multiply their tight counts:

$$T(n) = \Theta(n) \times \Theta(n) \times \Theta(\log n) = \Theta(n^2 \log n)$$

Example 132: Practice 1: Single Loop with Step 2

Find the time complexity.

```
for (i = 1; i <= n; i = i + 2)
    count++;
```

Solution: The loop runs for $i = 1, 3, 5, \ldots$, up to n, which is exactly $\lceil n/2 \rceil$ rounds. Ignoring the constant factor $\frac{1}{2}$, the tight asymptotic bound is $\Theta(n)$.

Example 133: Practice 2: Two Independent Loops, Different Limits

Find the time complexity.

```
for (i = 1; i <= n; i++)
    count++;
for (j = 1; j <= n*n; j++)
    count++;
```

Solution: The first loop takes exactly $\Theta(n)$ steps and the second loop takes exactly $\Theta(n^2)$ steps. Summing them together, the dominant quadratic term establishes a tight limit, so the total time complexity is $\Theta(n^2)$.

Example 134: Practice 3: Nested Loop with Triangular Pattern

Find the time complexity.

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        count++;
```

Solution: For each i, the inner loop runs exactly i times. The total count is:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

which is $\Theta(n^2)$.

Example 135: Practice 4: Nested Loop with One Logarithmic Loop

Find the time complexity.

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j = j * 2)
        count++;
```

Solution: The outer loop runs exactly n times ($\Theta(n)$). The inner loop runs exactly $\lfloor \log_2 n \rfloor + 1$ times ($\Theta(\log n)$) for each outer round. Since the work is fully deterministic, the total time complexity is tightly bounded:

$$\Theta(n) \times \Theta(\log n) = \Theta(n \log n)$$

Example 136: Practice 5: If-Else with Loop on Both Sides

Find the worst-case time complexity.

```
if (n % 2 == 0) {
    for (i = 1; i <= n; i++)
```

```
        count++;
} else {
    for (i = 1; i <= n*n; i++)
        count++;
}
```

Solution: When n is even, the branch takes exactly $\Theta(n)$ time. When n is odd, the branch takes exactly $\Theta(n^2)$ time. The worst-case performance is strictly determined by the odd branch, meaning the worst-case time complexity is tightly bounded by $\Theta(n^2)$.

Example 137: Python Example 1: Single Loop

This function adds up all numbers from 0 to $n-1$.

Python implementation: single-loop-sum.py

```python
def sum_numbers(n):
    total = 0
    for i in range(n):
        total = total + i    # basic operation
    return total
```

Analysis: The loop runs exactly n times, and each time, one addition happens. So,

$$C(n) = \sum_{i=0}^{n-1} 1 = n$$

This gives $\Theta(n)$.

Example 138: Python Example 2: Nested Loop, Both Depending on n

This function counts how many pairs (i, j) exist, where both i and j range from 0 to $n-1$.

Python implementation: nested-loop-pairs.py

```python
def count_pairs(n):
    count = 0
    for i in range(n):
        for j in range(n):
            count = count + 1    # basic operation
    return count
```

Analysis: The outer loop runs n times, and for every round of the outer loop, the inner loop also runs n times. So,

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = n \cdot n = n^2$$

This gives $\Theta(n^2)$.

Example 139: Python Example 3: Triangular Nested Loop

This function counts pairs (i, j) where j only goes up to i, not all the way to n.

Python implementation: triangular-loop.py

```python
def count_triangular(n):
    count = 0
    for i in range(n):
```

```
        for j in range(i):
            count = count + 1    # basic operation
    return count
```

Analysis: For a fixed i, inner loop runs i times. So, total count is

$$C(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

This gives $\Theta(n^2)$, the same order as a full nested loop, just with a smaller constant.

Example 140: Python Example 4: Loop with Doubling Step

This function counts how many times we can double a number, starting from 1, before it crosses n.

Python implementation: doubling-loop.py

```
def count_doublings(n):
    count = 0
    i = 1
    while i <= n:
        count = count + 1    # basic operation
        i = i * 2
    return count
```

Analysis: The value of i follows the pattern $1, 2, 4, 8, \ldots$, until it crosses n. If the loop runs k times, then $2^{k-1} \leq n$, so $k \approx \log_2 n$. This gives $\Theta(\log n)$.

Example 141: Python Example 5: Consecutive Statements with Mixed Complexity

This function has a simple loop, followed by a nested loop, one after another, not inside each other.

Python implementation: consecutive-statements.py

```
def mixed_work(n):
    total = 0

    for i in range(n):          # simple loop
        total = total + i

    for i in range(n):          # nested loop
        for j in range(n):
            total = total + 1

    return total
```

Analysis: The first loop takes $\Theta(n)$ steps. The second, nested loop takes $\Theta(n^2)$ steps. Since these two parts run one after another, we add them together.

$$C(n) = n + n^2$$

For large n, the n^2 term dominates the n term, so the overall complexity is $\Theta(n^2)$.

1.8 Recurrence Relations

What Is a Recurrence Relation

A recurrence relation is an equation that defines the running time of an algorithm in terms of its own running time on a smaller input. It is used for recursive algorithms, because a recursive algorithm calls itself on a smaller piece of the problem, and the total time depends on the time of these smaller calls, plus the extra work done outside the calls.

Example 142: Folding a Paper

Suppose folding a paper of size n takes some time, and folding always means first folding a paper of half the size, then doing one quick fold action on top. The time to fold size n depends on the time to fold size $n/2$, plus a small constant time for the final fold. This is exactly the idea behind a recurrence relation.

General Form of a Recurrence

A recurrence relation usually has two parts. One part is the recursive part, which tells the cost in terms of a smaller input. The other part is the base case, which tells the cost when input becomes very small, usually a constant.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad T(1) = c$$

Here, a is the number of recursive calls made, $\frac{n}{b}$ is the size of each smaller call, and $f(n)$ is the extra work done outside the recursive calls, like dividing the problem or combining results.

Example 143: Recurrence for Merge Sort

Merge sort splits an array of size n into two halves, sorts each half recursively, and then merges them. The recurrence relation is

$$T(n) = 2T\left(\frac{n}{2}\right) + cn, \qquad T(1) = c$$

Here, 2 recursive calls are made, each on half the size, and cn is the cost of merging.

Why We Need to Solve Recurrences

A recurrence relation only gives the running time indirectly, in terms of itself. To actually know the order of growth, like $\Theta(n \log n)$ or $\Theta(n^2)$, we need to solve the recurrence, that means converting it into a normal formula in terms of n only, without T appearing on the right side.

Methods to Solve a Recurrence

There are three common methods used to solve a recurrence relation.

- Substitution Method, where we guess the answer, and then prove it correct using mathematical induction.
- Recursion Tree Method, where we draw the recursive calls as a tree, and add up the work done at every level of the tree.
- Master Theorem, which is a ready-made formula, that directly gives the answer for recurrences of a specific common shape.

1.8.1 Substitution Method

Substitution Method

In the substitution method, we first guess a formula for $T(n)$, based on intuition or pattern. Then, we substitute this guessed formula back into the recurrence, and check using mathematical induction, whether the guess actually satisfies the recurrence, for all values of n.

Example 144: Substitution Example 1: $T(n) = 2T(n/2) + n$

Step 1: Guess the answer. Based on similar problems, we guess

$$T(n) \leq c \cdot n \log n, \quad \text{for some constant } c > 0$$

Step 2: Assume the guess is true for smaller values. Assume for $n/2$,

$$T\left(\frac{n}{2}\right) \leq c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right)$$

Step 3: Substitute into the recurrence.

$$T(n) \leq 2\left(c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right)\right) + n = c \cdot n \log\left(\frac{n}{2}\right) + n$$

$$= c \cdot n(\log n - 1) + n = c \cdot n \log n - c \cdot n + n$$

Step 4: Check the guess. We want

$$c \cdot n \log n - c \cdot n + n \leq c \cdot n \log n$$

This simplifies to $-c \cdot n + n \leq 0$, that means $c \geq 1$. So, choosing $c \geq 1$ works.
Step 5: Conclude. Since a valid c was found, $T(n) = O(n \log n)$. A similar argument with $\geq$ gives $T(n) = \Omega(n \log n)$. So, $T(n) = \Theta(n \log n)$.

Example 145: Substitution Example 2: $T(n) = T(n/2) + 1$

Step 1: Guess the answer. We guess

$$T(n) \leq c \cdot \log n, \quad \text{for some constant } c > 0$$

Step 2: Assume the guess is true for $n/2$.

$$T\left(\frac{n}{2}\right) \leq c \cdot \log\left(\frac{n}{2}\right)$$

Step 3: Substitute into the recurrence.

$$T(n) \leq c \cdot \log\left(\frac{n}{2}\right) + 1 = c(\log n - 1) + 1 = c \log n - c + 1$$

Step 4: Check the guess. We want

$$c \log n - c + 1 \leq c \log n$$

This simplifies to $-c + 1 \leq 0$, that means $c \geq 1$. So, choosing $c \geq 1$ works.
Step 5: Conclude. So, $T(n) = O(\log n)$. A similar argument gives $T(n) = \Omega(\log n)$. So, $T(n) = \Theta(\log n)$.

Example 146: Substitution Example 3: $T(n) = 2T(n/2) + n^2$

Step 1: Guess the answer. We guess

$$T(n) \leq c \cdot n^2, \quad \text{for some constant } c > 0$$

Step 2: Assume the guess is true for $n/2$.

$$T\left(\frac{n}{2}\right) \leq c \cdot \left(\frac{n}{2}\right)^2 = c \cdot \frac{n^2}{4}$$

Step 3: Substitute into the recurrence.

$$T(n) \leq 2\left(c \cdot \frac{n^2}{4}\right) + n^2 = \frac{c \cdot n^2}{2} + n^2$$

Step 4: Check the guess. We want

$$\frac{c \cdot n^2}{2} + n^2 \leq c \cdot n^2$$

This simplifies to $n^2 \leq \frac{c \cdot n^2}{2}$, that means $c \geq 2$. So, choosing $c \geq 2$ works.
Step 5: Conclude. So, $T(n) = O(n^2)$. A similar argument gives $T(n) = \Omega(n^2)$. So, $T(n) = \Theta(n^2)$.

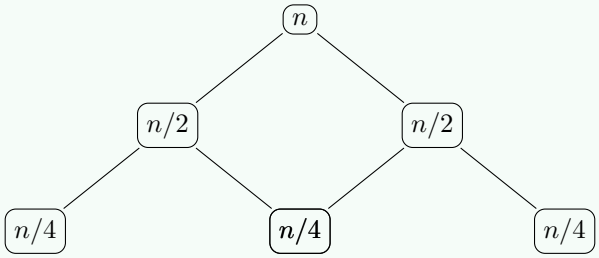
1.8.2 Recursion Tree Method

Recursion Tree Method

In the recursion tree method, we draw a tree, where the root is the original problem of size n, and the children of each node are the smaller recursive calls made by that node. Each node also carries the extra cost $f(n)$ done at that level, not counting its children's cost. We then add up the cost of every level, and add all levels together, to get the total cost.

Example 147: Recursion Tree Example 1: $T(n) = 2T(n/2) + n$

Step 1: Draw the root. The root is a problem of size n, with cost n at that level.
Step 2: Draw the children. The root makes 2 calls of size $n/2$ each. Level cost is $2 \times \frac{n}{2} = n$.
Step 3: Continue expanding. Each of those 2 nodes makes 2 calls of size $n/4$. Now 4 nodes exist, each cost $n/4$, level total $4 \times \frac{n}{4} = n$.

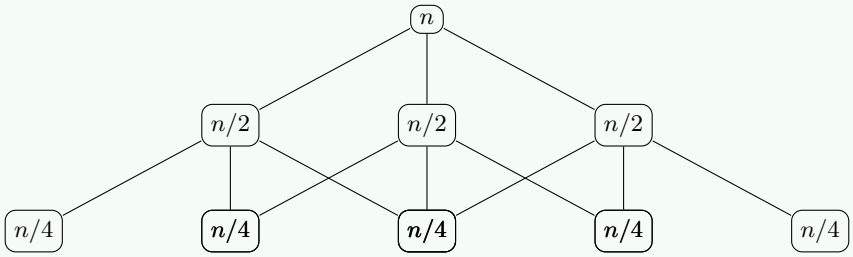


Step 4: Notice the pattern. Every level costs exactly n, since problem size halves but number of nodes doubles, and these effects cancel.
Step 5: Count the levels. Size goes $n, n/2, n/4, \ldots, 1$, giving $\log_2 n$ levels.
Step 6: Total cost.

$$T(n) = n \times \log_2 n = \Theta(n \log n)$$

Example 148: Recursion Tree Example 2: $T(n) = 3T(n/2) + n$

Step 1: Draw the root. Root is size n, cost n.
Step 2: Draw children. Root makes 3 calls of size $n/2$. Level cost $3 \times \frac{n}{2} = \frac{3n}{2}$.
Step 3: Continue expanding. Each of those 3 nodes makes 3 calls of size $n/4$, giving 9 nodes, level cost $9 \times \frac{n}{4} = \frac{9n}{4}$.



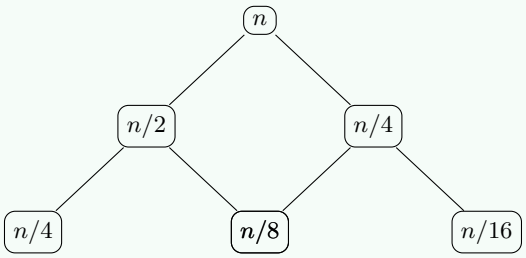
Step 4: Notice the pattern. At level i, there are 3^i nodes, each of size $n/2^i$, so level cost is $3^i \times \frac{n}{2^i} = n\left(\frac{3}{2}\right)^i$. This level cost increases as i increases, since $3/2 > 1$.
Step 5: Count the levels. Number of levels is $\log_2 n$, since size halves every level.
Step 6: Total cost. Since the cost grows with each level, the last level dominates the sum. The last level has $n^{\log_2 3}$ nodes, each doing constant work, giving

$$T(n) = \Theta(n^{\log_2 3})$$

which is approximately $\Theta(n^{1.58})$.

Example 149: Recursion Tree Example 3: $T(n) = T(n/2) + T(n/4) + n$

Step 1: Draw the root. Root is size n, cost n.
Step 2: Draw children. Root makes 2 calls, one of size $n/2$, one of size $n/4$. Level cost is $\frac{n}{2} + \frac{n}{4} = \frac{3n}{4}$.
Step 3: Continue expanding. The $n/2$ node makes calls of size $n/4$ and $n/8$. The $n/4$ node makes calls of size $n/8$ and $n/16$. Level cost is again less than n, around $\frac{9n}{16}$.



Step 4: Notice the pattern. Unlike Example 2, here the level cost keeps shrinking, since $\frac{1}{2} + \frac{1}{4} = \frac{3}{4} < 1$. So the root level cost, n, dominates the total sum.
Step 5: Total cost. Adding up a shrinking sum across all levels still gives a value close to the root cost, just scaled by a constant. So,

$$T(n) = \Theta(n)$$

1.8.3 Extended Master Theorem

Extended Master Theorem

The Extended Master Theorem solves recurrences of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

where $a \geq 1$, $b > 1$, $k \geq 0$ are constants, and p is any real number. Instead of comparing $f(n)$ with $n^{\log_b a}$ directly, we identify the constants a, b, k, p, then compare a with b^k.
Case 1: If $a > b^k$, the work done by the leaves dominates, then

$$T(n) = \Theta(n^{\log_b a})$$

Case 2: If $a = b^k$, the work is balanced across all levels, and the log power p decides the answer:

$$T(n) = \begin{cases} \Theta(n^{\log_b a} \log^{p+1} n) & p > -1 \\ \Theta(n^{\log_b a} \log \log n) & p = -1 \\ \Theta(n^{\log_b a}) & p < -1 \end{cases}$$

Case 3: If $a < b^k$, the work done at the root dominates, and p decides the answer:

$$T(n) = \begin{cases} \Theta(n^k \log^p n) & p \geq 0 \\ \Theta(n^k) & p < 0 \end{cases}$$

Example 150: Extended Master Theorem Example 1: $T(n) = 2T(n/2) + n$

Step 1: Identify constants. $a = 2$, $b = 2$, $f(n) = n = \Theta(n^1 \log^0 n)$, so $k = 1$, $p = 0$.
Step 2: Compare a with b^k.

$$a = 2, \quad b^k = 2^1 = 2 \quad \Rightarrow \quad a = b^k$$

Step 3: Apply Case 2, check p. Since $p = 0 > -1$, use the first sub-case.
Step 4: Compute the answer.

$$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n) = \Theta(n \log n)$$

Example 151: Extended Master Theorem Example 2: $T(n) = 9T(n/3) + n$

Step 1: Identify constants. $a = 9$, $b = 3$, $f(n) = n = \Theta(n^1 \log^0 n)$, so $k = 1$, $p = 0$.
Step 2: Compare a with b^k.

$$a = 9, \quad b^k = 3^1 = 3 \quad \Rightarrow \quad a > b^k$$

Step 3: Apply Case 1. The leaves dominate since $a > b^k$.
Step 4: Compute the answer.

$$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$$

Example 152: Extended Master Theorem Example 3: $T(n) = 3T(n/4) + n \log n$

Step 1: Identify constants. $a = 3$, $b = 4$, $f(n) = n \log n = \Theta(n^1 \log^1 n)$, so $k = 1$, $p = 1$.
Step 2: Compare a with b^k.

$$a = 3, \quad b^k = 4^1 = 4 \quad \Rightarrow \quad a < b^k$$

Step 3: Apply Case 3, check p. Since $p = 1 \geq 0$, use the first sub-case.
Step 4: Compute the answer.

$$T(n) = \Theta(n^k \log^p n) = \Theta(n \log n)$$

When the Extended Master Theorem Cannot Be Used

The Extended Master Theorem still fails to apply when the recurrence does not fit the exact form $a \cdot T(n/b) + \Theta(n^k \log^p n)$. This happens when $a < 1$, since that is not a valid divide-and-conquer split, or when the subproblem sizes are unequal, like $T(n) = T(n/2) + T(n/3) + n$, or when $f(n)$ is not of the form $n^k \log^p n$ at all, for example $f(n) = 2^n$. In such cases, we go back to the recursion tree method or the Akra-Bazzi method.

Example 153: A Case the Basic Master Theorem Missed, Now Solved: $T(n) = 2T(n/2) + n \log n$

Step 1: Identify constants. $a = 2$, $b = 2$, $f(n) = n \log n = \Theta(n^1 \log^1 n)$, so $k = 1$, $p = 1$.
Step 2: Compare a with b^k.

$$a = 2, \quad b^k = 2^1 = 2 \quad \Rightarrow \quad a = b^k$$

Step 3: Apply Case 2, check p. Since $p = 1 > -1$, use the first sub-case.
Step 4: Compute the answer.

$$T(n) = \Theta(n^{\log_2 2} \log^{1+1} n) = \Theta(n \log^2 n)$$

This is exactly the recurrence that the basic Master Theorem could not solve, since $f(n) = n \log n$ sits exactly on $n^{\log_b a} = n$ with an extra log factor. The extended version handles it directly through the p sub-case.

Example 154: Practice 1

Solve $T(n) = 4T(n/2) + n$ using the Extended Master Theorem.
Solution: $a = 4$, $b = 2$, $f(n) = n = \Theta(n^1 \log^0 n)$, so $k = 1$, $p = 0$. $b^k = 2^1 = 2$. Since $a = 4 > b^k = 2$, this is Case 1, giving $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

Example 155: Practice 2

Solve $T(n) = 2T(n/2) + n^2$ using the Extended Master Theorem.
Solution: $a = 2$, $b = 2$, $f(n) = n^2 = \Theta(n^2 \log^0 n)$, so $k = 2$, $p = 0$. $b^k = 2^2 = 4$. Since $a = 2 < b^k = 4$, this is Case 3, and $p = 0 \geq 0$, giving $T(n) = \Theta(n^2 \log^0 n) = \Theta(n^2)$.

Example 156: Practice 3

Solve $T(n) = 8T(n/2) + n^2$ using the Extended Master Theorem.
Solution: $a = 8$, $b = 2$, $f(n) = n^2 = \Theta(n^2 \log^0 n)$, so $k = 2$, $p = 0$. $b^k = 2^2 = 4$. Since $a = 8 > b^k = 4$, this is Case 1, giving $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Example 157: Practice 4

Solve $T(n) = T(n/2) + n$ using the Extended Master Theorem.
Solution: $a = 1$, $b = 2$, $f(n) = n = \Theta(n^1 \log^0 n)$, so $k = 1$, $p = 0$. $b^k = 2^1 = 2$. Since $a = 1 < b^k = 2$, this is Case 3, and $p = 0 \geq 0$, giving $T(n) = \Theta(n^1 \log^0 n) = \Theta(n)$.

Example 158: Practice 5

Solve $T(n) = 4T(n/2) + n \log^{-1} n$ using the Extended Master Theorem.
Solution: $a = 4$, $b = 2$, $f(n) = n \log^{-1} n = \Theta(n^1 \log^{-1} n)$, so $k = 1$, $p = -1$. $b^k = 2^1 = 2$. Since $a = 4 > b^k = 2$, this is Case 1, and Case 1 does not depend on p at all, giving $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

1.9 Analysis of Recursive Algorithms

Analysis of Recursive Algorithms

A recursive algorithm is an algorithm that calls itself on a smaller input, until it reaches a base case, which is the simplest input that needs no further recursion. To analyze a recursive algorithm, we first write its recurrence relation, by checking how many recursive calls happen, how much smaller the input becomes, and how much extra work happens outside the calls. Then, we solve this recurrence using substitution, recursion tree, or master theorem, to get the final order of growth.

General Plan for Recursive Algorithm Analysis

1. Decide the input size parameter, usually n.
2. Identify the basic operation, the operation that repeats with every call.
3. Find how many recursive calls are made, and on what smaller size.
4. Write the recurrence relation, including the base case.
5. Solve the recurrence to get the order of growth.

Factorial Function

Example 159: Understanding Factorial Recursion

Factorial of n, written $n!$, means multiplying all numbers from n down to 1. The recursive idea is, $n! = n \times (n-1)!$, and the base case is $0! = 1$. So, computing factorial of n depends on computing factorial of $n-1$ first.

Algorithm: Factorial(n)

1: **if** $n = 0$ **then**
2: **return** 1
3: **else**
4: **return** $n \times$ Factorial$(n-1)$
5: **end if**

Example 160: Pseudocode for Factorial

Factorial(n)

1. if n == 0
2. return 1
3. else
4. return n * Factorial(n - 1)

Python implementation: factorial.py

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Example 161: Analysis of Factorial Recursion

Step 1: Input size. The parameter n is the input size.
Step 2: Basic operation. The multiplication, $n \times$ Factorial$(n-1)$, is the basic operation.
Step 3: Recursive calls. Each call makes exactly 1 recursive call, on input size $n-1$.
Step 4: Recurrence relation.

$$T(n) = T(n-1) + c, \qquad T(0) = c$$

Here, c represents the constant work done outside the recursive call, like the multiplication and the comparison.
Step 5: Solve. Using the recursion tree method, each level reduces n by 1, and does constant work c. The tree has $n+1$ levels, from n down to 0. Total cost is

$$T(n) = c \times (n+1) = \Theta(n)$$

Fibonacci Function

Example 162: Understanding Fibonacci Recursion

The Fibonacci sequence is $0, 1, 1, 2, 3, 5, 8, \ldots$, where each number is the sum of the two numbers before it. The recursive idea is, $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$, with base cases $\text{Fib}(0) = 0$ and $\text{Fib}(1) = 1$. Unlike factorial, here each call makes two recursive calls, not one.

Algorithm: Fibonacci(n)

1: **if** $n \leq 1$ **then**
2: **return** n
3: **else**
4: **return** Fibonacci$(n-1)$ + Fibonacci$(n-2)$
5: **end if**

Example 163: Pseudocode for Fibonacci

Fibonacci(n)

1. if n <= 1
2. return n
3. else
4. return Fibonacci(n - 1) + Fibonacci(n - 2)

Python implementation: fibonacci.py

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Example 164: Analysis of Fibonacci Recursion

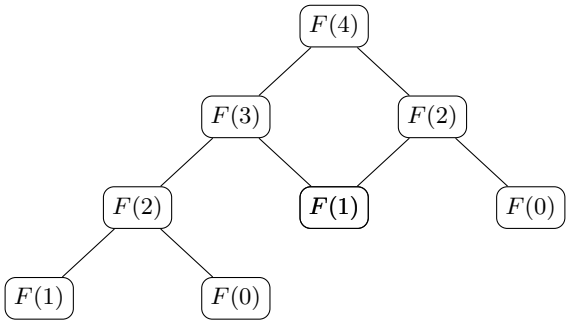
Step 1: Input size. The parameter n is the input size.
Step 2: Basic operation. The addition, combining the two recursive results, is the basic operation.
Step 3: Recursive calls. Each call makes 2 recursive calls, one on $n-1$, one on $n-2$.
Step 4: Recurrence relation.

$$T(n) = T(n-1) + T(n-2) + c, \qquad T(0) = T(1) = c$$

Step 5: Solve. This recurrence does not fit the Master Theorem, since the two subproblems have different sizes. Using the recursion tree method, each call branches into 2 more calls, so the tree looks similar to a binary tree, except the depth on the left branch is longer than the right branch. The total number of nodes in this tree can be shown to grow like the Fibonacci numbers themselves, giving

$$T(n) = \Theta(\phi^n), \qquad \phi \approx 1.618$$

where ϕ is the golden ratio. So, this simple looking function has exponential running time, $\Theta(2^n)$ is also commonly used as a simpler, looser bound for this.



The picture shows the recursion tree for Fib(4). Notice that $F(2)$ and $F(1)$ are computed many times, again and again, which is the reason this recursion is so slow.

Sum of Digits Function

Example 165: Understanding Sum of Digits Recursion

This function adds up all the digits of a number n. The recursive idea is, the sum of digits of n equals the last digit of n, plus the sum of digits of the remaining number, after removing the last digit. The base case is, when n becomes 0, the sum is 0.

Algorithm: SumOfDigits(n)

1: **if** $n = 0$ **then**
2: **return** 0
3: **else**
4: **return** $(n \bmod 10) + \text{SumOfDigits}(\lfloor n/10 \rfloor)$
5: **end if**

Example 166: Pseudocode for Sum of Digits

SumOfDigits(n)

1. if n == 0
2. return 0
3. else
4. return (n mod 10) + SumOfDigits(n / 10)

Python implementation: sum-of-digits.py

```
def sum_of_digits(n):
    if n == 0:
        return 0
    else:
        return (n % 10) + sum_of_digits(n // 10)
```

Example 167: Analysis of Sum of Digits Recursion

Step 1: Input size. Here, the input size is not n itself, but the number of digits in n, since each call removes one digit, not one unit. Let us call the number of digits d.
Step 2: Basic operation. The addition, $(n \bmod 10) + \text{SumOfDigits}(n/10)$, is the basic operation.
Step 3: Recursive calls. Each call makes exactly 1 recursive call, on a number with one less digit.
Step 4: Recurrence relation.

$$T(d) = T(d-1) + c, \qquad T(0) = c$$

Step 5: Solve. Same pattern as factorial, each level does constant work, and there are $d+1$ levels. Total cost is

$$T(d) = \Theta(d)$$

Since $d \approx \log_{10} n$, we can also write this as $\Theta(\log n)$, in terms of the value of n.

Power Function

Example 168: Understanding Power Recursion

This function computes x^n, that is x raised to power n. There are two ways to write this recursively, a slow way and a fast way. We look at both, to see how the recurrence changes the running time.

Example 169: Slow Version: One Reduction at a Time

Here, $x^n = x \times x^{n-1}$, with base case $x^0 = 1$.

Algorithm: PowerSlow(x, n)

```
1: if n = 0 then
2:     return 1
3: else
4:     return x × PowerSlow(x, n − 1)
5: end if
```

Python implementation: power-slow.py

```
def power_slow(x, n):
    if n == 0:
        return 1
    else:
        return x * power_slow(x, n - 1)
```

Recurrence: $T(n) = T(n-1) + c$, giving $T(n) = \Theta(n)$, same pattern as factorial.

Example 170: Fast Version: Halving the Power

Here, we use the idea, $x^n = (x^{n/2})^2$, when n is even, and $x^n = x \times (x^{(n-1)/2})^2$, when n is odd. This way, the power reduces to half each time, instead of reducing by 1.

Algorithm: PowerFast(x, n)

```
1: if n = 0 then
2:     return 1
3: else if n is even then
4:     half ← PowerFast(x, n/2)
5:     return half × half
6: else
7:     half ← PowerFast(x, (n − 1)/2)
8:     return x × half × half
9: end if
```

Example 171: Pseudocode for Power Fast

PowerFast(x, n)

1. if n == 0
2. return 1
3. else if n is even
4. half = PowerFast(x, n / 2)

5. return half * half
6. else
7. half = PowerFast(x, (n - 1) / 2)
8. return x * half * half

Python implementation: power-fast.py

```python
def power_fast(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        half = power_fast(x, n // 2)
        return half * half
    else:
        half = power_fast(x, (n - 1) // 2)
        return x * half * half
```

Example 172: Analysis of Power Fast Recursion

Step 1: Input size. The parameter n is the input size.
Step 2: Basic operation. The multiplication, $half \times half$, is the basic operation.
Step 3: Recursive calls. Each call makes exactly 1 recursive call, on input size roughly $n/2$.
Step 4: Recurrence relation.

$$T(n) = T(n/2) + c, \qquad T(0) = c$$

Step 5: Solve using Master Theorem. Here, $a = 1$, $b = 2$, $f(n) = c = \Theta(1) = \Theta(n^0)$. Compute $n^{\log_b a} = n^{\log_2 1} = n^0$. Since $f(n)$ matches $n^{\log_b a}$ exactly, this is Case 2, giving

$$T(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

This shows the fast version is much better than the slow version, $\Theta(\log n)$ compared to $\Theta(n)$, just by halving the power instead of reducing it by 1 each time.

Example 173: Practice 1: Reverse a Number

A function reverses the digits of a number n recursively, removing one digit and placing it at the end, in each call, until n becomes 0. Identify the recurrence and find the order of growth.
Solution: Each call reduces the number of digits by 1, doing constant work. So, $T(d) = T(d-1) + c$, giving $T(d) = \Theta(d) = \Theta(\log n)$.

Example 174: Practice 2: Recursive Array Sum

A function adds up all elements of an array of size n, by adding the first element to the sum of the rest of the array, recursively. Identify the recurrence and find the order of growth.
Solution: Each call reduces array size by 1, doing constant work to add one element. So, $T(n) = T(n-1) + c$, giving $T(n) = \Theta(n)$.

Example 175: Practice 3: Tower of Hanoi

The Tower of Hanoi problem moves n disks, by first moving $n-1$ disks to a spare rod, then moving the largest disk, then moving the $n-1$ disks again, on top of it. Identify the recurrence and find the order of growth.
Solution: Each call makes 2 recursive calls, each on size $n-1$, plus constant work for moving the largest disk. So, $T(n) = 2T(n-1) + c$. Solving by recursion tree, the tree doubles in nodes each level, while the work per node stays constant, giving $T(n) = \Theta(2^n)$.

Example 176: Practice 4: Recursive Counting Down

A function prints numbers from n down to 1, by printing n, then calling itself on $n-1$. Identify the recurrence and find the order of growth.
Solution: Each call reduces n by 1, doing constant work for the print statement. So, $T(n) = T(n-1) + c$, giving $T(n) = \Theta(n)$.

Example 177: Practice 5: Recursive GCD (Euclid's Method)

A function computes the GCD of two numbers a and b, using $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$, until b becomes 0. This problem is harder to analyze directly using n as size, since the reduction is not a simple halving or minus one, it depends on number theory properties.
Solution: It is known, using a deeper number theory result, that this recursion takes at most $O(\log(\min(a, b)))$ calls, because the smaller number reduces roughly by a multiplicative factor related to the golden ratio, in the worst case, every two calls.

1.10 Useful Formulas

Important Formulas

A. Core Logarithm Identities

(All logarithms below are base 2 unless stated otherwise; the base never affects Θ-notation.)

1. **Product rule:** $\log(mn) = \log m + \log n$
   Example: $\log(8n) = 3 + \log n$.
2. **Quotient rule:** $\log(m/n) = \log m - \log n$
   Example: $\log(n/4) = \log n - 2$.
3. **Power rule:** $\log(n^k) = k \log n$
   Example: $\log(n^3) = 3 \log n$.
4. **Change of base:** $\log_b n = \dfrac{\log_a n}{\log_a b}$
   Example: $\log_2 n = \dfrac{\ln n}{\ln 2}$, so $\log_2 n = \Theta(\ln n)$ – **this is exactly why the base of a log is never written inside Θ-notation**: changing the base only multiplies by a constant.
5. **Inverse identity:** $b^{\log_b n} = n$
   Example: $2^{\log_2 n} = n$.
6. **Base facts:** $\log_b 1 = 0, \quad \log_b b = 1$.
7. **Exponent–log swap (the "hidden" identity):** $a^{\log_b c} = c^{\log_b a}$
   Example: $n^{\log_2 7} = 7^{\log_2 n}$ – this rewriting is exactly how divide-and-conquer exponents like $\Theta(n^{\log_2 7})$ are manipulated.
8. **The "$n^{1/\log n}$ is a constant" trick:** $n^{1/\log_b n} = b$
   Example: $n^{1/\log_2 n} = 2$ for every n – a function that *looks* like it depends on n but is actually $\Theta(1)$.
9. **Sum of logs = log of a factorial:** $\displaystyle\sum_{i=1}^{n} \log i = \log(n!)$
   Example: this identity is the bridge to Stirling's Approximation in Section B below.
10. For very small x, $\log_b(1+x) \approx \frac{x}{\ln(b)}$, where ln is natural logarithm (base e).
11. **Iterated logarithm** $\log^* n$: the number of times log must be applied to n before the result is ≤ 1. It grows slower than $\log\log\cdots\log n$ for *any* fixed number of iterations.
    Example: $\log^*(65536) = 4$, since $\log\log\log\log(65536) \leq 1$; for all practically occurring n, $\log^* n \leq 5$.

B. Factorial and Stirling's Approximation

1. **Stirling's Approximation:**

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+O\left(\frac{1}{n}\right)\right) \quad \Longrightarrow \quad n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$$

Example: for $n = 10$, Stirling gives $10! \approx 3{,}598{,}696$, versus the true value $3{,}628{,}800$ – accurate to within 1%.

2. **Log-factorial form (the single most-used consequence of Stirling):**

$$\ln(n!) = n\ln n - n + O(\log n) \quad \Longrightarrow \quad \log(n!) = \Theta(n\log n)$$

Example: this is precisely why the comparison-based sorting lower bound is $\Omega(n\log n)$, and why $\log(n!)$ and $n\log n$ can always be swapped inside Θ-notation.

3. **Loose factorial bounds (no constants needed):**

$$\left(\frac{n}{e}\right)^n \le n! \le n^n$$

Example: instantly bounds $n!$ between an exponential and n^n without invoking $\sqrt{2\pi n}$.

4. **Factorial's place in the growth hierarchy:** for every constant c,

$$c^n = o(n!) \qquad \text{and} \qquad n! = o(n^n)$$

Example: $2^n = o(n!)$ and $n! = o(n^n)$, so $2^n \prec n! \prec n^n$.

5. **Central binomial coefficient:**

$$\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}} \quad \Longrightarrow \quad \binom{2n}{n} = \Theta\left(\frac{4^n}{\sqrt{n}}\right)$$

Example: shows $\binom{2n}{n} = O(4^n)$ but $\binom{2n}{n} \neq \Theta(4^n)$ – the $\sqrt{n}$ factor in the denominator matters.

6. **Sum of all binomial coefficients:**

$$\sum_{k=0}^{n}\binom{n}{k} = 2^n$$

Example: this is exactly the count of all 2^n subsets of an n-element set – the standard justification for a Brute-Force paradigm's exponential running time.

C. Summation (Series) Formulas

1. **Arithmetic series:**

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

2. **Sum of squares:**

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$$

3. **Sum of cubes:**

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \Theta(n^4)$$

4. **General power sum:**

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1}), \quad \text{for any constant } k \ge 0$$

5. **Finite geometric series**

- For $r > 1$: $S_n = \frac{a(r^n-1)}{r-1}$
- For $r < 1$: $S_n = \frac{a(1-r^n)}{1-r}$
- For S_∞ ($|r| < 1$): $S_\infty = \frac{a}{1-r}$

6. **Harmonic series:**
$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O(1/n) = \Theta(\log n), \qquad \gamma \approx 0.5772$$
Example: directly explains why a loop of the form `for(i=1;i<=n;i++) for(j=1;j<=n;j+=i)` runs $\Theta(n \log n)$ times in total.

7. **Sum of logarithms:**
$$\sum_{i=1}^{n} \log i = \log(n!) = \Theta(n \log n)$$

8. **Arithmetico-geometric series** ($r > 1$):
$$\sum_{i=1}^{n} i\, r^i = \Theta(n\, r^n)$$
Example: $\sum_{i=1}^{n} i \cdot 2^i = \Theta(n \cdot 2^n)$ – again dominated by the last (largest-index) term.

9. **Decaying recursion-tree workload:**
$$\sum_{i=0}^{\log n} \frac{n}{2^i} = \Theta(n)$$
Example: models a divide-and-conquer process where the work per level halves but the number of levels is $\Theta(\log n)$ – the total is still only $\Theta(n)$, **not** $\Theta(n \log n)$.

10. **Growing recursion-tree workload:**
$$\sum_{i=1}^{\log n} 2^i = \Theta(n)$$
Example: work doubling every level for $\Theta(\log n)$ levels also sums to only $\Theta(n)$, since the sum is dominated by the final, largest term.

D. Growth-Rate Hierarchy and Limit Tests

1. **Logarithms lose to every polynomial:**
$$\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} = 0, \quad \text{for any constant } \varepsilon > 0$$
Example: $\log n = o(\sqrt{n})$, and even $\log n = o(n^{0.001})$.

2. **Polynomials lose to every exponential:**
$$\lim_{n\to\infty} \frac{n^k}{c^n} = 0, \quad \text{for any constants } k > 0,\ c > 1$$
Example: $n^{100} = o(1.01^n)$.

3. **The full growth hierarchy** (for constants $0 < \varepsilon < 1$ and $c > 1$):
$$\log^* n \prec \log\log n \prec \log n \prec n^\varepsilon \prec n \prec n \log n \prec n^c \prec c^n \prec n! \prec n^n$$

4. **Ratio (limit) test for comparing f and g:** if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$, then
$$0 < L < \infty \Rightarrow f = \Theta(g), \qquad L = 0 \Rightarrow f = o(g), \qquad L = \infty \Rightarrow f = \omega(g)$$
Example: $\lim_{n\to\infty} \frac{3n^2+5}{n^2} = 3 \Rightarrow 3n^2 + 5 = \Theta(n^2)$.

5. **Euler's number limits** (used in hashing / amortized / probabilistic analyses):
$$\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e, \qquad \lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$$
Example: the probability that n items hashed into n buckets avoid a particular bucket entirely is $(1-1/n)^n \approx 1/e$.

6. **Exponential and log inequalities** (used to bound the expressions above):

$$1 + x \leq e^x \quad \text{for all real } x, \qquad \ln(1+x) \leq x \quad \text{for } x > -1$$

Example: gives the one-line bound $(1 - 1/n)^n \leq e^{-1}$ used just above.

E. Floors/ceilings never matter

For a recurrence $T(n) = a\,T(n/b) + f(n)$ with $a \geq 1$, $b > 1$, let $p = \log_b a$.

- **Floors/ceilings never matter:** $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$ has exactly the same solution, $\Theta(n \log n)$, as $T(n) = 2T(n/2) + O(n)$.

gateai.in

1.11 Practice Problems

Problem 1 ***[MCQ]*** *A problem instance of size n is solved by generating all 2^n possible subsets of an n-element set and independently testing a predicate P on each subset, with no reuse of work across subsets and no early elimination of candidates. This approach belongs to which paradigm?*

(A) Greedy Method

(B) Brute Force

(C) Dynamic Programming

(D) Branch and Bound

Problem 2 ***[MCQ]*** *A problem of size n is solved by reducing it to a single subproblem of size $n-1$ (or $n/2$), recursively solving that one subproblem, and then using its solution with $O(1)$ (or $O(n)$) extra work to construct the solution to the original instance – at no point is more than one independent subproblem generated. Which paradigm does this describe?*

(A) Divide and Conquer

(B) Decrease and Conquer

(C) Transform and Conquer

(D) Branch and Bound

Problem 3 ***[MSQ]*** *Consider the following abstractly-described computational processes on an input of size n. Which of them exhibit the Divide-and-Conquer paradigm, i.e., can be captured by a recurrence of the form $T(n) = aT(n/b) + f(n)$ with $a \geq 2$ independent subproblems that are each solved recursively and then combined?*

(A) The input is split into two halves; each half is processed independently and recursively; the two results are then combined in $O(n)$ time.

(B) The input is first transformed once, via an $O(n \log n)$ preprocessing pass, into a different representation, on which a single $O(n)$ linear scan then suffices – no recursive subdivision occurs at any point.

(C) At each step, a single comparison eliminates one half of the current size-n search space, and only the surviving half is recursively explored further.

(D) The input is partitioned into 7 overlapping groups of size $n/5$; each group is solved recursively and independently; the 7 results are then combined in $\Theta(n)$ time.

Problem 4 ***[MCQ]*** *A problem is solved in two distinct phases. Phase 1 converts the input instance, in $O(n \log n)$ time, into a different representation that carries no additional information but exposes structure absent from the original form. Phase 2 then solves the transformed instance using only a single $O(n)$ pass, exploiting the structure exposed in Phase 1. Which paradigm does this two-phase strategy describe?*

(A) Greedy Method

(B) Transform and Conquer

(C) Backtracking

(D) Branch and Bound

Problem 5 ***[MSQ]*** *Which of the following statements are TRUE about the Greedy Method?*

(A) It always produces the globally optimal solution for every problem.

(B) It makes a locally optimal choice at each step, based only on information available so far.

(C) Once a choice is made, it is never reconsidered or revised.

(D) It is guaranteed to produce a globally optimal solution only for problems that possess the greedy-choice property (together with optimal substructure).

Problem 6 ***[MCQ]*** *A problem's optimal value can be expressed by a recurrence relating $f(i, w)$ to $f(i-1, w)$ and $f(i-1, w-c_i)$ for a sequence of n decision variables and a bounded resource parameter w. Across the different recursive call paths that arise while unrolling this recurrence, the same (i, w) pairs are computed repeatedly (overlapping subproblems), and an optimal solution to the full problem can always be assembled from optimal solutions to these recurring subproblems (optimal substructure). Which paradigm is best suited to solving this efficiently?*

(A) Greedy Method

(B) Dynamic Programming

(C) Backtracking

(D) Brute Force

Problem 7 ***[MCQ]*** *A complete candidate solution is built incrementally, one decision variable at a time. At each step, if the current partial assignment violates a constraint imposed by the variables already fixed, the algorithm discards the most recent choice and tries a different value for that variable, continuing until either a complete valid assignment is found or all options at every level have been exhausted. No bound or estimate on the best achievable objective value is computed or used to prune the search early. Which paradigm does this describe?*

(A) Branch and Bound

(B) Backtracking

(C) Dynamic Programming

(D) Greedy Method

Problem 8 ***[MSQ]*** *Which of the following are TRUE distinctions between Backtracking and Branch and Bound, when both are viewed as traversals of a state-space tree?*

(A) Branch and Bound is typically applied to optimization problems; Backtracking is typically applied to decision/feasibility problems.

(B) Branch and Bound computes a bound at each node to prune subtrees that provably cannot improve upon the current best solution found so far.

(C) Backtracking prunes a branch only because it violates a constraint, never because of a numerical bound comparison.

(D) Both methods can explore only a fraction of the full state-space tree in practice, rather than always enumerating every leaf.

Problem 9 ***[MCQ]*** *Which of the following is NOT a factor that should influence the theoretical efficiency analysis of an algorithm?*

(A) Number of basic operations performed

(B) Order of growth with respect to input size

(C) The brand of compiler used to compile the program

(D) Whether the count depends on input configuration

Problem 10 ***[MCQ]*** *For an algorithm that processes a matrix of size $p \times q$, which is the most natural choice of input size parameter if the algorithm visits every cell exactly once?*

(A) p only

(B) q only

(C) $p + q$

(D) $p \times q$

Problem 11 ***[MSQ]*** *Which of the following are valid reasons why clock time (in seconds) is not used for theoretical algorithm analysis?*

(A) It depends on processor speed.

(B) It depends on compiler optimizations.

(C) It is the only accurate way to measure efficiency.

(D) It depends on the system load at the time of execution.

Problem 12 ***[MCQ]*** *Consider the following code:*

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (A[i] == B[j])
            count++;
```

What is the most appropriate basic operation for this algorithm?

(A) The increment of `count`

(B) The comparison `A[i] == B[j]`

(C) The loop variable initialization

(D) The array indexing of `A`

Problem 13 *Arrange the following functions in increasing order of asymptotic growth rate. Note: Assume the base of log to be* 2*.*

$$n^{1/\log n}, \quad \log\log n, \quad n\log n, \quad (\log n)^{\log n}$$

A. $n^{1/\log n},\ \log\log n,\ n\log n,\ (\log n)^{\log n}$

B. $\log\log n,\ n^{1/\log n},\ n\log n,\ (\log n)^{\log n}$

C. $n^{1/\log n},\ n\log n,\ \log\log n,\ (\log n)^{\log n}$

D. $(\log n)^{\log n},\ n\log n,\ \log\log n,\ n^{1/\log n}$

Problem 14 *Which one of the following expressions is asymptotically EQUAL to* $n^{\log n}$*? Note: Assume the base of log to be* 2*.*

A. $(\log n)^n$

B. $2^{(\log n)^2}$

C. n^2

D. $\log(n^n)$

Problem 15 *Consider the four functions*

$$f_1(n) = 3^n, \quad f_2(n) = 9^{n/2}, \quad f_3(n) = 27^{n/3}, \quad f_4(n) = \left(\sqrt{3}\right)^{2n}$$

Which one of the following statements is TRUE?

A. All four functions are of the same asymptotic order, $\Theta(3^n)$*.*

B. $f_3(n)$ *grows strictly faster than the other three.*

C. $f_1(n) = o(f_2(n))$ *and* $f_2(n) = o(f_3(n))$*.*

D. $f_4(n)$ *grows strictly slower than the other three.*

Problem 16 *Let* $f(n) = n$ *and* $g(n) = n^{1+\cos n}$ *where* n *is a positive integer. Which of the following statements is/are correct?*

I. $f(n) = O(g(n))$

II. $f(n) = \Omega(g(n))$

A. Only I

B. Only II

C. Both I and II

D. Neither I nor II

Problem 17 *Arrange the following functions in increasing order of asymptotic growth rate.*

$$2^n, \quad (n!)^{1/n}, \quad n!, \quad n^n$$

A. $(n!)^{1/n},\ 2^n,\ n!,\ n^n$

B. $(n!)^{1/n},\ n!,\ 2^n,\ n^n$

C. $2^n,\ (n!)^{1/n},\ n!,\ n^n$

D. $(n!)^{1/n},\ 2^n,\ n^n,\ n!$

Problem 18 *Arrange the following functions in increasing order of asymptotic growth rate. Note: Assume the base of log to be* 2*.*

$$(\log n)^2, \quad \sqrt{n}, \quad \log n, \quad \sqrt{\log(n!)}$$

A. $\log n,\ (\log n)^2,\ \sqrt{n},\ \sqrt{\log(n!)}$

B. $(\log n)^2,\ \log n,\ \sqrt{n},\ \sqrt{\log(n!)}$

C. $\log n,\ \sqrt{n},\ (\log n)^2,\ \sqrt{\log(n!)}$

D. $\sqrt{\log(n!)},\ \sqrt{n},\ (\log n)^2,\ \log n$

Problem 19 *Arrange the following functions in increasing order of asymptotic growth rate.*

$$n!, \quad (2n)!, \quad (n!)^2, \quad 4^n \cdot n!$$

A. $n!,\ 4^n \cdot n!,\ (n!)^2,\ (2n)!$

B. $n!,\ (n!)^2,\ 4^n \cdot n!,\ (2n)!$

C. $4^n \cdot n!,\ n!,\ (n!)^2,\ (2n)!$

D. $n!,\ 4^n \cdot n!,\ (2n)!,\ (n!)^2$

Problem 20 *Arrange the following functions in increasing order of asymptotic growth rate.*

$$\log \binom{n}{2}, \quad \log \binom{2n}{n}, \quad \log(n!), \quad n^2$$

A. $\log \binom{n}{2},\ \log \binom{2n}{n},\ \log(n!),\ n^2$

B. $\log \binom{2n}{n},\ \log \binom{n}{2},\ \log(n!),\ n^2$

C. $\log \binom{n}{2},\ \log(n!),\ \log \binom{2n}{n},\ n^2$

D. $n^2,\ \log(n!),\ \log \binom{2n}{n},\ \log \binom{n}{2}$

Problem 21 *Let* $P(n) = n^{\log\log n}$ *and* $Q(n) = (\log n)^{\log n}$*, where all logarithms are base* 2*. Which one of the following statements is TRUE?*

A. $P(n) = o(Q(n))$

B. $P(n) = \omega(Q(n))$

C. $P(n) = \Theta(Q(n))$

D. $P(n)$ *and* $Q(n)$ *are asymptotically incomparable*

Problem 22 *Let* $\log^*(n)$ *denote the iterated logarithm: the number of times* log *must be applied, starting from* n*, before the result is* ≤ 1 *(e.g.,* $\log^*(2) = 1$*,* $\log^*(4) = 2$*,* $\log^*(16) = 3$*,* $\log^*(65536) = 4$*). Which one of the following correctly arranges the given functions in increasing order of asymptotic growth rate?*

$$\log^*(n), \quad \log\log\log n, \quad \log\log n, \quad \log n$$

A. $\log^*(n),\ \log\log\log n,\ \log\log n,\ \log n$

B. $\log\log\log n,\ \log^*(n),\ \log\log n,\ \log n$

C. $\log n,\ \log\log n,\ \log\log\log n,\ \log^*(n)$

D. $\log^*(n),\ \log\log n,\ \log\log\log n,\ \log n$

Problem 23 *Let* $A(n) = \binom{2n}{n}$ *(the central binomial coefficient) and* $B(n) = 4^n$*. It is known that* $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$ *for large* n*. Which one of the following statements is TRUE?*

A. $A(n) = \Theta(B(n))$

B. $A(n) = O(B(n))$ *but* $A(n) \neq \Theta(B(n))$

C. $A(n) = \Omega(B(n))$ *but* $A(n) \neq O(B(n))$

D. $A(n)$ *and* $B(n)$ *are asymptotically incomparable*

Problem 24 *Consider a comparison-based algorithm* `PARTITIONMERGE`(A, n) *designed to rearrange an array* A *of* n *elements. It divides* A *into* 5 *equal contiguous blocks, and recursively invokes itself on* 7 *overlapping groups of size* $n/5$ *formed from adjacent blocks; it then spends* $\Theta(n)$ *additional time combining the* 7 *results using pairwise comparisons. If this is to be expressed as* $O(n^c)$ *for the smallest possible* ***integer*** *value of* c*, then the value of* c *is* _____.

Problem 25 *Which of the following statements is/are TRUE for all positive functions* f, g, h*?*

I. $f = \Omega(g)$ *and* $g = \Omega(h) \implies f = \Omega(h)$

II. $f = \omega(g) \implies f = \Omega(g)$*, but the converse need not hold*

III. $f = \Theta(g)$ *if and only if* $f = O(g)$ *and* $f = \Omega(g)$

IV. $f = o(g) \implies f = O(g)$ *and* $f \neq \Omega(g)$

A. *I and III only*

B. *I, II and III only*

C. *I, II, III and IV*

D. *II and IV only*

Problem 26 *Let* $f(n) = O(g(n))$ *and* $h(n) = \Omega(g(n))$*, where* f, g, h *are asymptotically positive functions. Which one of the following statements is TRUE?*

A. $f(n) = O(h(n))$ *must hold.*

B. $h(n) = O(f(n))$ *must hold.*

C. *Neither* $f(n) = O(h(n))$ *nor* $h(n) = O(f(n))$ *is guaranteed to hold in general.*

D. $f(n) = \Theta(h(n))$ *must hold.*

Problem 27 *Let* $f(n) = O(g(n))$ *and* $h(n) = O(g(n))$*, where* f, g, h *are asymptotically positive functions. Which one of the following statements is TRUE?*

A. $f(n) = O(h(n))$ *must hold.*

B. $h(n) = O(f(n))$ *must hold.*

C. *Neither* $f(n) = O(h(n))$ *nor* $h(n) = O(f(n))$ *is guaranteed to hold in general.*

D. $f(n) = \Theta(h(n))$ *must hold.*

Problem 28 *Which one of the following statements is TRUE, for positive functions* $f(n)$ *and* $g(n)$*?*

A. *For all* f, g*, exactly one of* $f(n) = O(g(n))$ *or* $f(n) = \Omega(g(n))$ *must hold.*

B. *There exist positive functions* f, g *such that neither* $f(n) = O(g(n))$ *nor* $f(n) = \Omega(g(n))$ *holds.*

C. $f(n) \neq O(g(n))$ *always implies* $f(n) = \Omega(g(n))$*.*

D. $f(n) = O(g(n))$ *together with* $f(n) \neq \Omega(g(n))$ *always implies* $f(n) = o(g(n))$*.*

Problem 29 *Let* $f(n) = O(g(n))$ *and* $g(n) = o(h(n))$*. Which one of the following MUST be TRUE?*

A. $f(n) = o(h(n))$

B. $h(n) = O(f(n))$

C. $f(n) = \Theta(h(n))$

D. $g(n) = O(f(n))$

Problem 30 *Let $B(n)$, $A(n)$, and $W(n)$ denote respectively the best-case, average-case, and worst-case running time of an algorithm on an input of size n. Which of the following is/are ALWAYS TRUE?*

I. $B(n) = O(A(n))$

II. $A(n) = O(W(n))$

III. $A(n) = \Theta(W(n))$

IV. $B(n) = \Theta(A(n))$

A. I and II only

B. II only

C. I, II, III and IV

D. III and IV only

Problem 31 *Given $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, where all functions are asymptotically positive, which of the following is/are TRUE?*

I. $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

II. $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

III. $f_1(n) - f_2(n) = O(g_1(n) - g_2(n))$

IV. $f_1(n)/f_2(n) = O(g_1(n)/g_2(n))$

A. I and II only

B. I, II and III

C. I, II, III and IV

D. III and IV only

Problem 32 *Consider the recurrence relation $T(n) = 3T(n/3) + \dfrac{n}{\log n}$, with $T(1) = 1$. Which one of the following gives the Asymptotic Notation of $T(n)$?*

A. $\Theta(n)$

B. $\Theta(n \log\log n)$

C. $\Theta(n \log n)$

D. $\Theta(n \log^2 n)$

Problem 33 *Consider the recurrence relation $T(n) = 2T(\sqrt{n}) + \log n$, with $T(2) = 1$. Which one of the following gives the Asymptotic Notation of $T(n)$?*

A. $\Theta(\log n)$

B. $\Theta(\log n \cdot \log\log n)$

C. $\Theta(\log^2 n)$

D. $\Theta(\sqrt{n} \log n)$

Problem 34 *Consider the following pseudocode for a recursive function $A(n)$, where every statement other than the recursive calls takes $O(1)$ time, and $P1$, $P2$, $P3$ are conditions whose outcomes are data-dependent.*

```
A(n):
    if n <= 1: return
    A(n/2)                          // Call 1 (always made)
    if P1:
        A(n/2); return              // Call 2
    else:
        A(n/2)                      // Call 2
        if P2:
            A(n/2); return          // Call 3
```

```
    else:
        A(n/2)                  // Call 3
        if P3:
            return
        else:
            A(n/2); return      // Call 4
```

If the worst-case Asymptotic Notation of $A(n)$ is $\Theta(n^\alpha)$, the value of α (accurate to two decimal places) is _____.

Problem 35 *Consider a recursive comparison-based algorithm $A(n)$ that, on an input of size n, makes two recursive calls – one on a sub-problem of size $n-1$ and one on a sub-problem of size $n-2$ – and combines their results using $O(1)$ additional comparisons. This gives the recurrence $T(n) = T(n-1) + T(n-2) + O(1)$, with $T(0) = T(1) = O(1)$. Which one of the following is the TIGHTEST correct Asymptotic Notation of $T(n)$?*

A. $\Theta(2^n)$

B. $\Theta(n^2)$

C. $\Theta(\varphi^n)$, *where* $\varphi = \dfrac{1+\sqrt{5}}{2}$

D. $\Theta(n!)$

Problem 36 *Consider the recurrence relation $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$, with $T(1) = 1$. Which one of the following gives the Asymptotic Notation of $T(n)$?*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. The floor and ceiling terms make this impossible to determine without knowing n exactly

Problem 37 *Consider the recurrence relation $T(n) = T(n/3) + T(2n/3) + n$, with $T(1) = 1$. Which one of the following gives the Asymptotic Notation of $T(n)$?*

A. $\Theta(n)$, *since one branch shrinks much faster than the other*

B. $\Theta(n \log n)$, *since the total work done at every level of the recursion tree is* $\Theta(n)$ *and the tree has* $\Theta(\log n)$ *levels*

C. $\Theta(n^{\log_3 2})$

D. $\Theta(n^2)$

Problem 38 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j = j + i)
        count++;
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(\log(n))$

Problem 39 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j = j + 1)
        count++;
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(\log n)$

C. $\Theta(n^2)$

D. $\Theta(\log(n!))$

Problem 40 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j *= 2)
        count++;
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(\log n)$

C. $\Theta(n^2)$

D. $\Theta(\log(n!))$

Problem 41 *Consider the following C-like code fragment.*

```
count = 0;
i = 1;
while (i <= n) {
    j = 1;
    while (j <= i) {
        count++;
        j = j + i;
    }
    i++;
}
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(\sqrt{n})$

Problem 42 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 1; i * i <= n; i++)
    for (j = 1; j <= n; j++)
        count++;
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(\sqrt{n})$

B. $\Theta(n)$

C. $\Theta(n\sqrt{n})$

D. $\Theta(n^2)$

Problem 43 *Consider the following C-like code fragment.*

```
count = 0;
sum = 0;
for (i = 1; i <= n; i++) {
    sum = sum + i;
    for (j = 1; j <= sum; j *= 2)
        count++;
}
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n \log^2 n)$

D. $\Theta(n^2)$

Problem 44 *Consider the following C-like code fragment.*

```
count = 0;
i = 1; j = n;
while (i < j) {
    i = i * 2;
    j = j - 1;
    count++;
}
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(\log n)$

B. $\Theta(\sqrt{n})$

C. $\Theta(n)$

D. $\Theta(n \log n)$

Problem 45 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        if ((i + j) % 2 == 0)
            count++;
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. *It cannot be expressed using* Θ*, since the* `if` *condition is satisfied unpredictably*

Problem 46 *Consider the following C-like code fragment.*

```
count = 0;
for (i = 2; i <= n; i++) {
    j = 2;
    while (j <= i) {
        j = j * j;
        count++;
    }
}
```

The Asymptotic Notation of the final value of `count` *is*

A. $\Theta(n)$

B. $\Theta(n \log \log n)$

C. $\Theta(n \log n)$

D. $\Theta(\log \log n)$

Problem 47 *An array* A *of* $n+1$ *integers is known to contain values only from the range* $[1, n]$*, and it is guaranteed that at least one value repeats. Consider the following comparison-based algorithm that locates a repeated value.*

```
LOCATE(A, lo, hi):
    if lo == hi:
        return lo
    mid = (lo + hi) / 2
    count = 0
    for each element a in A:
        if lo <= a <= mid:
            count = count + 1
    if count > (mid - lo + 1):
        return LOCATE(A, lo, mid)
    else:
        return LOCATE(A, mid + 1, hi)
```

Note that every call to `LOCATE` *scans the entire array A (of size $n+1$), regardless of how small the range $[lo, hi]$ has become. The Asymptotic Notation of the worst-case running time of* `LOCATE` *is*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(\log n)$

Problem 48 *Consider the sum $\displaystyle\sum_{i=2}^{n} \frac{1}{i \ln i}$ (natural logarithm). Using the substitution $u = \ln x$ in the integral test, the Asymptotic Notation of this sum is*

A. $\Theta(1)$

B. $\Theta(\log \log n)$

C. $\Theta(\log n)$

D. $\Theta(n)$

Problem 49 *The Asymptotic Notation of $\displaystyle\sum_{i=1}^{n} \frac{i}{i^2+1}$ is*

A. $\Theta(1)$

B. $\Theta(\log n)$

C. $\Theta(\sqrt{n})$

D. $\Theta(n)$

Problem 50 *Consider an algorithm* `PHASESCAN`(A, n) *that processes an array A of n elements in phases numbered $k = 1, 2, \ldots, \log n$. In phase k, it performs pairwise comparisons across a prefix of A of length 2^k, costing $\Theta(2^k)$ time, and then proceeds to phase $k+1$; the algorithm halts once the prefix length reaches n. The total running time is therefore captured by the sum $\displaystyle\sum_{k=1}^{\log n} 2^k$. The Asymptotic Notation of the total running time of* `PHASESCAN` *is*

A. $\Theta(\log n)$

B. $\Theta(n)$

C. $\Theta(n \log n)$

D. $\Theta(n^2)$

Problem 51 *Consider the sum $\displaystyle\sum_{k=0}^{\log n} \frac{n}{2^k}$ (assume base of log is 2), which commonly arises while computing the total work done across all levels of a divide-and-conquer recursion tree whose work per level halves. The Asymptotic Notation of this sum is*

A. $\Theta(\log n)$

B. $\Theta(n)$

C. $\Theta(n \log n)$

D. $\Theta(n^2)$

Problem 52 *The Asymptotic Notation of* $\sum_{i=1}^{n} i \log i$ *is*

A. $\Theta(n \log n)$

B. $\Theta(n^2)$

C. $\Theta(n^2 \log n)$

D. $\Theta(n^3)$

Problem 53 ***[NAT]*** *If* $f(n) = \Theta(n^2)$ *using constants* $c_1 = 2$, $c_2 = 5$, *and* $n_0 = 3$, *then at* $n = 3$, *the minimum possible value of* $f(n)$ *consistent with these constants is* _____.

Problem 54 ***[MCQ]*** *Which property correctly describes Transpose Symmetry?*

(A) $f(n) = \Theta(g(n))$ *iff* $g(n) = \Theta(f(n))$

(B) $f(n) = O(g(n))$ *iff* $g(n) = \Omega(f(n))$

(C) $f(n) = O(f(n))$ *for all* $f(n)$

(D) $f(n) = \Theta(g(n))$ *and* $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$

Problem 55 ***[MCQ]*** *If* $f(n) = o(g(n))$, *then* $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ *equals:*

(A) ∞

(B) A positive finite constant

(C) 0

(D) Cannot be determined

Problem 56 ***[MCQ]*** *Given* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 5$, *which of the following is the most precise correct classification?*

(A) $f(n) = o(g(n))$

(B) $f(n) = \omega(g(n))$

(C) $f(n) = \Theta(g(n))$

(D) $f(n) = O(g(n))$ *only, not* $\Omega(g(n))$

Problem 57 ***[MCQ]*** *Consider linear search on an unsorted array of* n *elements where the target is guaranteed to be present, with equal probability at any position. The average case number of comparisons is closest to:*

(A) n

(B) $\frac{n+1}{2}$

(C) $\log_2 n$

(D) n^2

Problem 58 ***[MSQ]*** *Which of the following statements correctly describe the relationship between worst case analysis and asymptotic notation?*

(A) Worst case can be expressed using O, Ω, *or* Θ, *depending on what has been proved.*

(B) Worst case must always be expressed using Big-Oh only.

(C) If the exact worst case formula is known, Θ *is the most precise notation to use.*

(D) Best case can never be written using Ω *notation.*

Problem 59 ***[MCQ]*** *A student claims: "Best case running time is always represented using Ω notation." This claim is:*

(A) True, by definition of best case

(B) False, since best case can be expressed using O, Ω, or Θ

(C) True, but only for sorting algorithms

(D) False, best case is always Θ only

Problem 60 ***[MSQ]*** *Which of the following code segments have time complexity $\Theta(n^2)$?*

(A)
```
for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            x++;
```

(B)
```
for (i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
            x++;
```

(C)
```
for (i = 1; i <= n; i = i * 2)
        for (j = 1; j <= n; j++)
            x++;
```

(D)
```
for (i = 1; i <= n*n; i++)
        x++;
```

Problem 61 ***[MCQ]*** *What is the worst case time complexity of the following code?*

```
if (n % 2 == 0) {
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            x++;
} else {
    for (i = 0; i < n; i++)
        x++;
}
```

(A) $\Theta(n)$

(B) $\Theta(n^2)$

(C) $\Theta(n^3)$

(D) $\Theta(\log n)$

Problem 62 ***[MCQ]*** *What is the time complexity of the code below, in terms of n?*

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++)
        x++;
```

where m is a fixed constant, independent of n.

(A) $\Theta(n)$

(B) $\Theta(m)$

(C) $\Theta(nm)$

(D) $\Theta(n^2)$

1.12 GATE PYQs

GATEPYQ 1 (GATE CSE 2026) *Consider an array A of integers of size n. The indices of A run from 1 to n. An algorithm is to be designed to check whether A satisfies the condition given below.*

$$\forall i, j \in \{1, \ldots, n-1\} \text{ such that } i > j, (A[i+1] - A[i]) > (A[j+1] - A[j])$$

Which one of the following gives the worst case time complexity of the fastest algorithm that can be designed for the problem?

A. $\Theta(n)$

B. $\Theta(\log(n))$

C. $\Theta(n \log(n))$

D. $\Theta(n^2)$

GATEPYQ 2 (GATE CSE 2026) *Consider the following functions, where n is a positive integer.*

$$n^{1/3}, \quad \log(n), \quad \log(n!), \quad 2^{\log(n)}$$

Which one of the following options lists the functions in increasing order of asymptotic growth rate? Note: Assume the base of log to be 2.

A. $\log(n), n^{1/3}, 2^{\log(n)}, \log(n!)$

B. $n^{1/3}, \log(n), \log(n!), 2^{\log(n)}$

C. $\log(n), n^{1/3}, \log(n!), 2^{\log(n)}$

D. $2^{\log(n)}, n^{1/3}, \log(n), \log(n!)$

GATEPYQ 3 (GATE CSE 2024) *Given an integer array of size N, we want to check if the array is sorted (in either ascending or descending order). An algorithm solves this problem by making a single pass through the array and comparing each element of the array only with its adjacent elements. The worst-case time complexity of this algorithm is*

A. *both $O(N)$ and $\Omega(N)$*

B. *$O(N)$ but not $\Omega(N)$*

C. *$\Omega(N)$ but not $O(N)$*

D. *neither $O(N)$ nor $\Omega(N)$*

GATEPYQ 4 (GATE CSE 2023) *Consider functions Function 1 and Function 2 expressed in pseudocode as follows:*

Function_1

```
while n > 1 do
  for i = 1 to n do
    x = x + 1;
  end for
  n = ⌊n/2⌋;
end while
```

Function_2

```
for i = 1 to 100 * n do
  x = x + 1;
end for
```

Let $f_1(n)$ and $f_2(n)$ denote the number of times the statement "$x = x + 1$" is executed in Function 1 and Function 2, respectively. Which of the following statements is/are TRUE?

A. $f_1(n) \in \Theta(f_2(n))$

B. $f_1(n) \in o(f_2(n))$

C. $f_1(n) \in \omega(f_2(n))$

D. $f_1(n) \in O(n)$

GATEPYQ 5 (GATE CSE 2023) *Let f and g be functions of natural numbers given by $f(n) = n$ and $g(n) = n^2$. Which of the following statements is/are TRUE?*

A. $f \in O(g)$

B. $f \in \Omega(g)$

C. $f \in o(g)$

D. $f \in \Theta(g)$

GATEPYQ 6 (GATE CSE 2022) *Which one of the following statements is TRUE for all positive functions $f(n)$?*

A. $f(n^2) = \Theta(f(n)^2)$, *where $f(n)$ is a polynomial*

B. $f(n^2) = o(f(n)^2)$

C. $f(n^2) = O(f(n)^2)$, *where $f(n)$ is an exponential function*

D. $f(n^2) = \Omega(f(n)^2)$

GATEPYQ 7 (GATE CSE 2021) *Consider the following three functions.*

$$f_1 = 10^n \quad f_2 = n^{\log n} \quad f_3 = n^{\sqrt{n}}$$

Which one of the following options arranges the functions in the increasing order of asymptotic growth rate?

A. f_3, f_2, f_1

B. f_2, f_1, f_3

C. f_1, f_2, f_3

D. f_2, f_3, f_1

GATEPYQ 8 (GATE CSE 2019) *There are n unsorted arrays: $A_1, A_2, \ldots, A_n$. Assume that n is odd. Each of $A_1, A_2, \ldots, A_n$ contains n distinct elements. There are no common elements between any two arrays. The worst-case Asymptotic Notation of computing the median of the medians of $A_1, A_2, \ldots, A_n$ is .*

A. $O(n)$

B. $O(n \log n)$

C. $O(n^2)$

D. $\Omega(n^2 \log n)$

GATEPYQ 9 (GATE CSE 2017) *Consider the following C function*

```
int fun (int n) {
    int i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j += i) {
            printf("%d %d", i, j);
        }
    }
}
```

Asymptotic Notation of fun in terms of Θ notation is

A. $\Theta(n\sqrt{n})$

B. $\Theta(n^2)$

C. $\Theta(n \log n)$

D. $\Theta(n^2 \log n)$

GATEPYQ 10 (GATE CSE 2017) *Consider the following functions from positive integers to real numbers:*

$$10, \quad \sqrt{n}, \quad n, \quad \log_2 n, \quad \frac{100}{n}$$

The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is:

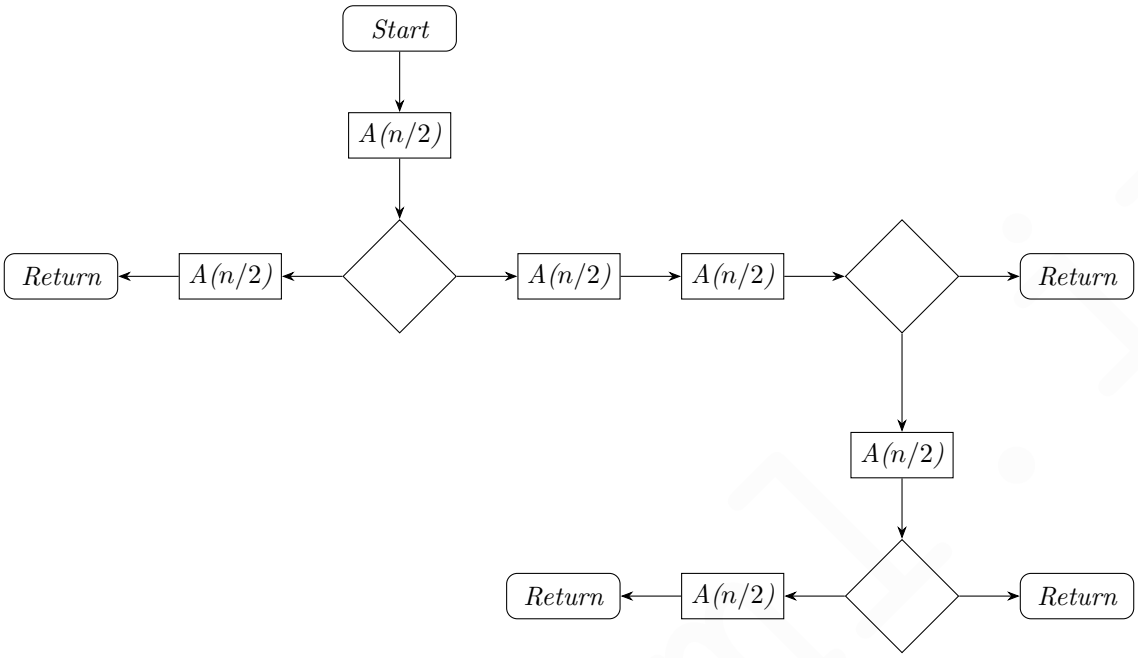
A. $\log_2 n, \quad \frac{100}{n}, \quad 10, \quad \sqrt{n}, \quad n$

B. $\frac{100}{n}, \quad 10, \quad \log_2 n, \quad \sqrt{n}, \quad n$

C. $10, \quad \frac{100}{n}, \quad \sqrt{n}, \quad \log_2 n, \quad n$

D. $\frac{100}{n}, \quad \log_2 n, \quad 10, \quad \sqrt{n}, \quad n$

GATEPYQ 11 (GATE CSE 2016) *The given diagram shows the flowchart for a recursive function $A(n)$. Assume that all statements, except for the recursive calls, have $O(1)$ Asymptotic Notation. If the worst case Asymptotic Notation of this function is $O(n^{\alpha})$, then the least possible value (accurate upto two decimal positions) of α is _____.*



GATEPYQ 12 (GATE CSE 2015 SET-3) *Let $f(n) = n$ and $g(n) = n^{1+\sin n}$ where n is a positive integer. Which of the following statements is/are correct?*

I. $f(n) = O(g(n))$

II. $f(n) = \Omega(g(n))$

A. Only I

B. Only II

C. Both I and II

D. Neither I nor II

GATEPYQ 13 (GATE CSE 2015 SET-3) *Consider the equality $\sum_{i=0}^{n} i^3 = X$ and the following choices for X:*

I. $\Theta(n^4)$

II. $\Theta(n^5)$

III. $O(n^5)$

IV. $\Omega(n^3)$

The equality above remains correct if X is replaced by

A. Only I

B. Only II

C. I or III or IV but not II

D. II or III or IV but not I

GATEPYQ 14 (GATE CSE 2015) *Consider the following C function.*

```
int fun1(int n){
    int i, j, k, p, q = 0;
    for (i = 1; i < n; ++i) {
        p = 0;
        for (j = n; j > 1; j = j / 2)
```

```
            ++p;
        for (k = 1; k < p; k = k * 2)
            ++q;
    }
    return q;
}
```

Which one of the following most closely approximates the return value of the function ***fun1****?*

A. n^3

B. $n(\log n)^2$

C. $n \log n$

D. $n \log(\log n)$

GATEPYQ 15 (GATE CSE 2013) *Consider the following function:*

```
int unknown(int n){
   int i, j, k = 0;
   for (i = n / 2; i <= n; i++)
        for (j = 2; j <= n; j = j * 2)
            k = k + n / 2;
    return (k);
}
```

The return value of the function is

A. $\Theta(n^2)$

B. $\Theta(n^2 \log n)$

C. $\Theta(n^3)$

D. $\Theta(n^3 \log n)$

GATEPYQ 16 (GATE CSE 2012) *Let $W(n)$ and $A(n)$ denote respectively, the worst case and average case running time of an algorithm executed on an input of size n. Which of the following is ALWAYS TRUE?*

A. $A(n) = \Omega(W(n))$

B. $A(n) = \Theta(W(n))$

C. $A(n) = O(W(n))$

D. $A(n) = o(W(n))$

GATEPYQ 17 (GATE CSE 2011) *Which of the given options provides the increasing order of asymptotic complexity of functions f_1, f_2, f_3 and f_4?*

$$f_1(n) = 2^n, \quad f_2(n) = n^{3/2}, \quad f_3(n) = n \log_2 n, \quad f_4(n) = n^{\log_2 n}$$

A. f_3, f_2, f_4, f_1

B. f_3, f_2, f_1, f_4

C. f_2, f_3, f_1, f_4

D. f_2, f_3, f_4, f_1

GATEPYQ 18 (GATE CSE 2010) *Two alternative packages A and B are available for processing a database having 10^k records. Package A requires $0.0001n^2$ time units and package B requires $10n \log_{10} n$ time units to process n records. What is the smallest value of k for which package B will be preferred over A?*

A. 12

B. 10

C. 6

D. 5

GATEPYQ 19 (GATE IT 2008) *Arrange the following functions in increasing asymptotic order:*

a. $n^{1/3}$

b. e^n

c. $n^{7/4}$

d. $n\log^9 n$

e. 1.0000001^n

A. a, d, c, e, b

B. d, a, c, e, b

C. a, c, d, e, b

D. a, c, d, b, e

GATEPYQ 20 (GATE CSE 2008) *Consider the following C functions:*

```
int f1(int n)
{
  if(n == 0 || n == 1)
    return n;
  else
    return (2*f1(n-1) + 3*f1(n-2));
}

int f2(int n)
{
  int i;
  int X[N], Y[N], Z[N] ;
  X[0] = Y[0] = Z[0] = 0;
  X[1] = 1; Y[1] = 2; Z[1] = 3;
  for(i = 2; i <= n; i++)
  {
    X[i] = Y[i-1] + Z[i-2];
    Y[i] = 2*X[i];
    Z[i] = 3*X[i];
  }
  return X[n] ;
}
```

The running time of `f1(n)` *and* `f2(n)` *are*

A. $\Theta(n)$ *and* $\Theta(n)$

B. $\Theta(2^n)$ *and* $\Theta(n)$

C. $\Theta(n)$ *and* $\Theta(2^n)$

D. $\Theta(2^n)$ *and* $\Theta(2^n)$

GATEPYQ 21 (GATE CSE 2008) *Consider the following functions:*

$$f(n) = 2^n, \quad g(n) = n!, \quad h(n) = n^n$$

Which of the following statements about the asymptotic behaviour of $f(n)$*,* $g(n)$*, and* $h(n)$ *is true?*

A. $f(n) = O(g(n)); \quad g(n) = O(h(n))$

B. $f(n) = \Omega(g(n)); \quad g(n) = O(h(n))$

C. $g(n) = O(f(n)); \quad h(n) = O(f(n))$

D. $h(n) = O(f(n)); \quad g(n) = \Omega(f(n))$

GATEPYQ 22 (GATE CSE 2007) *Consider the following C code segment:*

```
int IsPrime(n)
{
  int i, n;
  for(i = 2; i <= sqrt(n); i++)
     if(n % i == 0)
      {  printf("Not Prime\n"); return 0; }
  return 1;
}
```

Let $T(n)$ denote the number of times the for loop is executed by the program on input n. Which of the following is TRUE?

A. $T(n) = O(\sqrt{n})$ *and* $T(n) = \Omega(\sqrt{n})$

B. $T(n) = O(n)$ *and* $T(n) = \Omega(\sqrt{n})$

C. $T(n) = O(\sqrt{n})$ *and* $T(n) = \Omega(1)$

D. None of the above

GATEPYQ 23 (GATE CSE 2007) *What is the Asymptotic Notation of the following recursive function:*

```
int DoSomething (int n)
{
  if (n <= 2)
    return 1;
  else
    return (DoSomething (floor(sqrt(n))) + n);
}
```

A. $\Theta(n^2)$

B. $\Theta(n \log_2 n)$

C. $\Theta(\log_2 n)$

D. $\Theta(\log_2 \log_2 n)$

GATEPYQ 24 (GATE CSE 2006) *Consider the following C-program fragment in which i, j, and n are integer variables.*

```
for (i = n, j = 0; i > 0; i /= 2, j += i);
```

Let $Val(j)$ denote the value stored in the variable j after termination of the for loop. Which one of the following is true?

A. $Val(j) = \Theta(\log n)$

B. $Val(j) = \Theta(\sqrt{n})$

C. $Val(j) = \Theta(n)$

D. $Val(j) = \Theta(n \log n)$

GATEPYQ 25 (GATE IT 2004) *Let $f(n)$, $g(n)$ and $h(n)$ be functions defined for positive integers such that $f(n) = O(g(n))$, $g(n) \neq O(f(n))$, $g(n) = O(h(n))$, and $h(n) = O(g(n))$. Which one of the following statements is FALSE?*

A. $f(n) + g(n) = O(h(n) + h(n))$

B. $f(n) = O(h(n))$

C. $h(n) \neq O(f(n))$

D. $f(n)h(n) \neq O(g(n)h(n))$

GATEPYQ 26 (GATE CSE 2004) *The Asymptotic Notation of the following C function is (assume $n > 0$)*

```
int recursive (int n) {
   if (n == 1)
      return (1);
   else
      return (recursive (n - 1) + recursive (n - 1));
}
```

A. $O(n^2)$

B. $O(n \log n)$

C. $O(n)$

D. $O(2^n)$

GATEPYQ 27 (GATE CSE 2004) *Let $A[1, \ldots, n]$ be an array storing a bit (1 or 0) at each location, and $f(m)$ is a function whose Asymptotic Notation is $\Theta(m)$. Consider the following program fragment written in a C like language:*

```
counter = 0;
for (i = 1; i <= n; i++)
{
      if (A[i] == 1)
         counter++;
      else {
         f(counter);
         counter = 0;
      }
}
```

The complexity of this program fragment is

A. $\Omega(n^2)$

B. $\Omega(n \log n)$ *and* $O(n^2)$

C. $\Theta(n)$

D. $o(n)$

GATEPYQ 28 (GATE CSE 2003) *Consider the following three claims:*

1. $(n + k)^m = \Theta(n^m)$, *where k and m are constants*
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct?

A. *1 and 2*

B. *1 and 3*

C. *2 and 3*

D. *1, 2 and 3*

GATEPYQ 29 (GATE CSE 2000) *Consider the following functions:*

$$f(n) = 3^{n^{\sqrt{n}}}, \quad g(n) = 2^{\sqrt{n}\log_2 n}, \quad h(n) = n!$$

Which of the following is true?

A. $h(n)$ *is* $O(f(n))$

B. $h(n)$ *is* $O(g(n))$

C. $g(n)$ *is not* $O(f(n))$

D. $f(n)$ *is* $O(g(n))$

GATEPYQ 30 (GATE CSE 2000) *Let S be a sorted array of n integers. Let $T(n)$ denote the time taken for the most efficient algorithm to determine if there are two elements with sum less than 1000 in S. Which of the following statements is true?*

A. $T(n)$ *is* $O(1)$

B. $n \leq T(n) \leq n \log_2 n$

C. $n \log_2 n \leq T(n) < n^2$

D. $T(n) = \Theta(n^2)$

GATEPYQ 31 (GATE CSE 1999) *If* $T_1 = O(1)$*, give the correct matching for the following pairs:*

(M) $T_n = T_{n-1} + n$	*(U)* $T_n = O(n)$
(N) $T_n = T_{n/2} + n$	*(V)* $T_n = O(n \log n)$
(O) $T_n = T_{n/2} + n \log n$	*(W)* $T_n = O(n^2)$
(P) $T_n = T_{n-1} + \log n$	*(X)* $T_n = O(\log^2 n)$

A. M-W, N-V, O-U, P-X

B. M-W, N-U, O-X, P-V

C. M-V, N-W, O-X, P-U

D. M-W, N-U, O-V, P-V

GATEPYQ 32 (GATE CSE 1997) *Let* $T(n)$ *be the function defined by:*

$$T(1) = 1, \quad T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \sqrt{n} \quad \text{for } n \geq 2$$

Which of the following statements is true?

A. $T(n) = O(\sqrt{n})$

B. $T(n) = O(n)$

C. $T(n) = O(\log n)$

D. None of the above

GATEPYQ 33 (GATE CSE 1996) *Which of the following is false?*

A. $100n \log n = O\left(\frac{n \log n}{100}\right)$

B. $\sqrt{\log n} = O(\log \log n)$

C. If $0 < x < y$ *then* $n^x = O(n^y)$

D. $2^n \neq O(n^k)$

GATEPYQ 34 (GATE CSE 1994) *Consider the following two functions:*

$$g_1(n) = \begin{cases} n^3 & \text{for } 0 \leq n \leq 10,000 \\ n^2 & \text{for } n \geq 10,000 \end{cases}$$

$$g_2(n) = \begin{cases} n & \text{for } 0 \leq n \leq 100 \\ n^3 & \text{for } n > 100 \end{cases}$$

Which of the following is true?

A. $g_1(n)$ *is* $O(g_2(n))$

B. $g_1(n)$ *is* $O(n^3)$

C. $g_2(n)$ *is* $O(g_1(n))$

D. $g_2(n)$ *is* $O(n)$

GATEPYQ 35 (GATE CSE 1993) $\sum_{1 \leq k \leq n} O(n)$*, where* $O(n)$ *stands for order* n*, is:*

A. $O(n)$

B. $O(n^2)$

C. $O(n^3)$

D. $O(3n^2)$

E. $O(1.5n^2)$

GATEPYQ 36 (GATE CSE 2001) *Let $f(n) = n^2 \log n$ and $g(n) = n(\log n)^{10}$ be two positive functions of n. Which of the following statements is correct?*

A. $f(n) = O(g(n))$ *and* $g(n) \neq O(f(n))$

B. $g(n) = O(f(n))$ *and* $f(n) \neq O(g(n))$

C. $f(n) \neq O(g(n))$ *and* $g(n) \neq O(f(n))$

D. $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$

gatexamin.in

1.13 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
1	Introduction to Data Structures & Algorithms	https://www.youtube.com/watch?v=dykuS844oKA	
2	Analysis Framework	https://www.youtube.com/watch?v=sESoRbUQvDs	
3	Asymptotic Notations	https://www.youtube.com/watch?v=VX1P0fgjc5s	
4	Big O Notation	https://www.youtube.com/watch?v=0MgPdYCOZ9s	
5	Big Omega Notation	https://www.youtube.com/watch?v=zWr6t0jInwg	
6	Big Theta Notation	https://www.youtube.com/watch?v=XapTeNqtgrM	
7	Properties of Big O, Big Ω & Big Θ	https://www.youtube.com/watch?v=NjOa56Tm5XY	
8	Little o & Little ω Notation	https://www.youtube.com/watch?v=7u86WyVOK1s	

Lec. No	Title	URL	QR Code
9	Relationship Between O, Ω, Θ, o & ω	https://www.youtube.com/watch?v=88_k3WpTtGs	
10	Worst, Best & Average Case Analysis	https://www.youtube.com/watch?v=6lEOsxndxGM	
11	Analysis of Non-Recursive Algorithms	https://www.youtube.com/watch?v=ZK7P4g9c1VE	
12	Recurrence Relations – Base Case Method	https://www.youtube.com/watch?v=0SCqFIR7quU	
13	Recurrence Relations – Recursion Tree Method	https://www.youtube.com/watch?v=pX74ra6-gS4	
14	Master Theorem	https://www.youtube.com/watch?v=7bU3EUEWXOU	
15	Analysis of Recursive Algorithms	https://www.youtube.com/watch?v=IQc-55H57uM	
16	Practice Problems on Time Complexity	https://www.youtube.com/watch?v=znprYcc-OF4	

Lec. No	Title	URL	QR Code
17	GATE PYQs on Time Complexity	https://www.youtube.com/watch?v=IVUnUrgzQ00	

gatexam.in

Chapter 2

Stacks

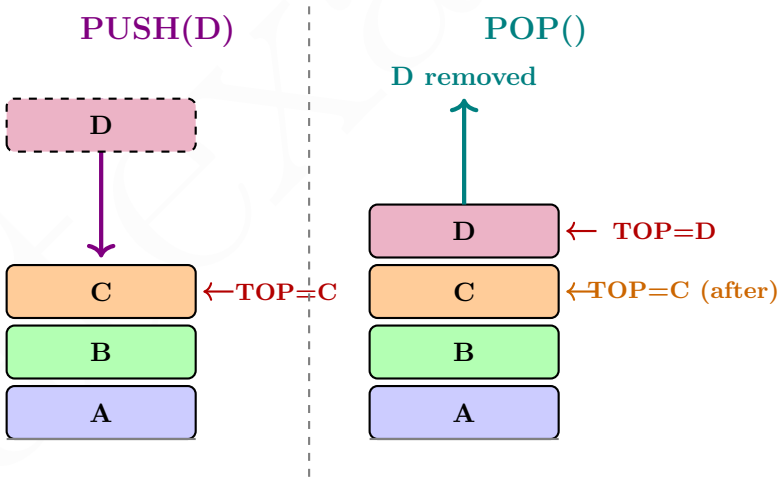
2.1 Core Operations

Stack and the LIFO Principle

A stack is a linear data structure where elements are added and removed from only one end, called the **top**. The rule followed is **Last In, First Out (LIFO)**, which means the element added most recently is the one that gets removed first. Think of a stack of plates on a dining table. You always place a new plate on the top, and you always take a plate from the top. You never take from the bottom or middle.

Example 178: Real World Example of LIFO

Think of a stack of books on a desk. If you place book A first, then book B, book C, then book D on top, the only book you can pick up right now is book D. After removing D, you can reach C. After removing C, you can reach B, and finally A. This is exactly LIFO.



Core Stack Operations

A stack supports three main operations. All three take constant time, $O(1)$, meaning no matter how many elements are in the stack, these operations always take the same amount of time.

- **Push(x):** Add element x on the top of the stack.
- **Pop():** Remove and return the element currently on top of the stack.
- **Peek() / Top():** Look at the element on top, without removing it.

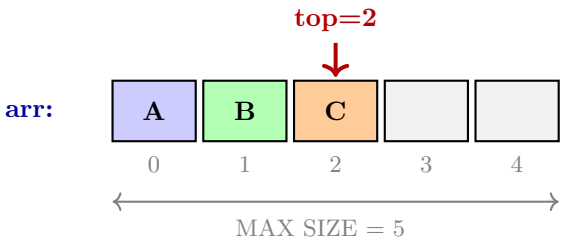
Two more helper operations are usually included.

- **IsEmpty():** Check if the stack has no elements.
- **IsFull():** Check if the stack has no more room (only for array-based implementation).

2.1.1 Array Implementation

Stack Using an Array

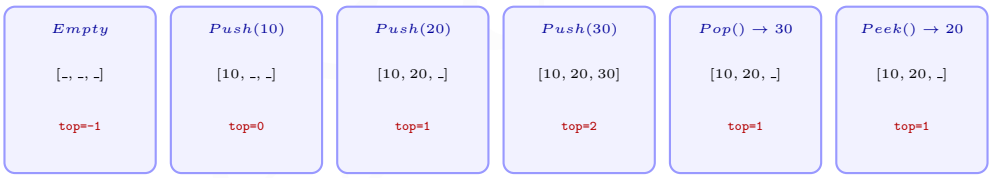
We can implement a stack using a fixed-size array. We keep a variable called `top`, which stores the index of the topmost element in the array. Initially, before any element is pushed, `top` is set to -1, which means the stack is empty. When we push an element, we first increase `top` by 1, and then store the element at index `top`. When we pop, we return the element at index `top`, and then decrease `top` by 1.



Example 179: Step by Step: Push and Pop Trace

Suppose we have an empty stack with maximum size 5. We perform: Push(10), Push(20), Push(30), Pop(), Peek().

- Start: `top = -1`, array = [_, _, _, _, _]
- Push(10): `top = 0`, array = [10, _, _, _, _]
- Push(20): `top = 1`, array = [10, 20, _, _, _]
- Push(30): `top = 2`, array = [10, 20, 30, _, _]
- Pop(): returns 30, `top = 1`, array = [10, 20, _, _, _]
- Peek(): returns 20 (top element), `top` stays at 1



Algorithm, Pseudocode and Python Implementation

Algorithm: Stack-Push(S, x)

```
1: if S.top = S.maxsize − 1 then
2:     error "Stack Overflow"
3: else
4:     S.top ← S.top + 1
5:     S.arr[S.top] ← x
6: end if
```

Algorithm: Stack-Pop(S)

```
1: if S.top = −1 then
2:     error "Stack Underflow"
3: else
4:     x ← S.arr[S.top]
5:     S.top ← S.top − 1
6:     return x
7: end if
```

Algorithm: Stack-Peek(S)

```
1: if S.top = −1 then
2:     error "Stack is Empty"
3: else
4:     return S.arr[S.top]
5: end if
```

Algorithm: Stack-IsEmpty(S)

```
1: if S.top = −1 then
2:     return true
3: else
4:     return false
5: end if
```

Example 180: Pseudocode: Full Stack Implementation

Stack Operations (C-style Pseudocode)

```
1. Initialize(S, maxsize):
2.     S.arr = array of size maxsize
3.     S.top = -1
4.     S.maxsize = maxsize
5.
6. Push(S, x):
7.     if S.top == S.maxsize - 1:
8.         print "Stack Overflow"
9.         return
10.    S.top = S.top + 1
11.    S.arr[S.top] = x
12.
13. Pop(S):
14.    if S.top == -1:
15.        print "Stack Underflow"
16.        return -1
17.    x = S.arr[S.top]
18.    S.top = S.top - 1
19.    return x
20.
21. Peek(S):
22.    if S.top == -1:
23.        print "Stack is Empty"
24.        return -1
25.    return S.arr[S.top]
26.
27. IsEmpty(S):
```

28. return (S.top == -1)
29.
30. IsFull(S):
31. return (S.top == S.maxsize - 1)

Python implementation: stack-array.py

```
class Stack:
    def __init__(self, maxsize):
        self.arr = [None] * maxsize    # fixed size array
        self.top = -1                   # stack is empty
        self.maxsize = maxsize

    def is_empty(self):
        return self.top == -1

    def is_full(self):
        return self.top == self.maxsize - 1

    def push(self, x):
        if self.is_full():
            print("Stack Overflow")    # no space left
            return
        self.top += 1
        self.arr[self.top] = x

    def pop(self):
        if self.is_empty():
            print("Stack Underflow")    # nothing to remove
            return -1
        x = self.arr[self.top]
        self.top -= 1
        return x

    def peek(self):
        if self.is_empty():
            print("Stack is Empty")
            return -1
        return self.arr[self.top]


# --- Driver Code ---
s = Stack(5)

s.push(10)
s.push(20)
s.push(30)

print("Peek:", s.peek())     # Output: 30
print("Pop:", s.pop())       # Output: 30
print("Pop:", s.pop())       # Output: 20
print("Peek:", s.peek())     # Output: 10
print("IsEmpty:", s.is_empty())  # Output: False
```

Example 181: Why All Operations are $O(1)$

All three main stack operations, Push, Pop, and Peek, take constant time $O(1)$. This is because, in an array-based implementation, each operation only does a fixed number of steps, no loops, no searching. Push increases `top` by 1 and stores a value, which is just 2 steps. Pop reads a value and decreases `top` by 1, again just 2 steps. Peek reads one value, just 1 step. None of these steps depend on how many elements are in the stack. So, the time stays constant, no matter whether the stack has 1 element or 1000 elements.

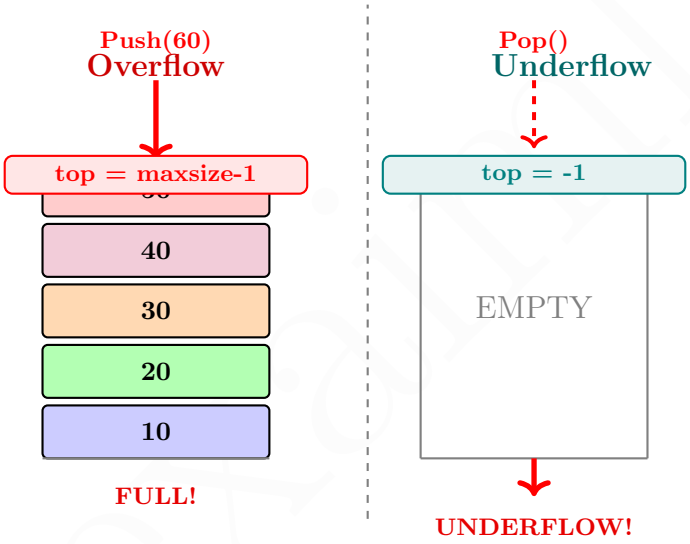
Time Complexity of Stack Operations

Operation	Time Complexity	Space
`Push(x)`	$O(1)$	$O(1)$
`Pop()`	$O(1)$	$O(1)$
`Peek()`	$O(1)$	$O(1)$
`IsEmpty()`	$O(1)$	$O(1)$
`IsFull()`	$O(1)$	$O(1)$

Example 182: Overflow and Underflow Conditions

Two important error conditions exist for an array-based stack.

- **Stack Overflow:** This happens when we try to push an element into a full stack, that means when `top == maxsize - 1`. The array has no more room.
- **Stack Underflow:** This happens when we try to pop or peek from an empty stack, that means when `top == -1`. There is no element to return.



2.2 Applications of Stacks

Why Stacks are Useful

Stacks are used in many real problems where the most recently seen item needs to be processed first. The LIFO property makes stacks perfect for expression evaluation, matching symbols, tracking function calls, and converting between expression notations. All these applications rely on the same idea, the last thing we stored is the first thing we need.

2.2.1 Expression Notations: Infix, Postfix, Prefix

Three Notations for Expressions

A mathematical expression like $A + B$ can be written in three different ways, depending on where the operator is placed relative to the operands.

- **Infix:** Operator is between operands. Example: $A + B$. This is the normal way humans write expressions. Requires parentheses and precedence rules to avoid ambiguity.
- **Postfix (Reverse Polish Notation):** Operator comes after operands. Example: $A\ B\ +$. No parentheses needed. Easy for computers to evaluate using a stack.
- **Prefix (Polish Notation):** Operator comes before operands. Example: $+\ A\ B$. Also no parentheses

needed.

Expression Notation Comparison

Notation	Example: $(A+B)*C$	Operator Position
Infix	$(A+B)*C$	Between operands
Postfix	$A\ B\ +\ C\ *$	After operands
Prefix	$*\ +\ A\ B\ C$	Before operands

2.2.2 Infix to Postfix Conversion (Shunting-Yard Algorithm)

Infix to Postfix Conversion

We use a stack to convert an infix expression to postfix. The idea is, we scan the infix expression left to right. Operands (numbers or variables) go directly to the output. Operators are pushed onto the stack, but before pushing, we first pop all operators of higher or equal precedence already in the stack, and send them to output. Opening brackets are pushed onto the stack. Closing brackets trigger popping until we find the matching opening bracket. At the end, all remaining operators in the stack are popped to output.

Operator Precedence Rules

- **Highest precedence:** `^` (exponentiation), right to left associativity
- **Medium precedence:** `*` and `/`, left to right associativity
- **Lowest precedence:** `+` and `-`, left to right associativity
- **Brackets:** `(` has lowest precedence when inside the stack

Infix to Postfix Conversion (Shunting-Yard Algorithm)

We use an operator stack to convert an infix expression to postfix by scanning the infix tokens from left to right:

- **Operands** (numbers or variables) go directly to the output.
- **Opening brackets** `(` are pushed directly onto the stack.
- **Closing brackets** `)` trigger popping all operators from the stack to the output until the matching opening bracket `(` is found. Discard both brackets.
- **Operators** (O_1) trigger popping an operator (O_2) from the top of the stack to the output if:
  1. O_2 has **strictly higher** precedence than O_1, OR
  2. O_2 has **equal** precedence to O_1 AND O_1 is **left-associative** (e.g., `+`, `-`, `*`, `/`).

  Repeat this check for the new stack top. Stop popping as soon as the condition fails, then push O_1 onto the stack. (*Note: Right-associative operators like `^` will not pop an equal-precedence operator*).
- **End of expression:** Pop all remaining operators from the stack and send them to the output.

Example 183: Step by Step: Convert $A+B*C-D$ to Postfix

Scan left to right. Stack starts empty, output is empty.

Symbol	Stack (bottom→top)	Output
A	Empty	A
$+$	$+$	A
B	$+$	$A\ B$
$*$	$+\ *$	$A\ B$
C	$+\ *$	$A\ B\ C$
$-$	$-$	$A\ B\ C\ *\ +$
D	$-$	$A\ B\ C\ *\ +\ D$
End	Empty	$A\ B\ C\ *\ +\ D\ -$

Final Postfix: $A\ B\ C\ *\ +\ D\ -$

Algorithm: Infix-To-Postfix(expression)

```
Initialize empty stack S and empty output list out
for each token t in expression (left to right) do
    if t is an operand then
        Append t to out
    else if t is ( then
        Push t onto S
    else if t is ) then
        while S is not empty and S.top ≠ ( do
            Pop from S and append to out
        end while
        Pop ( from S (discard it)
    else                                              ▷ t is an operator
        while S is not empty and precedence(S.top) ≥ precedence(t) do
            Pop from S and append to out
        end while
        Push t onto S
    end if
end for
while S is not empty do
    Pop from S and append to out
end while
return out
```

Example 184: Pseudocode: Infix to Postfix

InfixToPostfix(expr)

1. stack = empty, output = empty list
2. for each token t in expr:
3. if t is operand:
4. append t to output
5. else if t == '(':
6. push t onto stack
7. else if t == ')':
8. while stack top != '(':
9. pop and append to output

10. pop '(' from stack (discard)
11. else: (t is operator)
12. while stack not empty and prec(stack top) >= prec(t):
13. pop and append to output
14. push t onto stack
15. while stack not empty:
16. pop and append to output
17. return output

Python implementation: infix-to-postfix.py

```
def infix_to_postfix(expr):
    precedence = {'+': 1, '-': 1,
                  '*': 2, '/': 2,
                  '^': 3}
    stack  = []
    output = []

    for token in expr:
        if token.isalnum():           # operand
            output.append(token)

        elif token == '(':
            stack.append(token)

        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()               # remove '('

        else:                         # operator
            while (stack and
                   stack[-1] != '(' and
                   precedence.get(stack[-1], 0) >= precedence[token]):
                output.append(stack.pop())
            stack.append(token)

    while stack:                      # remaining operators
        output.append(stack.pop())

    return ' '.join(output)


# --- Driver Code ---
expr = ['A', '+', 'B', '*', 'C', '-', 'D']
print(infix_to_postfix(expr))
# Output: A B C * + D -
```

2.2.3 Postfix Expression Evaluation

Postfix Evaluation Using Stack

Evaluating a postfix expression using a stack is simple. We scan the postfix expression left to right. If the current token is an operand, we push it onto the stack. If the current token is an operator, we pop two operands from the stack, apply the operator, and push the result back. At the end, the stack contains exactly one value, which is the final answer.

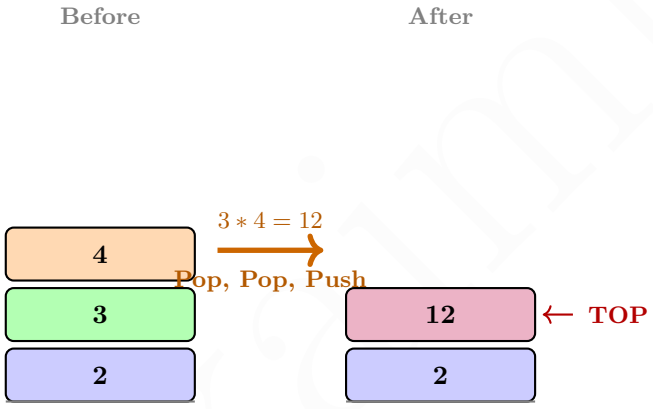
Example 185: Step by Step: Evaluate $2\ 3\ 4\ *\ +\ 5\ -$

Expression is already in postfix. Scan left to right.

Token	Action	Stack
2	Push 2	2
3	Push 3	2 3
4	Push 4	2 3 4
$*$	Pop 4 and 3, push $3 \times 4 = 12$	2 12
$+$	Pop 12 and 2, push $2 + 12 = 14$	14
5	Push 5	14 5
$-$	Pop 5 and 14, push $14 - 5 = 9$	9

Final Answer: Stack top = 9

Operator $*$ Applied on Stack



Algorithm: Evaluate-Postfix(expression)

```
Initialize empty stack S
for each token t in expression (left to right) do
    if t is an operand then
        Push t onto S
    else                                          ▷ t is an operator
        b ← Pop(S)
        a ← Pop(S)
        result ← Apply(t, a, b)
        Push result onto S
    end if
end for
return Pop(S)
```

Python implementation: postfix-eval.py

```
def evaluate_postfix(expr):
    stack = []

    for token in expr:
        if token.lstrip('-').isdigit():    # operand
            stack.append(int(token))

        else:                               # operator
            b = stack.pop()                # second operand
            a = stack.pop()                # first operand
```

```
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(a // b)

    return stack[0]    # final result


# --- Driver Code ---
expr = ['2', '3', '4', '*', '+', '5', '-']
print("Result:", evaluate_postfix(expr))
# Output: Result: 9
```

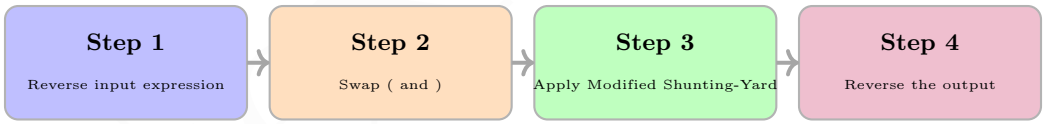
2.2.4 Infix to Prefix Conversion

Infix to Prefix Conversion

Converting infix to prefix uses a similar idea as infix to postfix, but with two key differences. First, we reverse the entire infix expression and swap each (with) and vice versa. Second, when running the Shunting-Yard algorithm on this modified expression, we **invert the equal precedence rule**:

- **Strictly Higher Precedence on Stack:** Pop operators from stack to output.
- **Strictly Lower Precedence on Stack:** Push the incoming operator onto the stack.
- **Equal Precedence (Associativity Rule):**
  - For **left-associative** operators (+, -, *, /): **Push** onto the stack (do *not* pop).
  - For **right-associative** operators (^): **Pop** the stack top to output before pushing.

At the end, we reverse the final output string to obtain the prefix expression.



Example 186: Step by Step: Convert $A + B * C$ to Prefix

Step 1: Reverse the expression: $C * B + A$
Step 2: No brackets in this example, so no swapping needed.
Step 3: Apply modified Shunting-Yard on $C * B + A$:

- $C \rightarrow$ output: C
- $* \rightarrow$ stack: $[*]$
- $B \rightarrow$ output: $C\ B$
- $+ \rightarrow$ stack top $*$ has *strictly higher* precedence than $+$, so pop $*$ to output, push $+ \rightarrow$ output: $C\ B\ *$
- $A \rightarrow$ output: $C\ B\ *\ A$
- End $\rightarrow$ pop $+ \rightarrow$ output: $C\ B\ *\ A\ +$

Step 4: Reverse: $+\ A\ *\ B\ C$
Final Prefix: $+\ A\ *\ B\ C$, which is same as $A + (B * C)$.

Algorithm: Infix-To-Prefix(expression)

1: Reverse the expression
2: Swap each `(` with `)` and each `)` with `(`
3: Process expression using Modified Shunting-Yard:
4: **if** operator op on stack top has *strictly higher* precedence than incoming op:
5: Pop stack top to output
6: **if** op on stack top and incoming op have *equal* precedence:
7: **if** op is left-associative (`+`, `-`, `*`, `/`): Push incoming op
8: **if** op is right-associative (`^`): Pop stack top to output
9: Reverse the resulting output
10: **return** the reversed output

“‘

Python implementation: infix-to-prefix.py

```
def infix_to_postfix_helper(expr):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack  = []
    output = []

    for token in expr:
        if token.isalnum():
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()
        else:
            while (stack and
                   stack[-1] != '(' and
                   precedence.get(stack[-1], 0) >= precedence[token]):
                output.append(stack.pop())
            stack.append(token)

    while stack:
        output.append(stack.pop())

    return output


def infix_to_prefix(expr):
    # Step 1: Reverse
    rev = expr[::-1]

    # Step 2: Swap brackets
    swapped = []
    for token in rev:
        if token == '(':
            swapped.append(')')
        elif token == ')':
            swapped.append('(')
        else:
            swapped.append(token)

    # Step 3: Apply postfix conversion
    postfix = infix_to_postfix_helper(swapped)

    # Step 4: Reverse result
    prefix = postfix[::-1]

    return ' '.join(prefix)


# --- Driver Code ---
```

```
expr = ['A', '+', 'B', '*', 'C']
print(infix_to_prefix(expr))
# Output: + A * B C
```

2.2.5 Prefix Expression Evaluation

Prefix Expression Evaluation Rules

To evaluate a prefix expression using a stack:

1. Scan the prefix expression from **right to left**.
2. If the symbol is an **operand** (number), push it onto the stack.
3. If the symbol is an **operator**, pop two elements from the stack:
   - Op_1 = first popped
   - Op_2 = second popped
4. Compute result = Op_1 operator Op_2 and push the result back onto the stack.
5. The final stack value is the expression's result.

Example 187: Step-by-Step: Evaluate Prefix Expression 1

Expression: $- \ * \ 4 \ 3 \ / \ 8 \ 2$

- Scan right to left:
- Push 2, Push 8 $\rightarrow$ Stack: $[2, 8]$
- Read / $\rightarrow$ Pop $Op_1 = 8, Op_2 = 2 \implies 8/2 = 4 \rightarrow$ Stack: $[4]$
- Push 3, Push 4 $\rightarrow$ Stack: $[4, 3, 4]$
- Read $*$ $\rightarrow$ Pop $Op_1 = 4, Op_2 = 3 \implies 4 * 3 = 12 \rightarrow$ Stack: $[4, 12]$
- Read $-$ $\rightarrow$ Pop $Op_1 = 12, Op_2 = 4 \implies 12 - 4 = 8 \rightarrow$ Stack: $[8]$

Final Result: 8

Example 188: Step-by-Step: Evaluate Prefix Expression 2

Expression: $+ \ * \ 2 \ 5 \hat{} 3 \ 2$

- Scan right to left:
- Push 2, Push 3 $\rightarrow$ Stack: $[2, 3]$
- Read $\hat{}$ $\rightarrow$ Pop $Op_1 = 3, Op_2 = 2 \implies 3^2 = 9 \rightarrow$ Stack: $[9]$
- Push 5, Push 2 $\rightarrow$ Stack: $[9, 5, 2]$
- Read $*$ $\rightarrow$ Pop $Op_1 = 2, Op_2 = 5 \implies 2 * 5 = 10 \rightarrow$ Stack: $[9, 10]$
- Read $+$ $\rightarrow$ Pop $Op_1 = 10, Op_2 = 9 \implies 10 + 9 = 19 \rightarrow$ Stack: $[19]$

Final Result: 19

2.2.6 Prefix to Infix Conversion

How to Convert Prefix to Infix

1. Scan the prefix expression from **right to left**.
2. If the current token is an operand, push it onto the stack as-is.
3. If the current token is an operator, pop two values: call the **first** popped value A and the **second** popped value B. Form the string $(A \text{ op } B)$ and push this back onto the stack.
4. After the scan ends, the stack contains exactly one string — this is the fully bracketed infix expression.

Example 189: Step-by-Step: Convert $+ \ * \ A \ B \ / \ C \ D$ to Infix

- Scan right to left:
- Push $D, C \rightarrow$ Stack: $[D, C]$
- Read $/ \rightarrow$ Pop $A = C, B = D \implies$ Push $(C/D) \rightarrow$ Stack: $[(C/D)]$
- Push $B, A \rightarrow$ Stack: $[(C/D), B, A]$
- Read $* \rightarrow$ Pop $A = A, B = B \implies$ Push $(A * B) \rightarrow$ Stack: $[(C/D), (A * B)]$
- Read $+ \rightarrow$ Pop $A = (A * B), B = (C/D) \implies$ Push $((A * B) + (C/D))$

Final Infix: $(A * B) + (C/D)$

Example 190: Step-by-Step: Convert $- \ + \ A \ * \ B \ C \hat{} \ D \ E$ to Infix

- Scan right to left:
- Push $E, D \rightarrow$ Stack: $[E, D]$
- Read $\hat{} \rightarrow$ Pop $A = D, B = E \implies$ Push $(D \hat{} E) \rightarrow$ Stack: $[(D \hat{} E)]$
- Push $C, B \rightarrow$ Stack: $[(D \hat{} E), C, B]$
- Read $* \rightarrow$ Pop $A = B, B = C \implies$ Push $(B * C) \rightarrow$ Stack: $[(D \hat{} E), (B * C)]$
- Push $A \rightarrow$ Stack: $[(D \hat{} E), (B * C), A]$
- Read $+ \rightarrow$ Pop $A = A, B = (B * C) \implies$ Push $(A + (B * C)) \rightarrow$ Stack: $[(D \hat{} E), (A + (B * C))]$
- Read $- \rightarrow$ Pop $A = (A + (B * C)), B = (D \hat{} E) \implies$ Push $((A + (B * C)) - (D \hat{} E))$

Final Infix: $((A + (B * C)) - (D \hat{} E))$

2.2.7 Postfix to Prefix Conversion

How to Convert Postfix to Prefix

1. Scan the postfix expression from **left to right**.
2. If the current token is an operand, push it onto the stack as-is.
3. If the current token is an operator, pop two values: call the **first** popped value B and the **second** popped value A. Form the string op A B (operator written first) and push this back onto the stack.
4. After the scan ends, the stack contains exactly one string — this is the prefix expression.

Example 191: Step-by-Step: Convert $A\ B\ +\ C\ D\ -\ *$ to Prefix

- Scan left to right:
- Push $A, B \rightarrow$ Stack: $[A, B]$
- Read $+ \rightarrow$ Pop $B = B, A = A \implies$ Push $+AB \rightarrow$ Stack: $[+AB]$
- Push $C, D \rightarrow$ Stack: $[+AB, C, D]$
- Read $- \rightarrow$ Pop $B = D, A = C \implies$ Push $-CD \rightarrow$ Stack: $[+AB, -CD]$
- Read $* \rightarrow$ Pop $B = -CD, A = +AB \implies$ Push $*+AB-CD$

Final Prefix: $*\ +\ A\ B\ -\ C\ D$

Example 192: Step-by-Step: Convert $A\ B\ C\ *\ +\ D\ E\ F\ -\ /\ -$ to Prefix

- Scan left to right:
- Push $A, B, C \rightarrow$ Stack: $[A, B, C]$
- Read $* \rightarrow$ Pop $B = C, A = B \implies$ Push $*BC \rightarrow$ Stack: $[A, *BC]$
- Read $+ \rightarrow$ Pop $B = *BC, A = A \implies$ Push $+A*BC \rightarrow$ Stack: $[+A*BC]$
- Push $D, E, F \rightarrow$ Stack: $[+A*BC, D, E, F]$
- Read $- \rightarrow$ Pop $B = F, A = E \implies$ Push $-EF \rightarrow$ Stack: $[+A*BC, D, -EF]$
- Read $/ \rightarrow$ Pop $B = -EF, A = D \implies$ Push $/D-EF \rightarrow$ Stack: $[+A*BC, /D-EF]$
- Read $- \rightarrow$ Pop $B = /D-EF, A = +A*BC \implies$ Push $-+A*BC/D-EF$

Final Prefix: $-\ +\ A\ *\ B\ C\ /\ D\ -\ E\ F$

2.2.8 Prefix to Postfix Conversion

How to Convert Prefix to Postfix

1. Scan the prefix expression from **right to left**.
2. If the current token is an operand, push it onto the stack as-is.
3. If the current token is an operator, pop two values: call the **first** popped value A and the **second** popped value B. Form the string $A\ B$ op (operator written last) and push this back onto the stack.
4. After the scan ends, the stack contains exactly one string — this is the postfix expression.

Example 193: Step-by-Step: Convert $*\ +\ A\ B\ -\ C\ D$ to Postfix

- Scan right to left:
- Push $D, C \rightarrow$ Stack: $[D, C]$
- Read $- \rightarrow$ Pop $A = C, B = D \implies$ Push $CD- \rightarrow$ Stack: $[CD-]$
- Push $B, A \rightarrow$ Stack: $[CD-, B, A]$
- Read $+ \rightarrow$ Pop $A = A, B = B \implies$ Push $AB+ \rightarrow$ Stack: $[CD-, AB+]$
- Read $* \rightarrow$ Pop $A = AB+, B = CD- \implies$ Push $AB+CD-*$

Final Postfix: $A\ B\ +\ C\ D\ -\ *$

Example 194: Step-by-Step: Convert $- \; / \; * \; A \; B \; C \; + \; D \; E$ to Postfix

- Scan right to left:
- Push $E, D \rightarrow$ Stack: $[E, D]$
- Read $+ \rightarrow$ Pop $A = D, B = E \implies$ Push $DE+ \rightarrow$ Stack: $[DE+]$
- Push $C, B, A \rightarrow$ Stack: $[DE+, C, B, A]$
- Read $* \rightarrow$ Pop $A = A, B = B \implies$ Push $AB* \rightarrow$ Stack: $[DE+, C, AB*]$
- Read $/ \rightarrow$ Pop $A = AB*, B = C \implies$ Push $AB*C/ \rightarrow$ Stack: $[DE+, AB*C/]$
- Read $- \rightarrow$ Pop $A = AB*C/, B = DE+ \implies$ Push $AB*C/DE+-$

Final Postfix: $A \; B \; * \; C \; / \; D \; E \; + \; -$

2.2.9 Balanced Parentheses / Brackets Check

Balanced Brackets Check

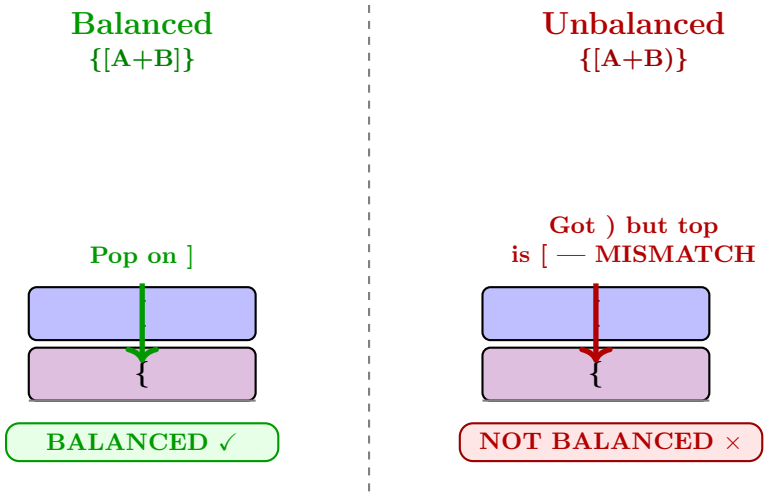
A string of brackets is balanced if every opening bracket has a corresponding closing bracket, in the correct order, and no closing bracket appears before its matching opening bracket. We use a stack to check this. We scan the string left to right. For every opening bracket `(`, `[`, or {, we push it onto the stack. For every closing bracket, we check if the top of the stack has the matching opening bracket. If yes, we pop it. If no, the string is not balanced. At the end, if the stack is empty, the string is balanced.

Example 195: Tracing Balanced Check on $\{[A + B] * (C - D)\}$

Each character is scanned left to right.

Token	Action	Stack
{	Push {	{
`[`	Push [	{[
`A`	Not a bracket (skip)	{[
`+`	Not a bracket (skip)	{[
`B`	Not a bracket (skip)	{[
`]`	Top is [— match — Pop	{
`*`	Not a bracket (skip)	{
`(`	Push (	{(
...	...	...
`)`	Top is (— match — Pop	{
}	Top is { — match — Pop	`Empty`

Stack is empty at end ⇒ BALANCED



Algorithm: Is-Balanced(expression)

```
Initialize empty stack S
for each character c in expression do
    if c ∈ { (, [, { } then
        Push c onto S
    else if c ∈ { ), ], } } then
        if S is empty then
            return false                          ▷ No matching opener
        end if
        top ← Pop(S)
        if (c = ) and top ≠ () or (c = ] and top ≠ [) or (c = } and top ≠ {) then
            return false                          ▷ Mismatch
        end if
    end if
end for
return S is empty
```

Python implementation: balanced-brackets.py

```python
def is_balanced(expr):
    stack   = []
    opening = {'(', '[', '{'}
    closing = {')': '(', ']': '[', '}': '{'}

    for ch in expr:
        if ch in opening:
            stack.append(ch)          # push opener

        elif ch in closing:
            if not stack:
                return False          # no opener to match
            top = stack.pop()
            if top != closing[ch]:    # wrong match
                return False

    return len(stack) == 0            # must be empty at end


# --- Driver Code ---
tests = [
    "{[A+B]*(C-D)}",
    "((A+B)",
    "{[A+B)}",
    "(A+(B*C))"
]

for t in tests:
    result = "Balanced" if is_balanced(t) else "NOT Balanced"
```

```
    print(f"{t:20s} -> {result}")

# Output:
# {[A+B]*(C-D)}        -> Balanced
# ((A+B)               -> NOT Balanced
# {[A+B)}              -> NOT Balanced
# (A+(B*C))            -> Balanced
```

2.2.10 Recursion and Call Stack Frames

How Recursion Uses the Call Stack

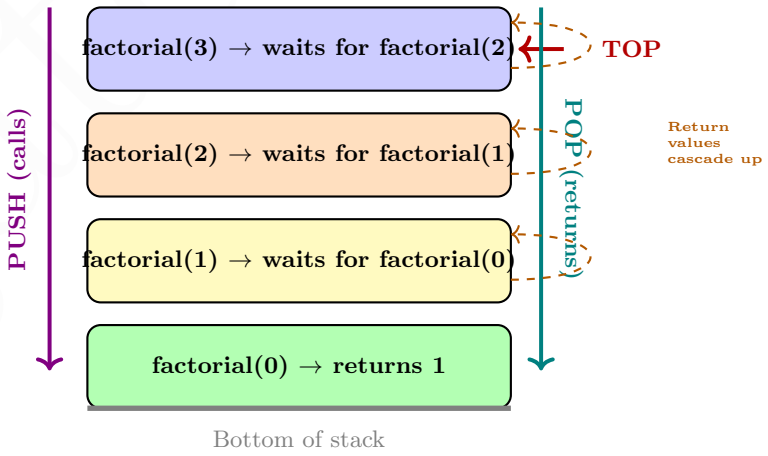
Every time a function calls itself (or calls another function), the computer needs to remember where to come back, what local variables were in use, and what the current values were. For this, the computer uses a special stack, called the **call stack**. Each time a function is called, a block of memory called a **stack frame** (or **activation record**) is pushed onto the call stack. This frame stores the local variables, parameters, and the return address. When the function finishes, its frame is popped off the call stack, and control returns to the previous frame.

Example 196: What a Stack Frame Contains

Each time a function is called, the system pushes a frame that stores:

- The function's local variables and parameters
- The return address (where to go after this function finishes)
- The return value (what the function should give back to the caller)

Call Stack for `factorial(3)`



Example 197: Tracing Call Stack for `factorial(3)`

Calls going DOWN (pushing frames):

- `factorial(3)` is called → pushes frame, calls `factorial(2)`
- `factorial(2)` is called → pushes frame, calls `factorial(1)`
- `factorial(1)` is called → pushes frame, calls `factorial(0)`

- `factorial(0)` is called → pushes frame, hits base case, returns 1

Returns going UP (popping frames):

- `factorial(0)` returns 1 → frame popped
- `factorial(1)` gets 1, returns $1 \times 1 = 1$ → frame popped
- `factorial(2)` gets 1, returns $2 \times 1 = 2$ → frame popped
- `factorial(3)` gets 2, returns $3 \times 2 = 6$ → frame popped

Algorithm: Factorial(n)

```
if n = 0 then
    return 1                                  ▷ Base case, frame popped here
else
    return n × Factorial(n − 1)               ▷ New frame pushed for each call
end if
```

Example 198: Pseudocode: Recursion with Frame Concept

Factorial(n):

1. **Push frame:** store n, return address
2. if n == 0:
3. result = 1
4. else:
5. result = n * Factorial(n-1) // new frame pushed here
6. **Pop frame:** send result back to caller
7. return result

Python implementation: recursion-call-stack.py

```python
import sys

def factorial(n, depth=0):
    indent = "  " * depth
    print(f"{indent}factorial({n}) called"
          f"  [stack depth = {depth}]")

    if n == 0:
        print(f"{indent}factorial(0) returns 1  [BASE CASE]")
        return 1

    result = n * factorial(n - 1, depth + 1)

    print(f"{indent}factorial({n}) returns {result}"
          f"  [frame popped]")
    return result

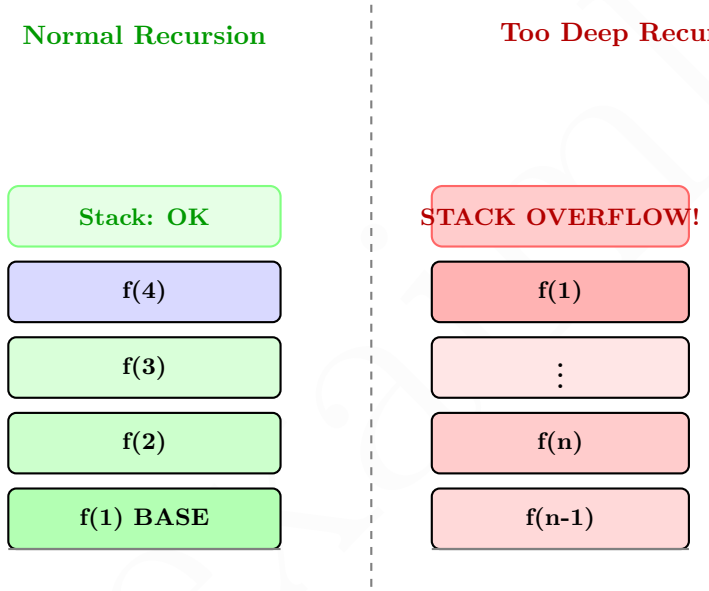

# --- Driver Code ---
print("=" * 50)
ans = factorial(4)
print("=" * 50)
print("Final Answer:", ans)

# Output (abbreviated):
# factorial(4) called  [stack depth = 0]
```

```
#   factorial(3) called  [stack depth = 1]
#     factorial(2) called  [stack depth = 2]
#       factorial(1) called  [stack depth = 3]
#         factorial(0) called  [stack depth = 4]
#         factorial(0) returns 1  [BASE CASE]
#       factorial(1) returns 1  [frame popped]
#     factorial(2) returns 2  [frame popped]
#   factorial(3) returns 6  [frame popped]
# factorial(4) returns 24  [frame popped]
# Final Answer: 24
```

Stack Overflow Due to Deep Recursion

Since each recursive call pushes a new frame onto the call stack, very deep recursion can fill up all available stack space. When the call stack runs out of space, a **Stack Overflow** error occurs. This is why recursive functions must always have a base case that stops recursion, and why the recursion depth matters in practice. In Python, the default recursion depth limit is around 1000 calls.



2.3 Practice Problems

Problem 63 *A stack is implemented with an array of* $A[0 \ldots N-1]$ *and a variable* `'pos'`*. The push and pop operations are defined by the following code:*

```
push (x)
    A[pos] <- x
    pos <- pos - 1
end push

pop()
    pos <- pos + 1
    return A[pos]
end pop
```

Which of the following will initialize an empty stack with capacity N *for the above implementation?*

A. $pos \leftarrow -1$

B. $pos \leftarrow 0$

C. $pos \leftarrow N$

D. $pos \leftarrow N - 1$

Problem 64 *Convert the prefix expression to infix:*

$$- * + ABC * - DE + FG$$

A. $(A - B) * C + (D * E) - (F + G)$

B. $(A + B) * C - (D - E) * (F - G)$

C. $(A + B - C) * (D - E) * (F + G)$

D. $(((A + B) * C) - ((D - E) * (F + G)))$

Problem 65 *Choose the equivalent prefix form of the following expression:*

$$(a + (b - c)) * ((d - e)/(f + g - h))$$

A. $* + a - bc/ - de - + fgh$

B. $* + a - bc - /de - + fgh$

C. $* + a - bc/ - ed - + fgh$

D. $* + ab - c/ - de - + fgh$

Problem 66 *If the sequence of operations –* `push(1)`*,* `push(2)`*,* `pop`*,* `push(1)`*,* `push(2)`*,* `pop`*,* `pop`*,* `pop`*,* `push(2)`*,* `pop` *are performed on a stack, the sequence of popped out values is:*

A. 2, 2, 1, 1, 2

B. 2, 2, 1, 2, 2

C. 2, 1, 2, 2, 1

D. 2, 1, 2, 2, 2

Problem 67 *The expression* $1 * 2 \wedge 3 * 4 \wedge 5 * 6$ *will be evaluated as:*

A. 3230

B. 16230

C. 49152

D. 173458

Problem 68 *Stack A has the entries a, b, c (with a on top). Stack B is empty. An entry popped out of stack A can be printed immediately or pushed to stack B. An entry popped out of the stack B can be only be printed. In this arrangement, which of the following permutations of a, b, c are ALWAYS possible irrespective of positions of b and c?*

A. $b\ a\ c$

B. $b\ c\ a$

C. $c\ a\ b$

D. $a\ b\ c$

Problem 69 ***[MCQ]*** *The following algorithm is executed on an initially empty stack S:*

Algorithm: StackOps(n)

```
1: for i = 1 to n do
2:     Push(i) onto S
3: end for
4: for i = 1 to n/2 do
5:     Pop from S
6: end for
7: return Peek(S)
```

For $n = 8$, what value does `Peek` *return?*

(A) 4

(B) 5

(C) 3

(D) 6

Problem 70 ***[MSQ]*** *An initially empty stack supports* `push` *and* `pop` *operations. The input sequence $1, 2, 3, 4$ is pushed in this order. Pops can happen at any point. Which of the following output sequences are POSSIBLE?*

(A) 2, 4, 3, 1

(B) 4, 3, 2, 1

(C) 1, 3, 2, 4

(D) 2, 3, 1, 4

Problem 71 ***[MCQ]*** *Consider the following pseudocode, where* `S` *is an initially empty stack of integers:*

```
for i = 1 to 5:
    if i % 2 == 0:
        push(S, i)
    else:
        if not isEmpty(S):
            pop(S)
```

What is the content of the stack from bottom to top after execution?

(A) 2, 4

(B) 4

(C) 2

(D) (empty)

Problem 72 ***[MCQ]*** *A postfix expression is being evaluated using a stack. After processing the token sequence $4\ 2\ 5\ *\ +\ 3$, and just before the next operator token is read, what are the top two elements of the stack from top to bottom?*

(A) 3, 14

(B) 14, 3

(C) 3, 10

(D) 10, 3

Problem 73 ***[NAT]*** *Consider the postfix expression:*

$$9 \quad 3 \quad / \quad 4 \quad 2 \quad * \quad + \quad 6 \quad 2 \quad / \quad -$$

The number of push operations performed on the stack during evaluation is _____.

Problem 74 ***[MSQ]*** *Consider the infix expression:*

$$(A + B) * (C - D)/E$$

where $*$ and $/$ have the same precedence (higher than $+$ and $-$), and all operators are left associative. Which of the following are correct?

(A) The postfix expression is $A\ B\ +\ C\ D\ -\ *\ E\ /$

(B) The prefix expression is $/\ *\ +\ A\ B\ -\ C\ D\ E$

(C) The postfix expression is $A\ B\ +\ C\ D\ *\ -\ E\ /$

(D) The number of operators in the postfix form equals the number in the infix form

Problem 75 ***[MCQ]*** *During the conversion of $A + B * C \wedge D - E$ to postfix (where $\wedge$ has highest precedence, right associative; $*$ has medium; $+$ and $-$ have lowest, left associative), at what point does the operator $+$ get added to the output?*

(A) When $-$ is encountered

(B) When the expression ends

(C) When $*$ is encountered

(D) When $\wedge$ is encountered

Problem 76 ***[NAT]*** *Consider the string S = "{{[()]}([])}". During the execution of the stack-based balanced bracket checking algorithm, the maximum size (number of elements) reached by the stack at any point during processing is _____.*

Problem 77 ***[NAT]*** *Consider the following algorithm that operates on an integer stack S:*

Algorithm: Transform(S)

```
1: T ← empty stack
2: while S is not empty do
3:     x ← Pop(S)
4:     if x > 0 then
5:         Push(T, x * 2)
6:     else
7:         Push(T, x)
8:     end if
9: end while
10: while T is not empty do
11:     Push(S, Pop(T))
12: end while
```

If S contains, from bottom to top, $[-2, 3, -1, 4]$, after `Transform(S)` the sum of the stack content is ?

Problem 78 ***[MCQ]*** *A single array $A[0 \ldots N-1]$ is used to implement two stacks. Stack 1 grows from index 0 upwards, and Stack 2 grows from index $N-1$ downwards. Variables `top1` and `top2` track the topmost elements. Initially, `top1 = -1` and `top2 = N`. The condition for the combined structure being full (no more push on either stack is possible) is:*

(A) `top1 == N/2` and `top2 == N/2 + 1`

(B) `top1 + 1 == top2`

(C) `top1 == top2`

(D) `top1 == N - 1` or `top2 == 0`

Problem 79 ***[MSQ]*** *Consider using a stack to simulate recursion for the following algorithm iteratively. Which of the following items MUST be saved in the stack frame for each recursive call?*

Algorithm: RecursiveSum(A, lo, hi)

```
1: if lo = hi then
2:     return A[lo]
3: end if
4: mid ← (lo + hi)/2
5: return RecursiveSum(A, lo, mid) + RecursiveSum(A, mid + 1, hi)
```

(A) The values of `lo` *and* `hi` *for the current call*

(B) The value of `mid` *computed in the current call*

(C) The return address indicating where to return after the current call completes

(D) The entire array `A`

Problem 80 ***[MCQ]*** *A function g is defined recursively on a stack of integers as follows:*

$$g(\emptyset) = 0, \qquad g(push(S, x)) = g(S) + x^2$$

If the stack S contains, from bottom to top, $[1, -2, 3, -4]$, what is $g(S)$?

(A) 10

(B) -30

(C) -2

(D) 30

Problem 81 ***[NAT]*** *Consider the following algorithm applied to a stack S containing $[1, 2, 3, 4, 5]$ from bottom to top:*

Algorithm: CountPositive(S)

```
1: T ← empty stack, count ← 0
2: while S is not empty do
3:     x ← Pop(S)
4:     if x > 2 then
5:         count ← count + 1
6:     end if
7:     Push(T, x)
8: end while
9: while T is not empty do
10:    Push(S, Pop(T))
11: end while
12: return count
```

The restored stack S contains _____ elements greater than 2.

Problem 82 ***[MCQ]*** *Consider the following algorithm that takes a positive integer n and uses a stack:*

Algorithm: Mystery(n)

```
1: S ← empty stack
2: while n > 0 do
3:     Push(S, n mod 2)
4:     n ← ⌊n/2⌋
5: end while
6: while S is not empty do
7:     print Pop(S)
8: end while
```

What is printed when this algorithm is run with $n = 2048$?

(A) 100000000000

(B) 11111111111

(C) 100000000001

(D) 10000000000

Problem 83 ***[MCQ]*** *Consider the following pseudocode that evaluates a condition on a stack S containing n elements:*

```
T = empty stack
flag = True
while S is not empty:
    x = pop(S)
    if T is not empty and peek(T) > x:
        flag = False
    push(T, x)
```

What property of the original stack S (read from bottom to top) does `flag = True` indicate?

(A) The stack is sorted in descending order from bottom to top

(B) The stack is sorted in ascending order from bottom to top

(C) All elements in the stack are equal

(D) The stack contains no duplicate elements

Problem 84 ***[NAT]*** *Consider the following algorithm operating on a stack S:*

Algorithm: Mystery(n)

```
1: S ← empty stack
2: for i ← 1 to n do
3:     Push(S, i)
4: end for
5: result ← 1
6: for i ← 1 to n do
7:     result ← result* Pop(S)
8: end for
9: return result
```

What is the value returned by `Mystery(7)`? _____

Problem 85 ***[NAT]*** *Consider the following algorithm:*

Algorithm: CountOps(seq)

```
1: S ← empty stack
2: count ← 0
3: for each character ch in seq do
4:     if ch = 'P' then
5:         Push(S, count)
6:         count ← count + 1
7:     else
8:         if S is not empty then
9:             Pop(S)
10:        end if
11:    end if
12: end for
13: return size of S
```

What is the value returned by `CountOps("PPPQPQQ")`? _____

Problem 86 ***[MCQ]*** *Consider the following recursive algorithm:*

Algorithm: Func(n)

```
1: if n ≤ 0 then
2:     return 0
3: end if
4: return Func(n − 1) + Func(n − 1) +n
```

The maximum depth of the call stack (number of simultaneous active frames) when `Func(4)` *is called is:*

(A) 4

(B) 5

(C) 8

(D) 16

Problem 87 ***[NAT]*** *Consider the following recursive algorithm:*

Algorithm: StackPrint(S)

```
1: if S is empty then
2:     return
3: end if
4: x ← Pop(S)
5: StackPrint(S)
6: print x
7: Push(S, x)
```

If stack S contains from bottom to top $[10, 20, 30, 40]$ *and we call* `StackPrint(S)`*, the first number printed is* _____.

Problem 88 ***[MCQ]*** *The following recursive algorithm operates on a stack:*

Algorithm: Mystery(S)

```
1: if S is empty then
2:     return 0
3: end if
4: top ← Pop(S)
5: result ← Mystery(S)
6: Push(S, top)
7: return result + top
```

What does this algorithm do?

(A) Returns the maximum element in the stack

(B) Returns the sum of all elements in the stack, leaving the stack unchanged

(C) Returns the minimum element in the stack

(D) Reverses the stack and returns the new top

Problem 89 ***[MCQ]*** *Consider the following algorithm that uses a stack to process a list:*

Algorithm: Process(lst)

```
1: S ← empty stack, Result ← empty list
2: for each x in lst do
3:     while S is not empty AND top of S < x do
4:         Append Pop(S) to Result
5:     end while
6:     Push(S, x)
7: end for
8: while S is not empty do
9:     Append Pop(S) to Result
10: end while
11: return Result
```

What does `Process([3, 1, 4, 1, 5, 9, 2, 6])` *return?*

(A) `[1, 3, 1, 4, 2, 5, 6, 9]`

(B) `[1, 1, 3, 4, 2, 5, 6, 9]`

(C) `[1, 3, 1, 4, 5, 2, 6, 9]`

(D) `[3, 1, 4, 1, 5, 9, 2, 6]`

Problem 90 ***[NAT]*** *The following recursive algorithm computes a value on a stack S (top is the last element):*

Algorithm: F(S)

```
1: if S is empty then
2:     return 0
3: end if
4: top ← Pop(S)
5: val ← max(F(S), 0) + top
6: Push(S, top)
7: return val
```

If the stack contains, from bottom to top, $[2, -3, 2, -1, 2]$*, what is* `F(S)`*?* _____

gateqaim.in

2.4 GATE PYQs

GATEPYQ 37 (GATE CSE 2025) *Consider a stack data structure into which we can PUSH and POP records. Assume that each record pushed in the stack has a positive integer key and that all keys are distinct. We wish to augment the stack data structure with an $O(1)$ time MIN operation that returns a pointer to the record with the smallest key present in the stack*

1) *without deleting the corresponding record, and*

2) *without increasing the complexities of the standard stack operations.*

Which one or more of the following approach(es) can achieve it?

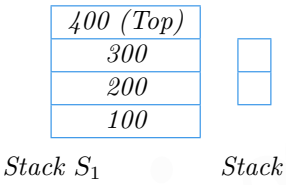
A. *Keep with every record in the stack, a pointer to the record with the smallest key below it.*

B. *Keep a pointer to the record with the smallest key in the stack.*

C. *Keep an auxiliary array in which the key values of the records in the stack are maintained in sorted order.*

D. *Keep a Min-Heap in which the key values of the records in the stack are maintained.*

GATEPYQ 38 (GATE CSE 2024) *Let S_1 and S_2 be two stacks. S_1 has capacity of 4 elements. S_2 has capacity of 2 elements. S_1 already has 4 elements: 100, 200, 300, and 400, whereas S_2 is empty, as shown below.*



Stack S_1 *Stack S_2*

Only the following three operations are available:

- `PushToS2`*: Pop the top element from S_1 and push it on S_2.*
- `PushToS1`*: Pop the top element from S_2 and push it on S_1.*
- `GenerateOutput`*: Pop the top element from S_1 and output it to the user.*

Note that the pop operation is not allowed on an empty stack and the push operation is not allowed on a full stack. Which of the following output sequences can be generated by using the above operations?

A. *100, 200, 400, 300*

B. *200, 300, 400, 100*

C. *400, 200, 100, 300*

D. *300, 200, 400, 100*

GATEPYQ 39 (GATE CSE 2024) *Consider the operator precedence and associativity rules for the integer arithmetic operators given in the table below.*

Operator	***Precedence***	***Associativity***
$+$	*Highest*	*Left*
$-$	*High*	*Right*
$*$	*Medium*	*Right*
$/$	*Low*	*Right*

The value of the expression $3+1+5*2/7+2-4-7-6/2$ as per the above rules is _____.

GATEPYQ 40 (GATE CSE 2015 SET-3) *The result evaluating the postfix expression:*

$$10 \quad 5 \quad + \quad 60 \quad 6 \quad / \quad * \quad 8 \quad -$$

is

A. *284*

B. *213*

C. 142

D. 71

GATEPYQ 41 (GATE CSE 2007) *The following postfix expression with single digit operands is evaluated using a stack:*

$$8 \quad 2 \quad 3 \quad \wedge \quad / \quad 2 \quad 3 \quad * \quad + \quad 5 \quad 1 \quad * \quad -$$

Note that $\wedge$ is the exponentiation operator. The top two elements of the stack after the first $*$ is evaluated are:

A. 6, 1

B. 5, 7

C. 3, 2

D. 1, 5

GATEPYQ 42 (GATE IT 2005) *A function f defined on stacks of integers satisfies the following properties: $f(\emptyset) = 0$ and $f(push(S, i)) = \max(f(S), 0) + i$ for all stacks S and integers i. If a stack S contains the integers 2, -3, 2, -1, 2 in order from bottom to top, what is $f(S)$?*

A. 6

B. 4

C. 3

D. 2

GATEPYQ 43 (GATE IT 2004) *A program attempts to generate as many permutations as possible of the string, 'abcd' by pushing the characters a, b, c, d in the same order onto a stack, but it may pop off the top character at any time. Which one of the following strings CANNOT be generated using this program?*

A. abcd

B. dcba

C. cbad

D. cabd

GATEPYQ 44 (GATE CSE 2004) *Assume that the operators $+$, $-$, $\times$ are left associative and $\wedge$ is right associative. The order of precedence (from highest to lowest) is $\wedge, \times, +, -$. The postfix expression corresponding to the infix expression $a + b \times c - d \wedge e \wedge f$ is:*

A. $abc \times +def \wedge \wedge -$

B. $abc \times +de \wedge f \wedge -$

C. $ab + c \times d - e \wedge f \wedge$

D. $a \times bc \wedge \wedge def$

GATEPYQ 45 (GATE CSE 2004) *The best data structure to check whether an arithmetic expression has balanced parentheses is a:*

A. Queue

B. Stack

C. Tree

D. List

GATEPYQ 46 (GATE CSE 2004) *A single array $A[1 \ldots MAXSIZE]$ is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top_1 and top_2 ($top_1 < top_2$) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, the condition for "stack full" is:*

A. ($top_1 = MAXSIZE/2$) and ($top_2 = MAXSIZE/2 + 1$)

B. $top_1 + top_2 = MAXSIZE$

C. $(top_1 = MAXSIZE/2)$ *or* $(top_2 = MAXSIZE)$

D. $top_1 = top_2 - 1$

GATEPYQ 47 (GATE CSE 2003) *Let S be a stack of size $n \geq 1$. Starting with the empty stack, suppose we push the first n natural numbers in sequence, and then perform n pop operations. Assume that Push and Pop operations take X seconds each, and Y seconds elapse between the end of one such stack operation and the start of the next operation. For $m \geq 1$, define the stack-life of m as the time elapsed from the end of Push(m) to the start of the pop operation that removes m from S. The average stack-life of an element of this stack is:*

A. $n(X + Y)$

B. $3Y + 2X$

C. $n(X + Y) - X$

D. $Y + 2X$

GATEPYQ 48 (GATE CSE 2002) *To evaluate an expression without any embedded function calls:*

A. *One stack is enough*

B. *Two stacks are needed*

C. *As many stacks as the height of the expression tree are needed*

D. *A Turing machine is needed in the general case*

GATEPYQ 49 (GATE CSE 1997) *Which of the following is essential for converting an infix expression to the postfix form efficiently?*

A. *An operator stack*

B. *An operand stack*

C. *An operand stack and an operator stack*

D. *A parse tree*

GATEPYQ 50 (GATE CSE 1995) *The postfix expression for the infix expression $A + B * (C + D)/F + D * E$ is:*

A. $AB + CD + *F/D + E*$

B. $ABCD + *F/DE * ++$

C. $A * B + CD/F * DE + +$

D. $A + *BCD/F * DE + +$

GATEPYQ 51 (GATE CSE 1994) *Which of the following permutations can be obtained in the output (in the same order) using a stack assuming that the input is the sequence 1, 2, 3, 4, 5 in that order?*

A. *3, 4, 5, 1, 2*

B. *3, 4, 5, 2, 1*

C. *1, 5, 2, 3, 4*

D. *5, 4, 3, 1, 2*

GATEPYQ 52 (GATE CSE 1991) *The following sequence of operations is performed on a stack:*

`PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP`

The sequence of values popped out is:

A. *20, 10, 20, 10, 20*

B. *20, 20, 10, 10, 20*

C. *10, 20, 20, 10, 20*

D. *20, 20, 10, 20, 10*

2.5 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
18	Stack and Stack Operations: Push, Pop & Peek	https://www.youtube.com/watch?v=WyngwwA7Flg	
19	Applications of Stack | Infix & Postfix Expressions	https://www.youtube.com/watch?v=eIY2ZTDairA	
20	Applications of Stack | Infix & Prefix Expressions	https://www.youtube.com/watch?v=6cOodLzGfi8	
21	Applications of Stack | Balanced Parentheses & Recursion	https://www.youtube.com/watch?v=V1gdGaVamVM	
22	GATE PYQs on Stacks	https://www.youtube.com/watch?v=4OfuDg2jKzk	
23	Practice Problems on Stacks	https://www.youtube.com/watch?v=EWJO_MKidhU	

Chapter 3

Queues

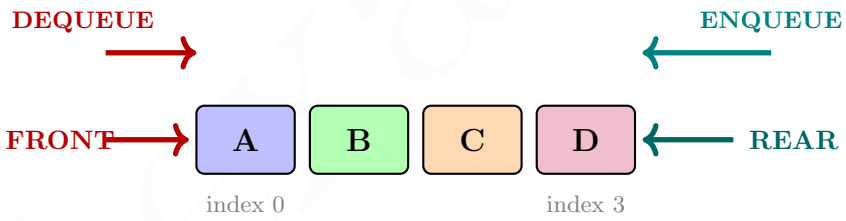
3.1 Core Operations

Queue and the FIFO Principle

A queue is a linear data structure where elements are added at one end and removed from the other end. The rule followed is **First In, First Out (FIFO)**, which means the element that entered the queue first is the one that leaves first. Think of people standing in a line at a ticket counter. The person who joined the line first gets the ticket first. New people join at the back of the line. Nobody cuts in from the front.

Example 199: Real World Example of FIFO

Think of a printer queue in an office. Document A is sent first, then document B, then document C. The printer prints A first, then B, then C. This is exactly FIFO. The first document sent is the first one printed, no matter what.



Core Queue Operations

A queue supports four main operations. All four take constant time, $O(1)$.

- **Enqueue(x):** Add element x at the REAR of the queue.
- **Dequeue():** Remove and return the element at the FRONT of the queue.
- **Front():** Look at the element at FRONT, without removing it.
- **Rear():** Look at the element at REAR, without removing it.
- **IsEmpty():** Check if the queue has no elements.
- **IsFull():** Check if the queue has no more room (for array implementation).

Time Complexity of Queue Operations

Operation	Time Complexity	What it does
`Enqueue(x)`	$O(1)$	Add at REAR
`Dequeue()`	$O(1)$	Remove from FRONT
`Front()`	$O(1)$	Peek at FRONT
`Rear()`	$O(1)$	Peek at REAR
`IsEmpty()`	$O(1)$	Check if empty
`IsFull()`	$O(1)$	Check if full

3.1.1 Array Implementation of Queue

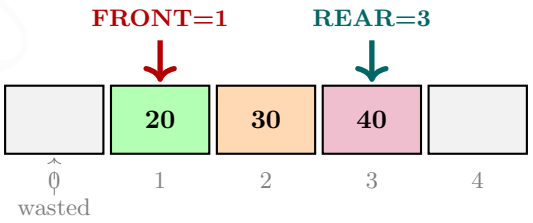
Queue Using an Array

We implement a queue using a fixed-size array. We keep two variables, **FRONT** and **REAR**. Initially, both are set to -1, which means the queue is empty. When we enqueue the first element, both FRONT and REAR become 0. After that, each enqueue increases REAR by 1 and stores the element there. Each dequeue reads the element at FRONT and increases FRONT by 1. The queue is empty when FRONT $>$ REAR, and full when REAR $=$ MAXSIZE -1.

Example 200: Step by Step Trace

Maximum size = 5. Perform: Enqueue(10), Enqueue(20), Enqueue(30), Dequeue(), Enqueue(40), Front().

- Start: FRONT $= -1$, REAR $= -1$, array = [_, _, _, _, _]
- Enqueue(10): FRONT $= 0$, REAR $= 0$, array = [10, _, _, _, _]
- Enqueue(20): FRONT $= 0$, REAR $= 1$, array = [10, 20, _, _, _]
- Enqueue(30): FRONT $= 0$, REAR $= 2$, array = [10, 20, 30, _, _]
- Dequeue(): returns 10, FRONT $= 1$, array = [_, 20, 30, _, _]
- Enqueue(40): FRONT $= 1$, REAR $= 3$, array = [_, 20, 30, 40, _]
- Front(): returns 20 (element at FRONT $= 1$), FRONT stays at 1



Problem with Linear Array Queue

In a simple array implementation, even after dequeue, the slots at the beginning are empty but we never reuse them, because FRONT keeps moving right. Eventually, REAR reaches the last index, and even if there are empty slots at the beginning, we report the queue as full. This wastes memory. The solution is the **Circular Queue**.

Problem: Wasted Slots in Linear Queue



Algorithm: Enqueue(Q, x)

1: **if** $Q.REAR = Q.MAXSIZE - 1$ **then**
2: **error** "Queue Overflow"
3: **else if** $Q.FRONT = -1$ and $Q.REAR = -1$ **then**
4: $Q.FRONT \leftarrow 0$
5: $Q.REAR \leftarrow 0$
6: $Q.arr[Q.REAR] \leftarrow x$
7: **else**
8: $Q.REAR \leftarrow Q.REAR + 1$
9: $Q.arr[Q.REAR] \leftarrow x$
10: **end if**

Algorithm: Dequeue(Q)

1: **if** $Q.FRONT = -1$ **then**
2: **error** "Queue Underflow"
3: **else**
4: $x \leftarrow Q.arr[Q.FRONT]$
5: **if** $Q.FRONT = Q.REAR$ **then**
6: $Q.FRONT \leftarrow -1$
7: $Q.REAR \leftarrow -1$
8: **else**
9: $Q.FRONT \leftarrow Q.FRONT + 1$
10: **end if**
11: **return** x
12: **end if**

Example 201: Pseudocode: Linear Queue

Initialize(Q, maxsize):

1. Q.arr = array of size maxsize
2. Q.FRONT = -1
3. Q.REAR = -1
4. Q.MAXSIZE = maxsize

Enqueue(Q, x):

1. if Q.REAR == Q.MAXSIZE - 1:
2. print "Queue Overflow" and return
3. if Q.FRONT == -1 and Q.REAR == -1:
4. Q.FRONT = 0, Q.REAR = 0
5. else:
6. Q.REAR = Q.REAR + 1
7. Q.arr[Q.REAR] = x

Dequeue(Q):

1. if Q.FRONT == -1: print "Queue Underflow" and return -1
2. x = Q.arr[Q.FRONT]
3. if Q.FRONT == Q.REAR: Q.FRONT = -1, Q.REAR = -1
4. else: Q.FRONT = Q.FRONT + 1
5. return x

Python implementation: queue-linear.py

```
class Queue:
    def __init__(self, maxsize):
        self.arr     = [None] * maxsize
        self.FRONT   = -1
        self.REAR    = -1
        self.MAXSIZE = maxsize

    def is_empty(self):
        return self.FRONT == -1

    def is_full(self):
        return self.REAR == self.MAXSIZE - 1

    def enqueue(self, x):
        if self.is_full():
            print("Queue Overflow")
            return
        if self.FRONT == -1 and self.REAR == -1:
            self.FRONT = 0
            self.REAR  = 0
        else:
            self.REAR += 1
        self.arr[self.REAR] = x

    def dequeue(self):
        if self.is_empty():
            print("Queue Underflow")
            return -1
        x = self.arr[self.FRONT]
        if self.FRONT == self.REAR:  # last element
            self.FRONT = -1
            self.REAR  = -1
        else:
            self.FRONT += 1
        return x

    def front(self):
        if self.is_empty():
            print("Queue is Empty")
            return -1
        return self.arr[self.FRONT]

    def rear(self):
        if self.is_empty():
            print("Queue is Empty")
            return -1
        return self.arr[self.REAR]

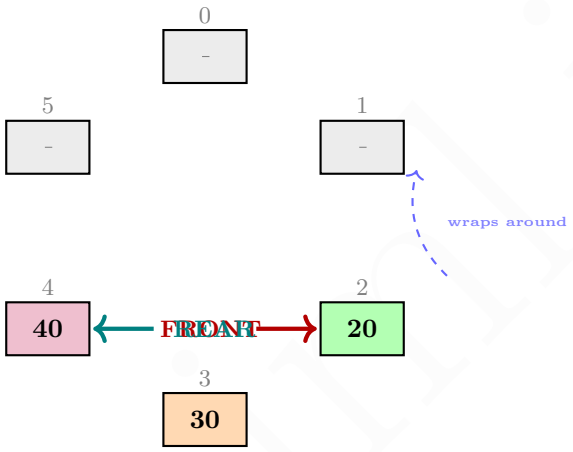

# --- Driver Code ---
q = Queue(5)
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
```

```
print("FRONT:", q.front())    # Output: 10
print("Dequeue:", q.dequeue())  # Output: 10
print("FRONT:", q.front())    # Output: 20
```

3.1.2 Circular Queue

Circular Queue

A circular queue fixes the problem of wasted slots. Instead of letting FRONT and REAR just move forward, we connect the end of the array back to the beginning, like a circle. When REAR reaches the last index, the next enqueue wraps REAR around to index 0, if that slot is free. We use the modulo operator for this. REAR moves as $(REAR + 1)$ mod $MAXSIZE$, and FRONT moves as $(FRONT + 1)$ mod $MAXSIZE$.



Circular Queue — MAXSIZE = 6, elements at index 2, 3, 4

Example 202: How Circular Wrapping Works

Suppose MAXSIZE = 5, and after several operations REAR is at index 4 (the last slot). The next enqueue would fail in a linear queue. But in a circular queue,

$$\text{new REAR} = (4 + 1) \bmod 5 = 0$$

So the element is placed at index 0, if it is free. Similarly, when FRONT is at index 4,

$$\text{new FRONT} = (4 + 1) \bmod 5 = 0$$

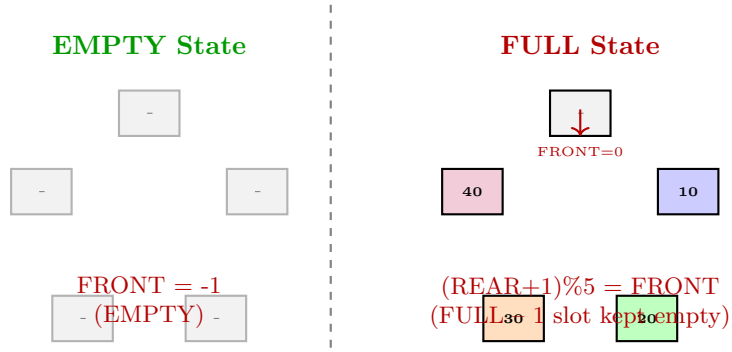
FRONT wraps back to the beginning.

Example 203: Full vs Empty Condition in Circular Queue

In a circular queue, distinguishing between full and empty needs care, because both can cause FRONT = REAR.

- **Empty:** FRONT $= -1$
- **Full:** $(REAR + 1)$ mod $MAXSIZE = FRONT$

This means we always keep one slot empty intentionally, so the full condition can be detected correctly.



Algorithm: Circular-Enqueue(Q, x)

if $(Q.REAR + 1) \bmod Q.MAXSIZE = Q.FRONT$ **then**
  error "Queue Full"
else if $Q.FRONT = -1$ **then**
  $Q.FRONT \leftarrow 0$
  $Q.REAR \leftarrow 0$
  $Q.arr[Q.REAR] \leftarrow x$
else
  $Q.REAR \leftarrow (Q.REAR + 1) \bmod Q.MAXSIZE$
  $Q.arr[Q.REAR] \leftarrow x$
end if

Algorithm: Circular-Dequeue(Q)

if $Q.FRONT = -1$ **then**
  error "Queue Empty"
else
  $x \leftarrow Q.arr[Q.FRONT]$
  if $Q.FRONT = Q.REAR$ **then**
    $Q.FRONT \leftarrow -1$
    $Q.REAR \leftarrow -1$
  else
    $Q.FRONT \leftarrow (Q.FRONT + 1) \bmod Q.MAXSIZE$
  end if
  return x
end if

Python implementation: queue-circular.py

```
class CircularQueue:
    def __init__(self, maxsize):
        self.arr     = [None] * maxsize
        self.FRONT   = -1
        self.REAR    = -1
        self.MAXSIZE = maxsize

    def is_empty(self):
        return self.FRONT == -1

    def is_full(self):
        return (self.REAR + 1) % self.MAXSIZE == self.FRONT

    def enqueue(self, x):
        if self.is_full():
            print("Queue Full")
            return
        if self.FRONT == -1:        # first element
            self.FRONT = 0
            self.REAR  = 0
        else:
            self.REAR = (self.REAR + 1) % self.MAXSIZE
        self.arr[self.REAR] = x
```

```
    def dequeue(self):
        if self.is_empty():
            print("Queue Empty")
            return -1
        x = self.arr[self.FRONT]
        if self.FRONT == self.REAR:    # last element
            self.FRONT = -1
            self.REAR  = -1
        else:
            self.FRONT = (self.FRONT + 1) % self.MAXSIZE
        return x

    def front(self):
        if self.is_empty():
            return -1
        return self.arr[self.FRONT]

    def rear(self):
        if self.is_empty():
            return -1
        return self.arr[self.REAR]

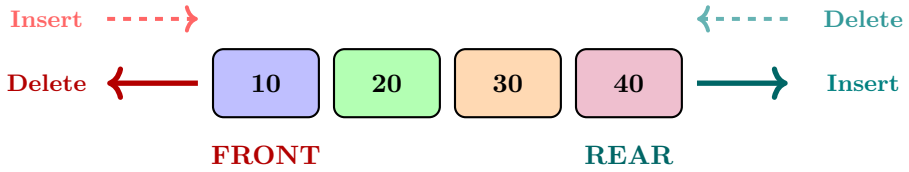
    def display(self):
        if self.is_empty():
            print("Queue is Empty")
            return
        print("Queue (FRONT to REAR):", end=" ")
        i = self.FRONT
        while True:
            print(self.arr[i], end=" ")
            if i == self.REAR:
                break
            i = (i + 1) % self.MAXSIZE
        print()


# --- Driver Code ---
cq = CircularQueue(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.enqueue(40)
cq.display()                # 10 20 30 40
print(cq.dequeue())         # 10
print(cq.dequeue())         # 20
cq.enqueue(50)              # wraps around
cq.enqueue(60)              # wraps around
cq.display()                # 30 40 50 60
```

3.1.3 Deque (Double-Ended Queue)

Deque

A Deque (pronounced "deck"), short for Double-Ended Queue, is a queue where we can add and remove elements from **both ends**, the FRONT as well as the REAR. It is a more general structure than both a stack and a simple queue. A stack only allows operations at one end. A queue allows enqueue at REAR and dequeue at FRONT only. A deque allows all four combinations, add at FRONT, add at REAR, remove from FRONT, remove from REAR.



All four operations allowed at both ends

Two Types of Deque

- **Input Restricted Deque:** Insertion is allowed only at REAR, but deletion is allowed at both FRONT and REAR. This is like a queue with extra deletion ability at REAR.
- **Output Restricted Deque:** Deletion is allowed only at FRONT, but insertion is allowed at both FRONT and REAR. This is like a queue with extra insertion ability at FRONT.

Algorithm: Deque Operations

1: **Insert-Front**(D, x): Insert x at the FRONT end
2: **Insert-Rear**(D, x): Insert x at the REAR end
3: **Delete-Front**(D): Remove and return element from FRONT
4: **Delete-Rear**(D): Remove and return element from REAR
5: **Get-Front**(D): Return element at FRONT without removing
6: **Get-Rear**(D): Return element at REAR without removing

Example 204: Pseudocode: Deque using Circular Array

Insert-Front(D, x):

1. if IsFull(D): print "Deque Full" and return
2. if IsEmpty(D): FRONT = 0, REAR = 0
3. else: FRONT = (FRONT - 1 + MAXSIZE) mod MAXSIZE
4. arr[FRONT] = x

Insert-Rear(D, x):

1. if IsFull(D): print "Deque Full" and return
2. if IsEmpty(D): FRONT = 0, REAR = 0
3. else: REAR = (REAR + 1) mod MAXSIZE
4. arr[REAR] = x

Delete-Front(D):

1. if IsEmpty(D): print "Deque Empty" and return -1
2. x = arr[FRONT]
3. if FRONT == REAR: FRONT = -1, REAR = -1
4. else: FRONT = (FRONT + 1) mod MAXSIZE
5. return x

Delete-Rear(D):

1. if IsEmpty(D): print "Deque Empty" and return -1
2. x = arr[REAR]
3. if FRONT == REAR: FRONT = -1, REAR = -1
4. else: REAR = (REAR - 1 + MAXSIZE) mod MAXSIZE
5. return x

Python implementation: deque-array.py

```
class Deque:
    def __init__(self, maxsize):
        self.arr     = [None] * maxsize
        self.FRONT   = -1
        self.REAR    = -1
        self.MAXSIZE = maxsize

    def is_empty(self):
        return self.FRONT == -1

    def is_full(self):
        return (self.REAR + 1) % self.MAXSIZE == self.FRONT

    def insert_front(self, x):
        if self.is_full():
            print("Deque Full")
            return
        if self.is_empty():
            self.FRONT = 0
            self.REAR  = 0
        else:
            # move FRONT one step back (wrap around)
            self.FRONT = (self.FRONT - 1 + self.MAXSIZE) % self.MAXSIZE
        self.arr[self.FRONT] = x

    def insert_rear(self, x):
        if self.is_full():
            print("Deque Full")
            return
        if self.is_empty():
            self.FRONT = 0
            self.REAR  = 0
        else:
            self.REAR = (self.REAR + 1) % self.MAXSIZE
        self.arr[self.REAR] = x

    def delete_front(self):
        if self.is_empty():
            print("Deque Empty")
            return -1
        x = self.arr[self.FRONT]
        if self.FRONT == self.REAR:
            self.FRONT = -1
            self.REAR  = -1
        else:
            self.FRONT = (self.FRONT + 1) % self.MAXSIZE
        return x

    def delete_rear(self):
        if self.is_empty():
            print("Deque Empty")
            return -1
        x = self.arr[self.REAR]
        if self.FRONT == self.REAR:
            self.FRONT = -1
            self.REAR  = -1
        else:
            self.REAR = (self.REAR - 1 + self.MAXSIZE) % self.MAXSIZE
        return x

    def get_front(self):
        return -1 if self.is_empty() else self.arr[self.FRONT]

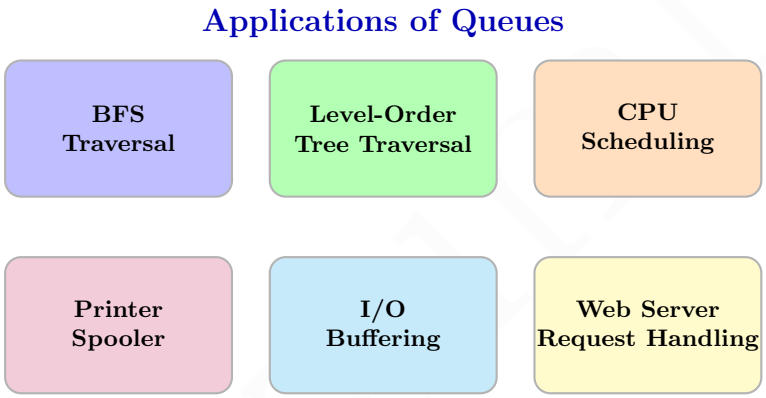
    def get_rear(self):
        return -1 if self.is_empty() else self.arr[self.REAR]
```

```
# --- Driver Code ---
d = Deque(6)
d.insert_rear(10)
d.insert_rear(20)
d.insert_rear(30)
d.insert_front(5)        # insert at FRONT
print("FRONT:", d.get_front())    # Output: 5
print("REAR:", d.get_rear())      # Output: 30
print("Delete REAR:", d.delete_rear())    # Output: 30
print("Delete FRONT:", d.delete_front()) # Output: 5
print("FRONT:", d.get_front())            # Output: 10
```

3.2 Applications of Queues

Where Queues are Used

Queues appear naturally in any situation where items are processed in the order they arrived. The FIFO property makes queues the right tool for scheduling, traversal, and buffering tasks.



Application: Breadth-First Search (BFS)

BFS uses a queue to explore all nodes level by level in a graph. Starting from a source node, we enqueue it, then repeatedly dequeue a node, visit it, and enqueue all its unvisited neighbours. The FIFO property ensures we explore nodes in the order they were discovered, which gives level-by-level traversal.

Application: Level-Order Traversal of a Tree

In a binary tree, level-order traversal visits all nodes level by level, from left to right. We enqueue the root, then repeatedly dequeue a node, print it, and enqueue its left and right children. The queue ensures that all nodes at level k are processed before any node at level $k + 1$.

Application: CPU Scheduling

Operating systems use queues to manage processes waiting for CPU time. Processes are added to the ready queue in arrival order, and the CPU picks the next process from the FRONT of the queue, following FIFO scheduling. This ensures fairness and prevents starvation of processes.

Application: Printer Spooler

When multiple users send print jobs to the same printer, each job is added to a print queue. The printer processes jobs in the order they were received. This is a classic real-world queue, where FIFO ensures fairness.

Application: I/O Buffering and Web Server Request Handling

Queues are used to buffer input and output data between a fast processor and a slow device. In web servers, incoming client requests are queued and handled one at a time in order of arrival, so no request is lost even if the server is momentarily busy.

gatexam.in

3.3 Practice Problems

Problem 91 ***[MCQ]*** *Consider an initially empty queue. The following sequence of operations is performed:*

```
Enqueue(5), Enqueue(10), Dequeue(), Enqueue(15),
Enqueue(20), Dequeue(), Dequeue(), Enqueue(25)
```

What is the element at the FRONT of the queue after all operations?

(A) 15

(B) 20

(C) 25

(D) 10

Problem 92 ***[MCQ]*** *Consider the following algorithm operating on an initially empty queue Q:*

Algorithm: Process(n)

```
for i = 1 to n do
    Enqueue(Q, i)
end for
for i = 1 to n/2 do
    x ← Dequeue(Q)
    Enqueue(Q, x * 2)
end for
return Front(Q)
```

For $n = 6$, what does `Front(Q)` return?

(A) 4

(B) 8

(C) 6

(D) 2

Problem 93 ***[MSQ]*** *An initially empty queue supports `Enqueue` and `Dequeue`. Consider input sequence $1, 2, 3, 4$ enqueued in this order, with dequeues happening at any point. Which of the following output sequences (order of dequeued elements) are POSSIBLE from a standard FIFO queue?*

(A) 1, 2, 3, 4

(B) 1, 3, 2, 4

(C) 2, 1, 3, 4

(D) 4, 3, 2, 1

Problem 94 ***[MSQ]*** *Which of the following statements about a FIFO queue are TRUE?*

(A) The element that has been in the queue the longest is always at the FRONT.

(B) It is possible to access the element in the middle of a queue in $O(1)$ time using the standard operations.

(C) Enqueue and Dequeue both take $O(1)$ time in a properly implemented queue.

(D) A queue and a stack on the same input sequence always produce the same output sequence when all elements are removed one by one.

Problem 95 ***[NAT]*** *Consider the following pseudocode operating on an initially empty queue Q:*

```
for i = 1 to 10:
    Enqueue(Q, i)

count = 0
while not IsEmpty(Q):
```

```
    x = Dequeue(Q)
    if x mod 3 == 0:
        count = count + 1

return count
```

What value is returned? _____

Problem 96 ***[MCQ]*** *A circular queue is implemented using an array of size 6 (indices 0 to 5). Initially FRONT = REAR = −1. After the operations: Enqueue(A), Enqueue(B), Enqueue(C), Dequeue(), Dequeue(), Enqueue(D), Enqueue(E), Enqueue(F), Enqueue(G), what are the values of FRONT and REAR?*

(A) FRONT = 2, REAR = 0

(B) FRONT = 2, REAR = 1

(C) FRONT = 2, REAR = 5

(D) FRONT = 3, REAR = 0

Problem 97 ***[MCQ]*** *A circular queue uses an array of size N with FRONT and REAR pointers. One slot is always kept empty to distinguish full from empty. The condition for the queue being FULL is:*

(A) $FRONT = REAR$

(B) $(REAR + 1) \bmod N = FRONT$

(C) $REAR = N - 1$

(D) $(FRONT + 1) \bmod N = REAR$

Problem 98 ***[NAT]*** *A circular queue is implemented using an array of size 7 (indices 0 to 6). Currently FRONT = 4 and REAR = 2, with the one-slot-empty convention for detecting full. The number of elements currently in the queue is* _____.

Problem 99 ***[MSQ]*** *Consider a circular queue implemented with an array of size 5 (indices 0 to 4), using the one-slot-empty convention. Currently FRONT = 3 and REAR = 2. Which of the following are TRUE?*

(A) The queue is full.

(B) Attempting another Enqueue will result in overflow.

(C) The queue contains exactly 4 elements.

(D) After one Dequeue, FRONT becomes 4.

Problem 100 ***[NAT]*** *A circular queue of size $N = 8$ (indices 0 to 7) uses the one-slot-empty convention. Currently FRONT = 5 and REAR = 3. After performing 2 Enqueue and 3 Dequeue operations (all valid), the new value of FRONT is* _____.

Problem 101 ***[MSQ]*** *A queue is implemented using two stacks S_1 (for Enqueue) and S_2 (for Dequeue), with the lazy transfer approach (transfer from S_1 to S_2 only when S_2 is empty). Which of the following are TRUE?*

(A) Every Enqueue operation is $O(1)$ in the worst case.

(B) Every Dequeue operation is $O(n)$ in the worst case, where n is the total number of elements.

(C) The amortized cost per operation over n operations is $O(1)$.

(D) The total number of push and pop operations across both stacks is $O(n^2)$ for n enqueue and n dequeue operations.

Problem 102 ***[MCQ]*** *When implementing a queue using two stacks S_1 and S_2, what is the worst case number of stack operations (push + pop combined) for a single Dequeue call, if S_1 contains n elements and S_2 is empty?*

(A) $O(1)$

(B) $O(\log n)$

(C) $O(n)$

(D) $O(n^2)$

Problem 103 ***[MCQ]*** *Consider the following algorithm that operates on an array A of size n using a queue Q:*

Algorithm: Rearrange(A, n)

```
1: Initialize empty queue Q
2: for i = 0 to n − 1 do
3:     if A[i] is even then
4:         Enqueue(Q, A[i])
5:     end if
6: end for
7: for i = 0 to n − 1 do
8:     if A[i] is odd then
9:         print A[i]
10:    end if
11: end for
12: while Q is not empty do
13:    print Dequeue(Q)
14: end while
```

For the input $A = [1, 4, 3, 6, 5, 2]$, what is the output sequence?

(A) 4, 6, 2, 1, 3, 5

(B) 1, 3, 5, 4, 6, 2

(C) 1, 3, 5, 2, 6, 4

(D) 1, 3, 5, 6, 4, 2

Problem 104 ***[MSQ]*** *Consider the following pseudocode operating on queue Q which initially contains elements $[2, 5, 1, 7, 3]$ from FRONT to REAR:*

```
S = empty stack
while not IsEmpty(Q):
    x = Dequeue(Q)
    if IsEmpty(S) or Peek(S) <= x:
        Push(S, x)
    else:
        while not IsEmpty(S) and Peek(S) > x:
            Enqueue(Q_new, Pop(S))
        Push(S, x)
while not IsEmpty(S):
    Enqueue(Q_new, Pop(S))
```

Which of the following are TRUE after this algorithm finishes?

(A) The original queue Q is empty.

(B) The new queue Q_new contains elements in sorted order.

(C) The stack S is empty.

(D) This algorithm implements a sorting procedure similar to a stack-based sort.

Problem 105 ***[MCQ]*** *A queue is used to generate binary numbers from 1 to n. The algorithm starts by enqueuing "0", then repeatedly dequeues the FRONT element, prints it, and enqueues FRONT+"0" and FRONT+"1". For $n = 7$, what is the 5th binary number printed?*

(A) 001

(B) 101

(C) 011

(D) 110

Problem 106 ***[MCQ]*** *Consider the following function f defined on queues of integers, similar in style to a well-known functional definition:*

$$f(\emptyset) = 0, \qquad f(Enqueue(Q, x)) = f(Q) + x$$

If a queue Q currently contains the elements $3, -2, 5, -1, 4$ from FRONT to REAR (FRONT is dequeued first), what is $f(Q)$?

(A) 9

(B) 4

(C) 5

(D) 3

Problem 107 ***[MCQ]*** *A stack is implemented using a single queue as the only underlying data structure (queue operations are $O(1)$ each). The implementation is designed so that Pop always removes the correct (most recently pushed) element directly from the front of the queue, with all the necessary rearrangement work done at Push time instead. What are the worst-case time complexities of Push and Pop respectively?*

(A) Push $O(1)$*, Pop* $O(n)$

(B) Push $O(\log n)$*, Pop* $O(1)$

(C) Push $O(n)$*, Pop* $O(n)$

(D) Push $O(n)$*, Pop* $O(1)$

Problem 108 ***[MCQ]*** *The following algorithm checks a property of a string using a queue. What property does it check?*

Algorithm: Check(s)

```
Initialize empty queue Q
for each character c in s do
    Enqueue(Q, c)
end for
left ← 0, right ← length(s) − 1
Initialize empty list arr from Q
while left < right do
    if arr[left] ≠ arr[right] then
        return false
    end if
    left ← left + 1, right ← right − 1
end while
return true
```

(A) Whether the string contains all unique characters

(B) Whether the string is a palindrome

(C) Whether the string is sorted

(D) Whether the string has balanced brackets

Problem 109 ***[MSQ]*** *Consider a queue implemented using two stacks (S_1 for Enqueue, S_2 for Dequeue with lazy transfer). Suppose we perform n Enqueue operations followed by n Dequeue operations. Which of the following correctly describe the total number of push and pop operations across both stacks?*

(A) Each element is pushed exactly twice in total: once onto S_1 and once onto S_2.

(B) Each element is popped exactly twice in total: once from S_1 and once from S_2.

(C) Total push and pop operations across both stacks is $4n$.

(D) Total push and pop operations across both stacks is $O(n^2)$.

Problem 110 ***[MCQ]*** *The following pseudocode is executed:*

```
MYSTERY(Q)
    if Q is empty
        return 0
    front = DEQUEUE(Q)
    rest = MYSTERY(Q)
    if front mod 2 == 0
        return front + rest
    else
        return rest

MAIN(n)
    Q = new empty queue
    for i = 1 to n
        ENQUEUE(Q, i)
    return MYSTERY(Q)
```

Assuming n is even, what is the value returned by `MAIN(n)`?

A. $\frac{n(n+1)}{2}$

B. $\frac{n(n+2)}{4}$

C. $\frac{n^2}{4}$

D. n^2

Problem 111 ***[MCQ]*** *A circular queue Q is implemented using an array of size n, using the standard convention where one slot is always kept vacant to distinguish the Full and Empty conditions. Initially `front` = `rear` = 0 (queue empty). Consider the following recursive procedure:*

```
ENQ_REC(k)
    if FULL(Q)
        return 0
    val = 2^(k-1)
    ENQUEUE(Q, val)
    return val + ENQ_REC(k + 1)
```

The call `ENQ_REC(1)` is executed on the initially empty queue, and its return value is stored in `total`. What is the value of `total`, in terms of n?

A. $2^{n-1} - 1$

B. $2^n - 1$

C. $2^{n-2} - 1$

D. $2^n - 2$

gateqa.in

3.4 GATE PYQs

GATEPYQ 53 (GATE CSE 2026) *Consider a stack S and a queue Q. Both of them are initially empty and have the capacity to store ten elements each. The elements $1, 2, 3, 4,$ and 5 arrive one by one, in that order. When an element arrives, it is assigned either to S (pushed on S) or to Q (enqueued to Q). Once all the five elements are stored, the output is generated in two steps. First, stack S is emptied by popping all elements. Then queue Q is emptied by dequeueing all elements. The output obtained by following this process is $4 \quad 3 \quad 1 \quad 2 \quad 5$.*

Given the output, the objective is to predict whether an element was assigned to S or Q.

Which of the following options is/are possible valid assignment(s) of the elements? Note: In the options, the notation x_S denotes that element x was assigned to S and y_Q denotes that element y was assigned to Q.

A. $1_S, 2_Q, 3_S, 4_S, 5_Q$

B. $1_Q, 2_Q, 3_S, 4_S, 5_Q$

C. $1_Q, 2_Q, 3_Q, 4_S, 5_S$

D. $1_S, 2_S, 3_S, 4_Q, 5_Q$

GATEPYQ 54 (GATE CSE 2023) *Consider a sequence a of elements $a_0 = 1, a_1 = 5, a_2 = 7, a_3 = 8, a_4 = 9,$ and $a_5 = 2$. The following operations are performed on a stack S and a queue Q, both of which are initially empty.*

I: push the elements of a from a_0 to a_5 in that order into S.

II: enqueue the elements of a from a_0 to a_5 in that order into Q.

III: pop an element from S.

IV: dequeue an element from Q.

V: pop an element from S.

VI: dequeue an element from Q.

VII: dequeue an element from Q and push the same element into S.

VIII: Repeat operation VII three times.

IX: pop an element from S.

X: pop an element from S.

The top element of S after executing the above operations is _____.

GATEPYQ 55 (GATE CSE 2022) *Consider the queues Q_1 containing four elements and Q_2 containing none. The only operations allowed on these two queues are Enqueue$(Q, element)$ and Dequeue(Q). The minimum number of Enqueue operations on Q_1 required to place the elements of Q_1 in Q_2 in reverse order without using any additional storage is _____.*

GATEPYQ 56 (GATE CSE 2021) *Consider the following sequence of operations on an empty stack.*

```
push(54); push(52); pop(); push(55); push(62); s = pop();
```

Consider the following sequence of operations on an empty queue.

```
enqueue(21); enqueue(24); dequeue(); enqueue(28); enqueue(32); q = dequeue();
```

The value of $s + q$ is _____.

GATEPYQ 57 (GATE CSE 2016) *Let Q denote a queue containing sixteen numbers and S be an empty stack. Head(Q) returns the element at the head of the queue Q without removing it from Q. Similarly Top(S) returns the element at the top of S without removing it from S. Consider the algorithm given below:*

```
while Q is not Empty do
    if S is Empty OR Top(S) ≤ Head(Q) then
        x := Dequeue(Q);
        Push(S, x);
    else
        x := Pop(S);
        Enqueue(Q, x);
    end if
```

end while

The maximum possible number of iterations of the while loop in the algorithm is _____.

GATEPYQ 58 (GATE CSE 2016) *A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)?*

A. *Both operations can be performed in $O(1)$ time*

B. *At most one operation can be performed in $O(1)$ time but the worst case time for the other operation will be $\Omega(n)$*

C. *The worst case time complexity for both operations will be $\Omega(n)$*

D. *Worst case time complexity for both operations will be $\Omega(\log n)$*

GATEPYQ 59 (GATE CSE 2014) *Suppose a stack implementation supports an instruction `REVERSE`, which reverses the order of elements on the stack, in addition to the `PUSH` and `POP` instructions. Which one of the following statements is TRUE with respect to this modified stack?*

A. *A queue cannot be implemented using this stack.*

B. *A queue can be implemented where `ENQUEUE` takes a single instruction and `DEQUEUE` takes a sequence of two instructions.*

C. *A queue can be implemented where `ENQUEUE` takes a sequence of three instructions and `DEQUEUE` takes a single instruction.*

D. *A queue can be implemented where both `ENQUEUE` and `DEQUEUE` take a single instruction each.*

GATEPYQ 60 (GATE CSE 2013) *Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.*

```
MultiDequeue(Q){
   m = k
   while (Q is not empty) and (m > 0) {
      Dequeue(Q)
      m = m - 1
   }
}
```

What is the worst case time complexity of a sequence of n queue operations on an initially empty queue?

A. $\Theta(n)$

B. $\Theta(n+k)$

C. $\Theta(nk)$

D. $\Theta(n^2)$

GATEPYQ 61 (GATE CSE 2012) *Suppose a circular queue of capacity $(n-1)$ elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, $REAR = FRONT = 0$. The conditions to detect queue full and queue empty are:*

A. *full:* $(REAR+1) \bmod n == FRONT$
empty: $REAR == FRONT$

B. *full:* $(REAR+1) \bmod n == FRONT$
empty: $(FRONT+1) \bmod n == REAR$

C. *full:* $REAR == FRONT$
empty: $(REAR+1) \bmod n == FRONT$

D. *full:* $(FRONT+1) \bmod n == REAR$
empty: $REAR == FRONT$

GATEPYQ 62 (GATE IT 2007) *Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are:*

i. *isEmpty(Q) returns true if the queue is empty, false otherwise.*

ii. delete(Q) deletes the element at the front of the queue and returns its value.

iii. insert(Q, i) inserts the integer i at the rear of the queue.

Consider the following function:

```
void f (queue Q) {
   int i ;
   if (!isEmpty(Q)) {
      i = delete(Q);
      f(Q);
      insert(Q, i);
   }
}
```

What operation is performed by the above function f?

A. Leaves the queue Q unchanged

B. Reverses the order of the elements in the queue Q

C. Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order

D. Empties the queue Q

GATEPYQ 63 (GATE CSE 2006) *An implementation of a queue Q, using two stacks S_1 and S_2, is given below:*

```
void insert(Q,x){
    push (S1,x);
}
void delete(Q){
   if (stack-empty (S2)) then
         if (stack-empty (S1)) then {
               print("Q is empty");
               return;
           }
         else while (! (stack-empty)(S1)){
                x=pop(S1);
                push(S2,x);
            }
       x=pop(S2);
}
```

Let n insert and m ($\leq n$) delete operations be performed in an arbitrary order on an empty queue Q. Let x and y be the number of push and pop operations performed respectively in the processes. Which one of the following is true for all m and n?

A. $m + n \leq x \leq 2n$ and $2m \leq y \leq n + m$

B. $m + n \leq x \leq 2n$ and $2m \leq y \leq 2n$

C. $2m \leq x \leq 2n$ and $2m \leq y \leq n + m$

D. $2m \leq x \leq 2n$ and $2m \leq y \leq 2n$

GATEPYQ 64 (GATE CSE 2001) *What is the minimum number of stacks of size n required to implement a queue of size n?*

A. One

B. Two

C. Three

D. Four

3.5 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
24	Queue and Operations: Enqueue & Dequeue	https://www.youtube.com/watch?v=AgMELP5gdNg	
25	Circular Queue	https://www.youtube.com/watch?v=ifkmen-8IYs	
26	Circular Queue with One Slot Empty	https://www.youtube.com/watch?v=SstC6kapmOo	
27	GATE PYQs on Queues	https://www.youtube.com/watch?v=oXz9Ol6sVtU	
28	Practice Problems on Queues	https://www.youtube.com/watch?v=IEQK7NnKGvk	

Chapter 4

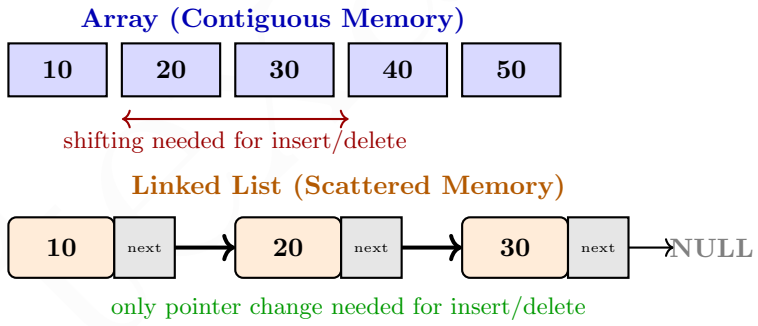
Linked Lists

Concept: What is a Linked List

A linked list is a linear data structure where elements are stored in nodes. Unlike arrays, these nodes are not stored in continuous memory locations. Instead, each node holds two things, the actual data, and a pointer that stores the address of the next node. This pointer is what links one node to the next, forming a chain.
Think of a treasure hunt game. Each clue tells you where the next clue is hidden. You start from the first clue, follow the address written on it, reach the next clue, and keep going until you reach the end. Each clue is a node. The address written on each clue is the pointer. The whole chain of clues is the linked list.

Example 205: Array vs Linked List: Key Difference

In an array, all elements sit next to each other in memory, like seats in a row. To insert in the middle, you must shift many elements. In a linked list, nodes can be anywhere in memory. To insert in the middle, you just change a few pointers. No shifting needed. This makes insertion and deletion much easier in a linked list, but random access like array$[i]$ is not possible in $O(1)$.



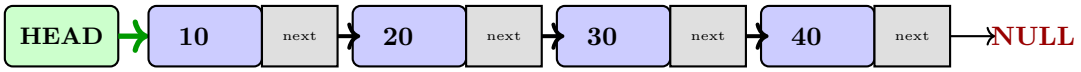
Array vs Linked List: Quick Comparison

Property	Array	Linked List
Memory layout	Contiguous	Scattered (dynamic)
Access by index	$O(1)$	$O(n)$
Insert at head	$O(n)$ (shift)	$O(1)$
Insert at tail	$O(1)$ (amortized)	$O(n)$ or $O(1)$ with tail ptr
Delete by value	$O(n)$	$O(n)$
Memory per element	Less (data only)	More (data + pointer)
Size	Fixed (static)	Dynamic (grows freely)

4.1 Singly Linked List

Concept: Singly Linked List

In a singly linked list, each node has exactly two parts: the **data field** which holds the value, and the **next pointer** which holds the address of the next node. The last node's next pointer is NULL, meaning end of list. We maintain a pointer called **head** which always points to the first node. Traversal is one-directional only, from head towards NULL. We cannot go backward.



Each node: [data | next pointer]

Python implementation: singly-node.py

```
class Node:
    def __init__(self, data):
        self.data = data    # stores the value
        self.next = None    # pointer to next node (NULL initially)

class SinglyLinkedList:
    def __init__(self):
        self.head = None    # empty list: head is NULL
```

4.1.1 Singly LL — Insertion Operations

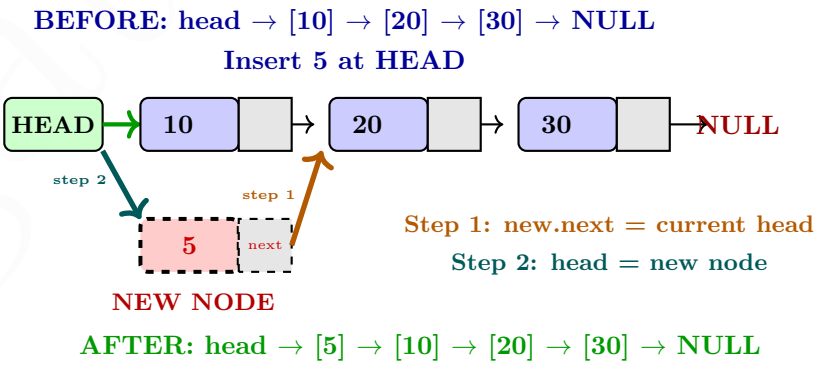
Insert at Head

Concept: Insert at Head

Insert at head means adding a new node before the current first node. The new node becomes the new head. Steps:

1. Create a new node with the given data.
2. Set the new node's next pointer to point to the current head.
3. Update head to point to the new node.

This takes $O(1)$ time because no traversal is needed. We only change two pointers.



Algorithm: SLL-Insert-At-Head(L, data)

```
1: new ← CreateNode(data)
2: new.next ← L.head        ▷ Step 1: new node points to current head
3: L.head ← new             ▷ Step 2: head now points to new node
```

Example 206: Pseudocode: Insert at Head (C-like)

SLL-Insert-At-Head(L, data):

1. new = CreateNode(data)
2. new.next = L.head
3. L.head = new

Python implementation: sll-insert-head.py

```
def insert_at_head(self, data):
    new_node      = Node(data)        # create new node
    new_node.next = self.head         # step 1: new points to old head
    self.head     = new_node          # step 2: head now points to new

# --- Driver Code ---
ll = SinglyLinkedList()
ll.insert_at_head(30)    # 30 -> NULL
ll.insert_at_head(20)    # 20 -> 30 -> NULL
ll.insert_at_head(10)    # 10 -> 20 -> 30 -> NULL
# each new element becomes the first element
```

Insert at Tail (End)

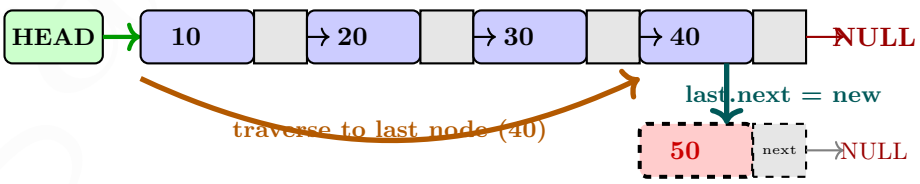
Concept: Insert at Tail

Insert at tail means adding a new node after the current last node. The new node becomes the new last node. Steps:

1. Create a new node with the given data. Its next is NULL.
2. If the list is empty, the new node becomes the head directly.
3. Otherwise, traverse from head to the last node (the one whose next is NULL).
4. Set that last node's next pointer to point to the new node.

This takes $O(n)$ time because we must traverse the entire list to find the last node. If we maintain a separate **tail pointer**, this becomes $O(1)$.

Insert 50 at TAIL: traverse to last node, then link



AFTER: [10] → [20] → [30] → [40] → [50] → NULL

Algorithm: SLL-Insert-At-Tail(L, data)

```
1: new ← CreateNode(data)
2: if L.head = NULL then
3:     L.head ← new                          ▷ empty list: new node is the head
4:     return
5: end if
6: curr ← L.head
7: while curr.next ≠ NULL do                 ▷ traverse to last node
8:     curr ← curr.next
9: end while
10: curr.next ← new                          ▷ last node now points to new node
```

Python implementation: sll-insert-tail.py

```python
def insert_at_tail(self, data):
    new_node = Node(data)
    if self.head is None:          # empty list
        self.head = new_node
        return
    curr = self.head
    while curr.next is not None:   # go to last node
        curr = curr.next
    curr.next = new_node           # link new node at end

# --- Driver Code ---
ll = SinglyLinkedList()
ll.insert_at_tail(10)    # 10 -> NULL
ll.insert_at_tail(20)    # 10 -> 20 -> NULL
ll.insert_at_tail(30)    # 10 -> 20 -> 30 -> NULL
# elements added at the end, order preserved
```

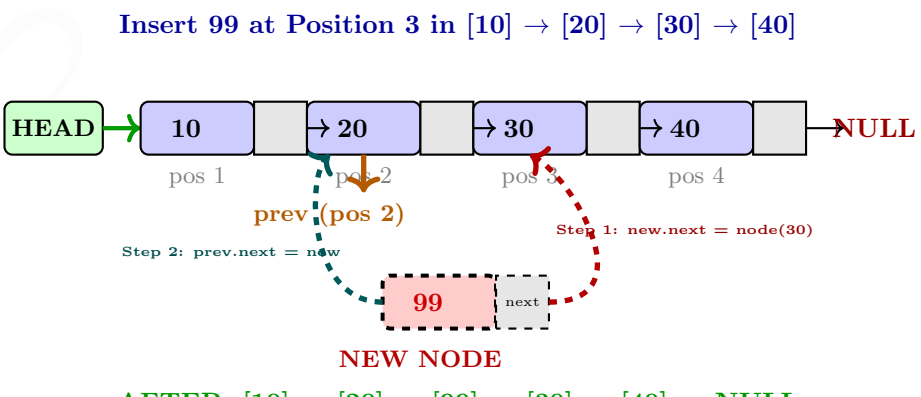
Insert at Middle (Given Position)

Concept: Insert at Given Position

To insert a new node at position k (1-indexed):

1. If $k = 1$, it is insert at head.
2. Otherwise, traverse to node at position $k - 1$ (the node just before the insertion point).
3. Set the new node's next to point to the node currently at position k.
4. Set the previous node's next to point to the new node.

Order of the two pointer assignments matters. We must first set new.next, then change prev.next. If we do it the other way, we lose the reference to the rest of the list. Time taken is $O(k)$.



Algorithm: SLL-Insert-At-Position(L, data, pos)

1: **if** $pos = 1$ **then**
2: SLL-Insert-At-Head(L, $data$)
3: **return**
4: **end if**
5: $new \leftarrow$ CreateNode($data$)
6: $curr \leftarrow L.head$, $count \leftarrow 1$
7: **while** $curr \neq$ NULL **and** $count < pos - 1$ **do**
8: $curr \leftarrow curr.next$
9: $count \leftarrow count + 1$
10: **end while**
11: **if** $curr =$ NULL **then**
12: **error** "Position out of range"
13: **end if**
14: $new.next \leftarrow curr.next$ ▷ Step 1: new points forward
15: $curr.next \leftarrow new$ ▷ Step 2: prev points to new

Python implementation: sll-insert-position.py

```
def insert_at_position(self, data, pos):
    if pos == 1:
        self.insert_at_head(data)
        return
    new_node = Node(data)
    curr  = self.head
    count = 1
    while curr is not None and count < pos - 1:
        curr  = curr.next
        count += 1
    if curr is None:
        print("Position out of range")
        return
    new_node.next = curr.next     # step 1: new points forward first!
    curr.next     = new_node      # step 2: prev points to new

# --- Driver Code ---
ll = SinglyLinkedList()
for v in [10, 20, 30, 40]:
    ll.insert_at_tail(v)
# 10 -> 20 -> 30 -> 40 -> NULL

ll.insert_at_position(99, 3)
# 10 -> 20 -> 99 -> 30 -> 40 -> NULL
```

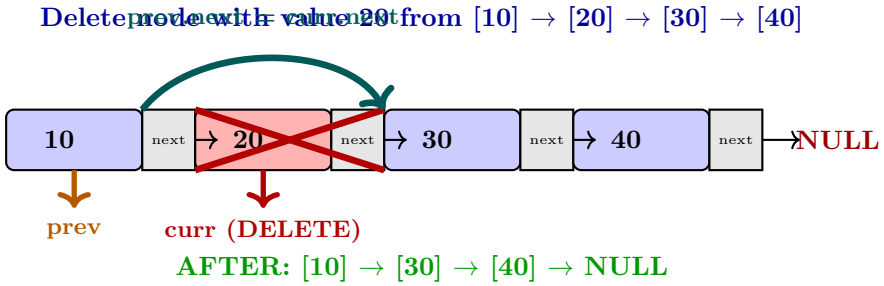
4.1.2 Singly LL — Deletion Operations

Concept: Deletion in Singly LL

To delete a node from a singly linked list, we need to change the **previous** node's next pointer to skip over the node being deleted. We cannot go backward, so we always maintain a **prev** pointer while traversing.
Three possible cases:

- **Delete head:** Move head to the second node. No prev needed.
- **Delete by value:** Traverse with prev and curr together. When found, do prev.next = curr.next.
- **Delete by position:** Same idea but count positions instead of comparing values.

All cases take $O(n)$ in worst case (except delete head which is $O(1)$).



Algorithm: SLL-Delete-By-Value(L, val)

```
1: if L.head = NULL then
2:     error "List is empty"
3:     return
4: end if
5: if L.head.data = val then                                  ▷ delete head node
6:     L.head ← L.head.next
7:     return
8: end if
9: prev ← L.head, curr ← L.head.next
10: while curr ≠ NULL do
11:     if curr.data = val then
12:         prev.next ← curr.next                             ▷ skip over curr
13:         return
14:     end if
15:     prev ← curr
16:     curr ← curr.next
17: end while
18: error "Value not found"
```

Python implementation: sll-delete.py

```python
def delete_by_value(self, val):
    if self.head is None:
        print("List is empty")
        return
    if self.head.data == val:      # delete head
        self.head = self.head.next
        return
    prev = self.head
    curr = self.head.next
    while curr is not None:
        if curr.data == val:
            prev.next = curr.next    # skip over curr
            return
        prev = curr
        curr = curr.next
    print(f"Value {val} not found")


def delete_by_position(self, pos):
    if self.head is None:
        print("List is empty")
        return
    if pos == 1:                     # delete head
        self.head = self.head.next
        return
    curr  = self.head
    count = 1
    while curr.next is not None and count < pos - 1:
        curr  = curr.next
        count += 1
    if curr.next is None:
        print("Position out of range")
        return
```

```
        curr.next = curr.next.next     # skip over node at pos

# --- Driver Code ---
ll = SinglyLinkedList()
for v in [10, 20, 30, 40]:
    ll.insert_at_tail(v)

ll.delete_by_value(20)
# 10 -> 30 -> 40 -> NULL

ll.delete_by_position(2)
# 10 -> 40 -> NULL
```

4.1.3 Singly LL — Search Operation

Concept: Search in Singly LL

To search for a value, we start at head and check each node one by one. Since there is no index or sorting guarantee, we must scan linearly. We stop when we find the value or reach NULL. Time complexity is $O(n)$ in the worst case.

Algorithm: SLL-Search(L, val)

1: $curr \leftarrow L.head$, $pos \leftarrow 1$
2: **while** $curr \neq$ NULL **do**
3: **if** $curr.data = val$ **then**
4: **return** pos ▷ found at position pos
5: **end if**
6: $curr \leftarrow curr.next$
7: $pos \leftarrow pos + 1$
8: **end while**
9: **return** -1 ▷ not found

Python implementation: sll-search.py

```
def search(self, val):
    curr = self.head
    pos  = 1
    while curr is not None:
        if curr.data == val:
            return pos          # found: return position
        curr = curr.next
        pos += 1
    return -1                   # not found

def display(self):
    curr = self.head
    while curr:
        print(curr.data, end=" -> ")
        curr = curr.next
    print("NULL")

# --- Driver Code ---
ll = SinglyLinkedList()
for v in [10, 20, 30, 40]:
    ll.insert_at_tail(v)

print(ll.search(30))    # Output: 3
print(ll.search(99))    # Output: -1
```

4.1.4 Singly LL — Reversal

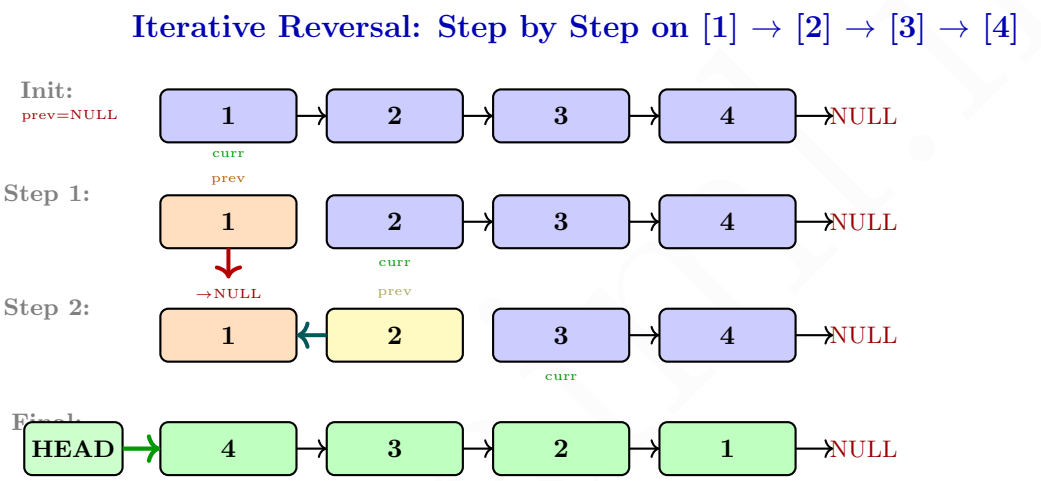
Iterative Reversal

Concept: Iterative Reversal

We use three pointers: **prev**, **curr**, and **next_node**. Initially prev = NULL, curr = head.
At each step:

1. Save curr.next into next_node (so we don't lose the rest of list).
2. Reverse the link: curr.next = prev.
3. Move prev to curr.
4. Move curr to next_node.

When curr becomes NULL, prev is the new head. Time: $O(n)$. Space: $O(1)$.



Algorithm: SLL-Reverse-Iterative(L)

```
prev ← NULL
curr ← L.head
while curr ≠ NULL do
    next ← curr.next          ▷ save next before overwriting
    curr.next ← prev          ▷ reverse the link
    prev ← curr               ▷ move prev forward
    curr ← next               ▷ move curr forward
end while
L.head ← prev                 ▷ prev is the new head
```

Recursive Reversal

Concept: Recursive Reversal

In recursive reversal, we keep calling the function on the next node until we reach the last node. The last node becomes the new head. On the way back (unwinding the recursion):

1. We have the new head from deeper recursion.
2. We do node.next.next = node (make the next node point back to current node).
3. We set node.next = NULL (break the old forward link).

The new head is passed all the way back up. Time: $O(n)$. Space: $O(n)$ recursion stack.

Algorithm: SLL-Reverse-Recursive(node)

1: **if** *node* = NULL **or** *node.next* = NULL **then**
2: **return** *node* ▷ base case: single node or empty
3: **end if**
4: *new_head* ← SLL-Reverse-Recursive(*node.next*)
5: *node.next.next* ← *node* ▷ next node points back to current
6: *node.next* ← NULL ▷ break old forward link
7: **return** *new_head*

Python implementation: sll-reverse.py

```
def reverse_iterative(self):
    prev = None
    curr = self.head
    while curr is not None:
        next_node = curr.next    # save next
        curr.next = prev         # reverse link
        prev      = curr         # advance prev
        curr      = next_node    # advance curr
    self.head = prev             # new head


def _reverse_recursive(self, node):
    if node is None or node.next is None:
        return node                        # base case
    new_head       = self._reverse_recursive(node.next)
    node.next.next = node                  # reverse the link
    node.next      = None                  # break old link
    return new_head

def reverse_recursive(self):
    self.head = self._reverse_recursive(self.head)

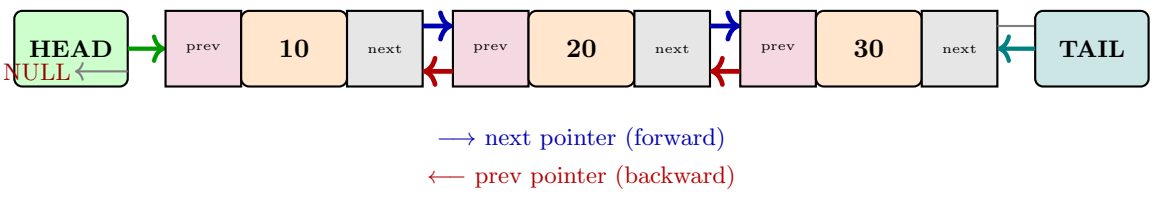

# --- Driver Code ---
ll = SinglyLinkedList()
for v in [1, 2, 3, 4, 5]:
    ll.insert_at_tail(v)

ll.display()               # 1 -> 2 -> 3 -> 4 -> 5 -> NULL
ll.reverse_iterative()
ll.display()               # 5 -> 4 -> 3 -> 2 -> 1 -> NULL
ll.reverse_recursive()
ll.display()               # 1 -> 2 -> 3 -> 4 -> 5 -> NULL
```

4.2 Doubly Linked List

Concept: Doubly Linked List

In a doubly linked list, each node has three parts: the **data** field, the **next** pointer pointing to the next node, and the **prev** pointer pointing to the previous node. We maintain both a **head** pointer (first node) and a **tail** pointer (last node). The first node's prev is NULL, and the last node's next is NULL. We can traverse in both forward and backward direction.



Python implementation: doubly-node.py

```python
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None    # forward pointer
        self.prev = None    # backward pointer

class DoublyLinkedList:
    def __init__(self):
        self.head = None    # first node
        self.tail = None    # last node
```

4.2.1 Doubly LL — Insertion Operations

Insert at Head

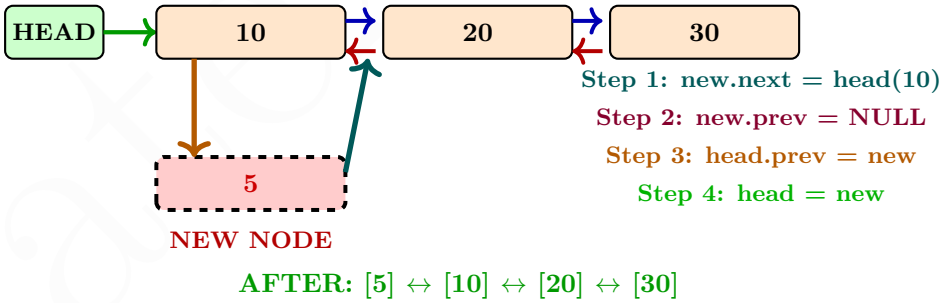
Concept: DLL Insert at Head

Steps to insert at head in a doubly linked list:

1. Create a new node.
2. Set new.next = current head (new node points forward to old head).
3. Set new.prev = NULL (new node is the first, nothing before it).
4. If the list is not empty, set head.prev = new (old head now points back to new node).
5. Update head = new.
6. If the list was empty, update tail = new as well.

Time: $O(1)$.

DLL Insert 5 at HEAD: [10] ↔ [20] ↔ [30]



Algorithm: DLL-Insert-At-Head(L, data)

```
new ← CreateNode(data)
new.next ← L.head                          ▷ Step 1
new.prev ← NULL                            ▷ Step 2
if L.head ≠ NULL then
    L.head.prev ← new                      ▷ Step 3: old head looks back to new
end if
L.head ← new                               ▷ Step 4: head = new
if L.tail = NULL then
    L.tail ← new                           ▷ list was empty: tail = new too
end if
```

Insert at Tail (End)

Concept: DLL Insert at Tail

Steps to insert at tail in a doubly linked list:

1. Create a new node.
2. Set new.prev = current tail (new node points backward to old tail).
3. Set new.next = NULL (new node is last, nothing after it).
4. If the list is not empty, set tail.next = new (old tail now points forward to new).
5. Update tail = new.
6. If the list was empty, update head = new as well.

Time: $O(1)$ because we maintain a tail pointer.

Algorithm: DLL-Insert-At-Tail(L, data)

```
1:  new ← CreateNode(data)
2:  new.prev ← L.tail                          ▷ Step 1: new points back to old tail
3:  new.next ← NULL                            ▷ Step 2
4:  if L.tail ≠ NULL then
5:      L.tail.next ← new                      ▷ Step 3: old tail looks forward to new
6:  end if
7:  L.tail ← new                               ▷ Step 4: tail = new
8:  if L.head = NULL then
9:      L.head ← new                           ▷ list was empty
10: end if
```

Insert at Middle (Given Position)

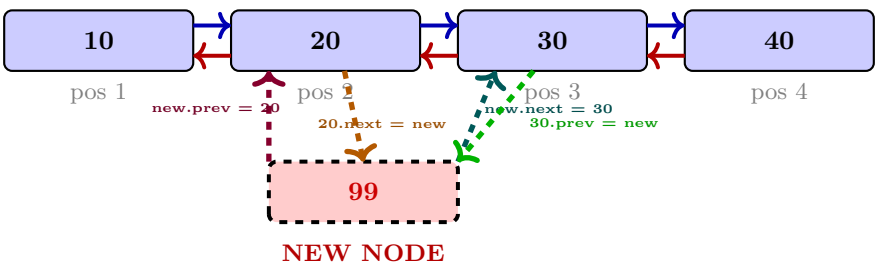
Concept: DLL Insert at Given Position

Steps to insert at position k in a doubly linked list:

1. Traverse to node at position $k-1$ (call it prev_node).
2. Let next_node = prev_node.next.
3. Set new.next = next_node and new.prev = prev_node.
4. Set prev_node.next = new.
5. If next_node is not NULL, set next_node.prev = new.

We must update four pointers total: new.next, new.prev, prev_node.next, and next_node.prev. Time: $O(k)$.

DLL Insert 99 at Position 3 in [10] ↔ [20] ↔ [30] ↔ [40]



AFTER: [10] ↔ [20] ↔ [99] ↔ [30] ↔ [40]

Algorithm: DLL-Insert-At-Position(L, data, pos)

```
1: if pos = 1 then
2:     DLL-Insert-At-Head(L, data)
3:     return
4: end if
5: new ← CreateNode(data)
6: curr ← L.head, count ← 1
7: while curr ≠ NULL and count < pos − 1 do
8:     curr ← curr.next, count ← count + 1
9: end while
10: if curr = NULL then error "Out of range"
11: end if
12: next_node ← curr.next
13: new.next ← next_node                          ▷ pointer 1
14: new.prev ← curr                               ▷ pointer 2
15: curr.next ← new                               ▷ pointer 3
16: if next_node ≠ NULL then
17:     next_node.prev ← new                      ▷ pointer 4
18: else
19:     L.tail ← new                              ▷ new is now tail
20: end if
```

Python implementation: dll-operations.py

```python
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_at_head(self, data):
        new_node = DNode(data)
        new_node.next = self.head
        new_node.prev = None
        if self.head:
            self.head.prev = new_node
        self.head = new_node
        if self.tail is None:
            self.tail = new_node

    def insert_at_tail(self, data):
        new_node = DNode(data)
        new_node.prev = self.tail
        new_node.next = None
        if self.tail:
            self.tail.next = new_node
        self.tail = new_node
        if self.head is None:
            self.head = new_node

    def insert_at_position(self, data, pos):
        if pos == 1:
            self.insert_at_head(data)
            return
        new_node = DNode(data)
        curr  = self.head
        count = 1
        while curr and count < pos - 1:
            curr  = curr.next
            count += 1
        if curr is None:
```

```
            print("Out of range")
            return
        next_node     = curr.next
        new_node.next = next_node     # pointer 1
        new_node.prev = curr          # pointer 2
        curr.next     = new_node      # pointer 3
        if next_node:
            next_node.prev = new_node  # pointer 4
        else:
            self.tail = new_node       # new is now tail

    def display_forward(self):
        curr = self.head
        print("HEAD <-> ", end="")
        while curr:
            print(curr.data, end=" <-> ")
            curr = curr.next
        print("NULL")

    def display_backward(self):
        curr = self.tail
        print("TAIL <-> ", end="")
        while curr:
            print(curr.data, end=" <-> ")
            curr = curr.prev
        print("NULL")

# --- Driver Code ---
dll = DoublyLinkedList()
dll.insert_at_tail(10)
dll.insert_at_tail(20)
dll.insert_at_tail(30)
dll.insert_at_tail(40)
dll.display_forward()      # 10 <-> 20 <-> 30 <-> 40 <-> NULL
dll.insert_at_head(5)
dll.display_forward()      # 5 <-> 10 <-> 20 <-> 30 <-> 40 <-> NULL
dll.insert_at_position(99, 3)
dll.display_forward()      # 5 <-> 10 <-> 99 <-> 20 <-> 30 <-> 40 <-> NULL
dll.display_backward()     # 40 <-> 30 <-> 20 <-> 99 <-> 10 <-> 5 <-> NULL
```

4.2.2 Doubly LL — Deletion Operations

Concept: DLL Deletion

Deleting a node in a doubly linked list is easier than in a singly linked list because we can reach the previous node directly via the **prev** pointer. No separate prev tracking variable is needed during traversal. Once we find the node to delete (call it **del**), we always do the same four pointer fixups:

1. If del.prev $\neq$ NULL: set del.prev.next = del.next (skip del going forward).
2. If del.prev = NULL: del is the head, so update head = del.next.
3. If del.next $\neq$ NULL: set del.next.prev = del.prev (skip del going backward).
4. If del.next = NULL: del is the tail, so update tail = del.prev.

Finding the node takes $O(n)$. The actual pointer fixup always takes $O(1)$.

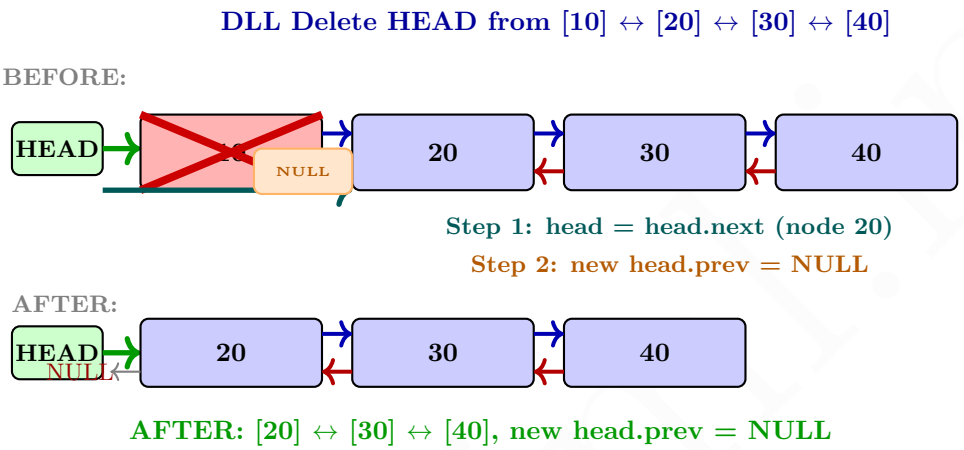
Delete at Head

Concept: DLL Delete at Head

Deleting the head is the simplest case because we already know which node to delete. Steps:

1. If the list is empty, report error.
2. If only one node exists: set head = NULL and tail = NULL.
3. Otherwise: move head to head.next, then set the new head's prev pointer to NULL.

Time: $O(1)$.



Algorithm: DLL-Delete-At-Head(L)

```
1: if L.head = NULL then
2:     error "List is empty"
3:     return
4: end if
5: if L.head = L.tail then                          ▷ only one node
6:     L.head ← NULL
7:     L.tail ← NULL
8:     return
9: end if
10: L.head ← L.head.next                            ▷ move head forward
11: L.head.prev ← NULL                              ▷ new head has no previous
```

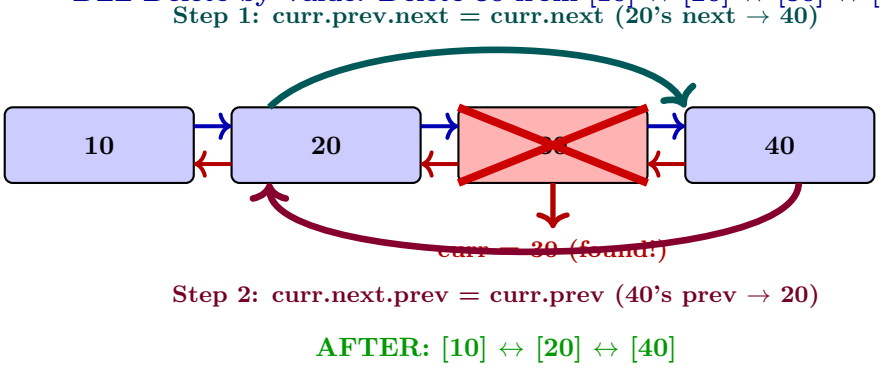
Delete by Value

Concept: DLL Delete by Value

Traverse from head using curr until we find the node whose data equals the target value. Once found, apply the four pointer fixups. Because of the prev pointer available in each node, we do not need a separate prev variable during traversal.

1. Traverse: curr = head, move curr = curr.next until curr.data = val or curr = NULL.
2. If not found: report error.
3. Fix forward link: if curr.prev $\neq$ NULL then curr.prev.next = curr.next, else head = curr.next.
4. Fix backward link: if curr.next $\neq$ NULL then curr.next.prev = curr.prev, else tail = curr.prev.

Time: $O(n)$ for traversal, $O(1)$ for deletion.



Algorithm: DLL-Delete-By-Value(L, val)

```
1:  curr ← L.head
2:  while curr ≠ NULL do
3:      if curr.data = val then
4:          if curr.prev ≠ NULL then
5:              curr.prev.next ← curr.next                ▷ fix forward link
6:          else
7:              L.head ← curr.next                        ▷ deleting head
8:          end if
9:          if curr.next ≠ NULL then
10:             curr.next.prev ← curr.prev                ▷ fix backward link
11:         else
12:             L.tail ← curr.prev                        ▷ deleting tail
13:         end if
14:         return
15:     end if
16:     curr ← curr.next
17: end while
18: error "Value not found"
```

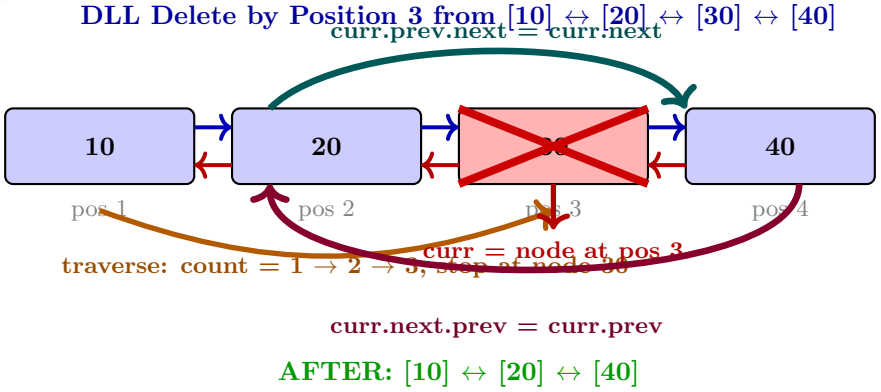
Delete by Position

Concept: DLL Delete by Position

Traverse to the node at position k (1-indexed). Once we reach that node, apply the same four pointer fixups as delete by value. Position 1 means delete head.

1. If pos = 1: call Delete-At-Head.
2. Otherwise: traverse curr to position k by counting steps.
3. If we run out of nodes before reaching k: position is out of range.
4. Apply the four pointer fixups on curr.

Time: $O(k)$ for traversal, $O(1)$ for deletion.



Algorithm: DLL-Delete-By-Position(L, pos)

1: **if** $L.head = \text{NULL}$ **then**
2: **error** "List is empty"
3: **return**
4: **end if**
5: **if** $pos = 1$ **then**
6: DLL-Delete-At-Head(L)
7: **return**
8: **end if**
9: $curr \leftarrow L.head$, $count \leftarrow 1$
10: **while** $curr \neq \text{NULL}$ **and** $count < pos$ **do**
11: $curr \leftarrow curr.next$
12: $count \leftarrow count + 1$
13: **end while**
14: **if** $curr = \text{NULL}$ **then**
15: **error** "Position out of range"
16: **return**
17: **end if**
18: **if** $curr.prev \neq \text{NULL}$ **then**
19: $curr.prev.next \leftarrow curr.next$ ▷ fix forward link
20: **else**
21: $L.head \leftarrow curr.next$ ▷ was head
22: **end if**
23: **if** $curr.next \neq \text{NULL}$ **then**
24: $curr.next.prev \leftarrow curr.prev$ ▷ fix backward link
25: **else**
26: $L.tail \leftarrow curr.prev$ ▷ was tail
27: **end if**

Python implementation: dll-delete.py

```
def delete_at_head(self):
    if self.head is None:
        print("List is empty")
        return
    if self.head == self.tail:      # only one node
        self.head = None
        self.tail = None
        return
    self.head      = self.head.next
    self.head.prev = None           # new head has no predecessor


def delete_by_value(self, val):
    curr = self.head
    while curr:
        if curr.data == val:
            if curr.prev:
                curr.prev.next = curr.next    # fix forward link
            else:
                self.head = curr.next         # deleting head
            if curr.next:
                curr.next.prev = curr.prev    # fix backward link
            else:
                self.tail = curr.prev         # deleting tail
            return
        curr = curr.next
    print(f"{val} not found")


def delete_by_position(self, pos):
    if self.head is None:
        print("List is empty")
        return
    if pos == 1:
        self.delete_at_head()
```

```
            return
        curr  = self.head
        count = 1
        while curr and count < pos:
            curr  = curr.next
            count += 1
        if curr is None:
            print("Position out of range")
            return
        if curr.prev:
            curr.prev.next = curr.next    # fix forward link
        else:
            self.head = curr.next         # was head
        if curr.next:
            curr.next.prev = curr.prev    # fix backward link
        else:
            self.tail = curr.prev         # was tail


# --- Driver Code ---
dll = DoublyLinkedList()
for v in [10, 20, 30, 40, 50]:
    dll.insert_at_tail(v)
dll.display_forward()        # 10 <-> 20 <-> 30 <-> 40 <-> 50 <-> NULL

dll.delete_at_head()
dll.display_forward()        # 20 <-> 30 <-> 40 <-> 50 <-> NULL

dll.delete_by_value(40)
dll.display_forward()        # 20 <-> 30 <-> 50 <-> NULL

dll.delete_by_position(2)
dll.display_forward()        # 20 <-> 50 <-> NULL
```

4.2.3 Doubly LL — Search Operation

Concept: DLL Search

Search in a doubly linked list is the same as in a singly linked list from the forward direction: scan from head to tail. However, with DLL we also have the option to search from tail to head using prev pointers. Both take $O(n)$ time. Searching from both ends simultaneously can save time in some cases.

Python implementation: dll-delete-search.py

```
def delete_by_value(self, val):
    curr = self.head
    while curr:
        if curr.data == val:
            if curr.prev:
                curr.prev.next = curr.next     # step 1
            else:
                self.head = curr.next           # deleting head
            if curr.next:
                curr.next.prev = curr.prev     # step 2
            else:
                self.tail = curr.prev           # deleting tail
            return
        curr = curr.next
    print(f"{val} not found")


def search(self, val):
    curr = self.head
    pos  = 1
    while curr:
```

```
        if curr.data == val:
            return pos
        curr = curr.next
        pos += 1
    return -1


# --- Driver Code ---
dll = DoublyLinkedList()
for v in [10, 20, 30, 40]:
    dll.insert_at_tail(v)

print("Search 30:", dll.search(30))    # 3
print("Search 99:", dll.search(99))    # -1

dll.delete_by_value(20)
dll.display_forward()    # 10 <-> 30 <-> 40 <-> NULL

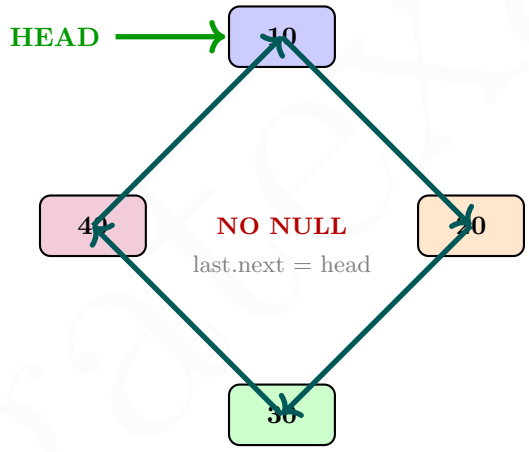
dll.delete_by_value(10)    # delete head
dll.display_forward()    # 30 <-> 40 <-> NULL

dll.delete_by_value(40)    # delete tail
dll.display_forward()    # 30 <-> NULL
```

4.3 Circular Linked List

Concept: Circular Linked List

In a circular linked list, the last node's next pointer does not point to NULL. Instead, it points back to the first node (head), forming a circle. There is no natural end. We can keep traversing forever by following next pointers. We maintain a **head** pointer. To detect when we have gone around once, we check if we have come back to head. Circular lists are used in round-robin scheduling and buffer implementations.



Key Difference from SLL

SLL: last.next = NULL

CLL: last.next = head

No NULL termination

Traverse: stop when you reach head again

Python implementation: circular-node.py

```
class CNode:
    def __init__(self, data):
        self.data = data
        self.next = None    # will point to head when in circular list

class CircularLinkedList:
    def __init__(self):
        self.head = None    # empty list

    def display(self):
        if self.head is None:
            print("Empty")
            return
        curr = self.head
```

```
        print(curr.data, end=" -> ")
        curr = curr.next
        while curr != self.head:    # stop when we come back to head
            print(curr.data, end=" -> ")
            curr = curr.next
        print("(back to head)")
```

4.3.1 Circular LL — Insertion Operations

Insert at Head

Concept: CLL Insert at Head

Inserting at head in a circular list requires us to also update the **last node's next** pointer to point to the new head. Steps:

1. Create a new node.
2. If the list is empty: new.next = new (points to itself), head = new. Done.
3. Otherwise: Traverse to the last node (the one whose next == head).
4. Set new.next = head (new node points to old head).
5. Set last.next = new (last node now points to new head).
6. Update head = new.

Time: $O(n)$ because we must traverse to find the last node. If we maintain a **tail pointer**, this becomes $O(1)$.

Insert at Tail (End)

Concept: CLL Insert at Tail

Inserting at tail in a circular list means inserting just before the head (since it is a circle). Steps:

1. Create a new node.
2. If the list is empty: new.next = new, head = new. Done.
3. Otherwise: Traverse to the current last node (whose next == head).
4. Set new.next = head (new node wraps around to head).
5. Set last.next = new (old last now points to new last).

Head does not change. Time: $O(n)$ without tail pointer.

Insert at Middle (Given Position)

Concept: CLL Insert at Given Position

Inserting at position k is similar to singly linked list, but we must be careful about the circular wrap. Steps:

1. Traverse to node at position $k-1$.
2. Set new.next = curr.next.
3. Set curr.next = new.

For inserting at position 1 (head), use Insert at Head. This takes $O(k)$ time.

Algorithm: CLL-Insert-At-Head(L, data)

```
1:  new ← CreateNode(data)
2:  if L.head = NULL then
3:      new.next ← new                                  ▷ points to itself
4:      L.head ← new
5:      return
6:  end if
7:  last ← L.head
8:  while last.next ≠ L.head do                         ▷ find last node
9:      last ← last.next
10: end while
11: new.next ← L.head                                   ▷ step 1: new points to old head
12: last.next ← new                                     ▷ step 2: last now points to new
13: L.head ← new                                        ▷ step 3: update head
```

Algorithm: CLL-Insert-At-Tail(L, data)

```
1:  new ← CreateNode(data)
2:  if L.head = NULL then
3:      new.next ← new
4:      L.head ← new
5:      return
6:  end if
7:  last ← L.head
8:  while last.next ≠ L.head do
9:      last ← last.next
10: end while
11: new.next ← L.head                                   ▷ new wraps around to head
12: last.next ← new                                     ▷ old last points to new
```

Algorithm: CLL-Insert-At-Position(L, data, pos)

```
1:  if pos = 1 then
2:      CLL-Insert-At-Head(L, data)
3:      return
4:  end if
5:  new ← CreateNode(data)
6:  curr ← L.head, count ← 1
7:  while count < pos − 1 do
8:      curr ← curr.next
9:      count ← count + 1
10:     if curr = L.head then error "Out of range"
11:     end if
12: end while
13: new.next ← curr.next                                ▷ standard: new points forward
14: curr.next ← new                                     ▷ standard: prev points to new
```

Python implementation: cll-insert.py

```python
class CircularLinkedList:
    def __init__(self):
        self.head = None

    def _find_last(self):
        """Find the last node (whose next == head)"""
        if self.head is None:
            return None
        last = self.head
        while last.next != self.head:
            last = last.next
        return last

    def insert_at_head(self, data):
```

```
        new_node = CNode(data)
        if self.head is None:
            new_node.next = new_node    # points to itself
            self.head     = new_node
            return
        last          = self._find_last()
        new_node.next = self.head       # step 1: new -> old head
        last.next     = new_node        # step 2: last -> new
        self.head     = new_node        # step 3: head = new

    def insert_at_tail(self, data):
        new_node = CNode(data)
        if self.head is None:
            new_node.next = new_node
            self.head     = new_node
            return
        last          = self._find_last()
        new_node.next = self.head       # new wraps to head
        last.next     = new_node        # old last -> new

    def insert_at_position(self, data, pos):
        if pos == 1:
            self.insert_at_head(data)
            return
        new_node = CNode(data)
        curr  = self.head
        count = 1
        while count < pos - 1:
            curr  = curr.next
            count += 1
            if curr == self.head:
                print("Out of range")
                return
        new_node.next = curr.next
        curr.next     = new_node

    def display(self):
        if self.head is None:
            print("Empty")
            return
        curr = self.head
        print(curr.data, end=" -> ")
        curr = curr.next
        while curr != self.head:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("(back to head)")


# --- Driver Code ---
cll = CircularLinkedList()
cll.insert_at_tail(10)
cll.insert_at_tail(20)
cll.insert_at_tail(30)
cll.display()               # 10 -> 20 -> 30 -> (back to head)

cll.insert_at_head(5)
cll.display()               # 5 -> 10 -> 20 -> 30 -> (back to head)

cll.insert_at_position(99, 3)
cll.display()               # 5 -> 10 -> 99 -> 20 -> 30 -> (back to head)
```

4.3.2 Circular LL — Deletion Operations

Concept: CLL Deletion — Key Rules

Deleting from a circular linked list needs special care because there is no NULL to stop traversal, and deleting the head requires updating the last node's next pointer to the new head. All deletions follow these rules:

- **Single node:** head = NULL, done.
- **Delete head:** Find last node. Set last.next = head.next. Update head = head.next.
- **Delete non-head:** Traverse with prev and curr. Set prev.next = curr.next.
- **Stopping condition:** Always check curr != head (not NULL) when traversing.

Time: $O(n)$ for all cases.

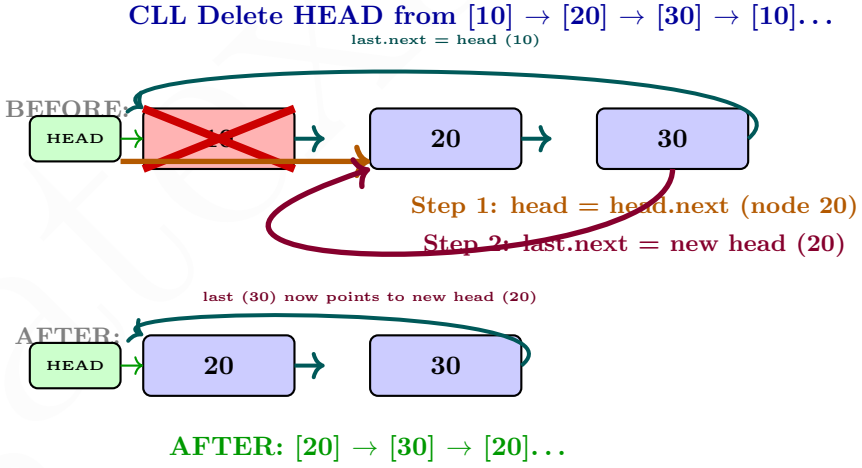
Delete at Head

Concept: CLL Delete at Head

Deleting the head is the trickiest case in CLL because the last node's next must also be updated to point to the new head. Without this fix, the circle breaks.

1. If list is empty: report error.
2. If only one node (head.next == head): set head = NULL.
3. Otherwise: traverse to the last node (whose next == head).
4. Set head = head.next (advance head).
5. Set last.next = new head (repair the circular link).

Time: $O(n)$ because we must traverse to the last node.



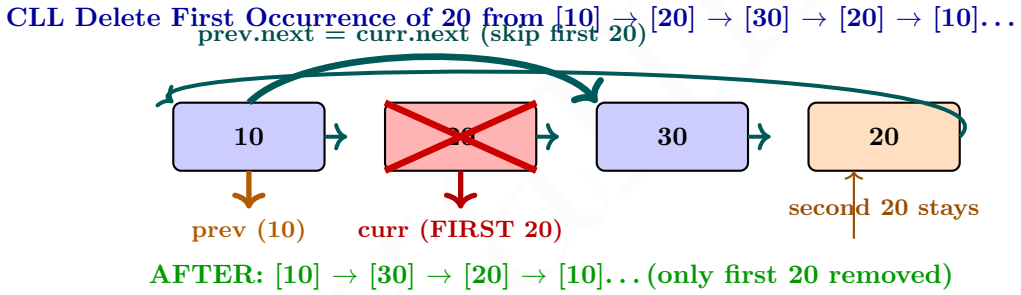
Algorithm: CLL-Delete-At-Head(L)

```
1: if L.head = NULL then
2:     error "List is empty"
3:     return
4: end if
5: if L.head.next = L.head then                          ▷ only one node
6:     L.head ← NULL
7:     return
8: end if
9: last ← L.head
10: while last.next ≠ L.head do                          ▷ find the last node
11:     last ← last.next
12: end while
13: L.head ← L.head.next                                 ▷ step 1: advance head
14: last.next ← L.head                                   ▷ step 2: repair circular link
```

Delete by Value — First Occurrence

Concept: CLL Delete by Value (First Occurrence)

Traverse with two pointers, **prev** and **curr**. Start prev = head, curr = head.next. Compare curr.data with the target. When found, set prev.next = curr.next, which removes curr from the circle. If the head itself matches, call Delete-At-Head instead. Stop when curr comes back to head (full circle done).



Algorithm: CLL-Delete-First-Occurrence(L, val)

```
1: if L.head = NULL then error "Empty"
2:     return
3: end if
4: if L.head.data = val then
5:     CLL-Delete-At-Head(L)
6:     return
7: end if
8: prev ← L.head, curr ← L.head.next
9: while curr ≠ L.head do
10:     if curr.data = val then
11:         prev.next ← curr.next                        ▷ skip over curr
12:         return
13:     end if
14:     prev ← curr
15:     curr ← curr.next
16: end while
17: error "Value not found"
```

Delete by Value — All Occurrences

Concept: CLL Delete All Occurrences

We repeatedly delete all nodes whose data equals the target value. Since nodes can appear anywhere including at the head, we need to:

1. First, delete all head occurrences in a loop (head may change multiple times).
2. Then, traverse the rest of the circle once, deleting any matching non-head node using prev.next = curr.next.
3. After deletion, do NOT advance curr (the new curr is the old curr.next, which is now at prev.next). Advance only when no deletion happened.

Time: $O(n)$ single pass after handling head.

Algorithm: CLL-Delete-All-Occurrences(L, val)

```
1:  if L.head = NULL then return
2:  end if
3:  while L.head ≠ NULL and L.head.data = val do
4:      CLL-Delete-At-Head(L)                          ▷ handle head occurrences
5:      if L.head = NULL then return
6:      end if
7:  end while
8:  prev ← L.head, curr ← L.head.next
9:  while curr ≠ L.head do
10:     if curr.data = val then
11:         prev.next ← curr.next                      ▷ remove curr
12:         curr ← curr.next                           ▷ advance curr without moving prev
13:     else
14:         prev ← curr
15:         curr ← curr.next
16:     end if
17: end while
```

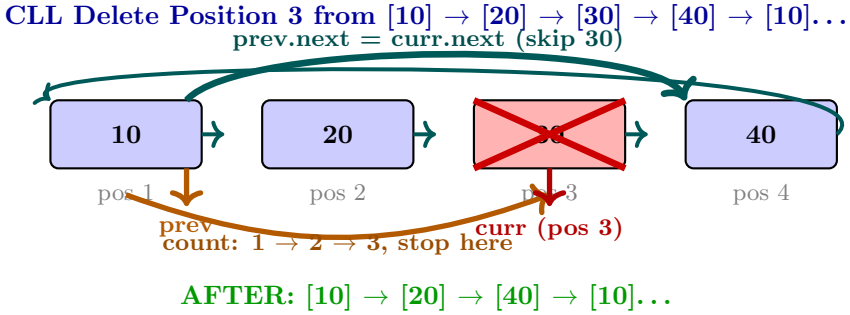
Delete by Position

Concept: CLL Delete by Position

Traverse to position k (1-indexed) by counting from head. Stop when count reaches k.

1. If pos = 1: call Delete-At-Head.
2. Otherwise: start from head, count steps until we reach position k.
3. If position exceeds the list length (we come back to head before reaching k): report error.
4. Set prev.next = curr.next to delete the node at position k.

Time: $O(k)$.



Algorithm: CLL-Delete-By-Position(L, pos)

```
1: if L.head = NULL then error "Empty"
2:     return
3: end if
4: if pos = 1 then
5:     CLL-Delete-At-Head(L)
6:     return
7: end if
8: prev ← L.head, curr ← L.head.next, count ← 2
9: while curr ≠ L.head do
10:    if count = pos then
11:        prev.next ← curr.next                          ▷ skip over curr
12:        return
13:    end if
14:    prev ← curr
15:    curr ← curr.next
16:    count ← count + 1
17: end while
18: error "Position out of range"
```

Python implementation: cll-delete-all.py

```python
def _find_last(self):
    if self.head is None:
        return None
    last = self.head
    while last.next != self.head:
        last = last.next
    return last


def delete_at_head(self):
    if self.head is None:
        print("Empty list")
        return
    if self.head.next == self.head:    # only one node
        self.head = None
        return
    last           = self._find_last()
    self.head      = self.head.next     # step 1: advance head
    last.next      = self.head          # step 2: repair circle


def delete_first_occurrence(self, val):
    if self.head is None:
        print("Empty list")
        return
    if self.head.data == val:
        self.delete_at_head()
        return
    prev = self.head
    curr = self.head.next
    while curr != self.head:
        if curr.data == val:
            prev.next = curr.next     # skip over curr
            return
        prev = curr
        curr = curr.next
    print(f"{val} not found")


def delete_all_occurrences(self, val):
    if self.head is None:
        return
    # Step 1: remove all head occurrences
    while self.head is not None and self.head.data == val:
```

```
            self.delete_at_head()
            if self.head is None:
                return

        # Step 2: traverse and remove non-head occurrences
        prev = self.head
        curr = self.head.next
        while curr != self.head:
            if curr.data == val:
                prev.next = curr.next     # skip curr, do NOT move prev
                curr      = curr.next
            else:
                prev = curr
                curr = curr.next


    def delete_by_position(self, pos):
        if self.head is None:
            print("Empty list")
            return
        if pos == 1:
            self.delete_at_head()
            return
        prev  = self.head
        curr  = self.head.next
        count = 2
        while curr != self.head:
            if count == pos:
                prev.next = curr.next     # skip over curr
                return
            prev  = curr
            curr  = curr.next
            count += 1
        print("Position out of range")


    # --- Driver Code ---
    cll = CircularLinkedList()
    for v in [10, 20, 20, 30, 20, 40]:
        cll.insert_at_tail(v)
    cll.display()
    # 10 -> 20 -> 20 -> 30 -> 20 -> 40 -> (back to head)

    cll.delete_first_occurrence(20)
    cll.display()
    # 10 -> 20 -> 30 -> 20 -> 40 -> (back to head)

    cll.delete_all_occurrences(20)
    cll.display()
    # 10 -> 30 -> 40 -> (back to head)

    cll.delete_at_head()
    cll.display()
    # 30 -> 40 -> (back to head)

    cll.delete_by_position(2)
    cll.display()
    # 30 -> (back to head)
```

4.3.3 Circular LL — Search Operation

Concept: CLL Search

Searching in a circular list is similar to singly linked list, but we must stop when we come back to the head, not when we hit NULL (there is no NULL). We traverse until we find the value or come full circle back to head.

Python implementation: cll-delete-search.py

```
def delete_by_value(self, val):
    if self.head is None:
        print("Empty list")
        return
    # Case: only one node
    if self.head.data == val and self.head.next == self.head:
        self.head = None
        return
    # Case: delete head
    if self.head.data == val:
        last = self._find_last()
        self.head      = self.head.next
        last.next      = self.head
        return
    # Case: delete non-head node
    prev = self.head
    curr = self.head.next
    while curr != self.head:
        if curr.data == val:
            prev.next = curr.next     # skip over curr
            return
        prev = curr
        curr = curr.next
    print(f"{val} not found")


def search(self, val):
    if self.head is None:
        return -1
    curr = self.head
    pos  = 1
    while True:
        if curr.data == val:
            return pos
        curr = curr.next
        pos += 1
        if curr == self.head:     # came full circle
            break
    return -1


# --- Driver Code ---
cll = CircularLinkedList()
for v in [10, 20, 30, 40, 50]:
    cll.insert_at_tail(v)
cll.display()
# 10 -> 20 -> 30 -> 40 -> 50 -> (back to head)

print("Search 30:", cll.search(30))    # 3
print("Search 99:", cll.search(99))    # -1

cll.delete_by_value(30)
cll.display()
# 10 -> 20 -> 40 -> 50 -> (back to head)

cll.delete_by_value(10)    # delete head
cll.display()
# 20 -> 40 -> 50 -> (back to head)
```

All Three Variants: Operations Summary

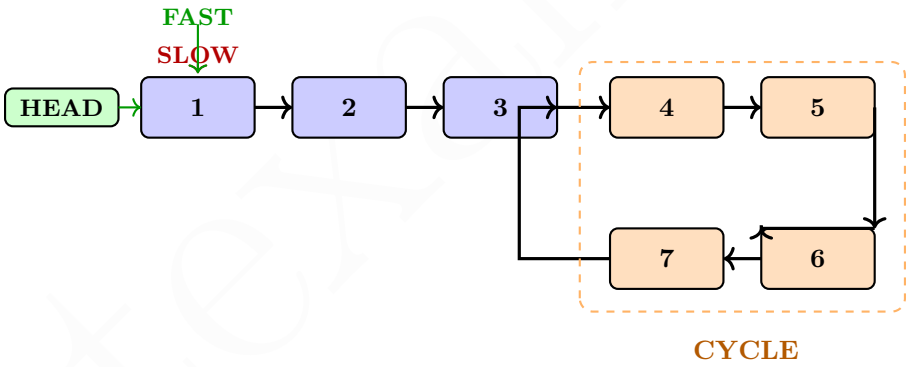
Operation	Singly LL	Doubly LL	Circular LL
Insert at Head	$O(1)$	$O(1)$	$O(n)$, or $O(1)$ with tail ptr
Insert at Tail	$O(n)$, or $O(1)$ with tail	$O(1)$	$O(n)$
Insert at Middle	$O(k)$	$O(k)$	$O(k)$
Delete by Value	$O(n)$	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$	$O(n)$
Reverse	$O(n)$	Swap head and tail	Not standard
Traversal Direction	Forward only	Both directions	Forward (loops)
NULL termination	Yes (last.next = NULL)	Yes	No (last.next = head)

4.4 Classic Problems

4.4.1 Floyd's Cycle Detection Algorithm

Floyd's Two-Pointer Algorithm

This algorithm uses two pointers, **slow** and **fast**. Slow moves one step at a time. Fast moves two steps at a time. If there is no cycle, fast will reach NULL. If there is a cycle, fast and slow will eventually meet inside the cycle, because fast is catching up to slow at a rate of one step per round. This is like two runners on a circular track, the faster runner always laps the slower one.



Algorithm: Detect-Cycle(L)

```
slow ← L.head
fast ← L.head
while fast ≠ NULL and fast.next ≠ NULL do
    slow ← slow.next                ▷ move 1 step
    fast ← fast.next.next           ▷ move 2 steps
    if slow = fast then
        return true                 ▷ cycle detected
    end if
end while
return false                        ▷ no cycle
```

4.4.2 Finding the Start of a Cycle

Finding Cycle Start

Once Floyd's algorithm detects a cycle (slow and fast meet at some point inside the cycle), we can find where the cycle starts. The mathematical proof shows that the distance from head to cycle start equals the distance from the meeting point to cycle start (going forward). So, after the meeting is detected, we reset one pointer to head, and keep the other at the meeting point. Both move one step at a time. The node where they meet again is the start of the cycle.

Idea

Let: F = distance from HEAD to cycle start, C = cycle length, a = distance from cycle start to meeting point.

When slow and fast meet: slow travelled $F + a$, fast travelled $F + a + C$.

Since fast = 2× slow: $F + a + C = 2(F + a) \implies C = F + a \implies F = C - a$.

Resetting one pointer to head and moving both 1 step at a time ensures they meet exactly at the cycle start.

Algorithm: Find-Cycle-Start(L)

```
slow ← L.head, fast ← L.head
has_cycle ← false
while fast ≠ NULL and fast.next ≠ NULL do
    slow ← slow.next
    fast ← fast.next.next
    if slow = fast then
        has_cycle ← true
        break
    end if
end while
if has_cycle = false then
    return NULL
end if
slow ← L.head                                   ▷ reset slow to head
while slow ≠ fast do
    slow ← slow.next
    fast ← fast.next
end while
return slow                                     ▷ this is the cycle start
```

Python implementation: cycle-detection.py

```python
def detect_cycle(self):
    slow = self.head
    fast = self.head
    while fast and fast.next:
        slow = slow.next            # 1 step
        fast = fast.next.next       # 2 steps
        if slow == fast:
            return True             # cycle exists
    return False


def find_cycle_start(self):
    slow = self.head
    fast = self.head
    has_cycle = False

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            has_cycle = True
```

```
            break

    if not has_cycle:
        return None

    slow = self.head            # reset slow to head
    while slow != fast:         # both move 1 step
        slow = slow.next
        fast = fast.next

    return slow                 # start of cycle


# --- Creating a cycle for testing ---
ll = LinkedList()
for val in [1, 2, 3, 4, 5, 6]:
    ll.insert_at_tail(val)

# Manually create cycle: node 6 points back to node 3
curr = ll.head
nodes = []
while curr:
    nodes.append(curr)
    curr = curr.next

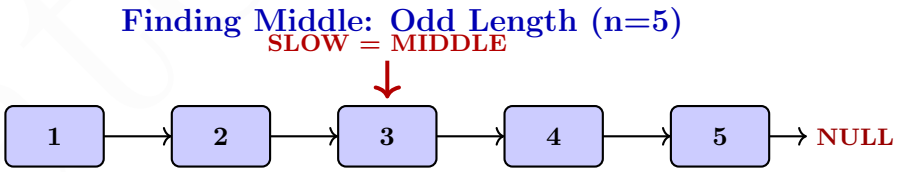
nodes[5].next = nodes[2]    # creates cycle: 6 -> 3

print("Cycle detected:", ll.detect_cycle())    # True
start = ll.find_cycle_start()
print("Cycle starts at node with value:", start.data)  # 3
```

4.4.3 Finding the Middle Node

Slow and Fast Pointer for Middle

We use two pointers again, slow and fast. Slow moves one step at a time, fast moves two steps at a time. When fast reaches the end, slow is at the middle. This is because fast covers twice the distance in the same number of steps. So when fast has covered the whole list, slow has covered half. For a list of length n, this takes $O(n)$ time with only one pass.



Step	SLOW position	FAST position
Start	node 1	node 1
1	node 2	node 3
2	node 3	node 5

Conclusion: SLOW = node 3 = MIDDLE when FAST reaches the last node

Algorithm: Find-Middle(L)

1: **if** $L.head = \text{NULL}$ **then**
2: **return** NULL
3: **end if**
4: $slow \leftarrow L.head$
5: $fast \leftarrow L.head$
6: **while** $fast.next \neq \text{NULL}$ **and** $fast.next.next \neq \text{NULL}$ **do**
7: $slow \leftarrow slow.next$
8: $fast \leftarrow fast.next.next$
9: **end while**
10: **return** $slow$ ▷ slow is at middle

Python implementation: find-middle.py

```
def find_middle(self):
    if self.head is None:
        return None
    slow = self.head
    fast = self.head
    while fast.next and fast.next.next:
        slow = slow.next          # 1 step
        fast = fast.next.next     # 2 steps
    return slow                   # slow is at middle

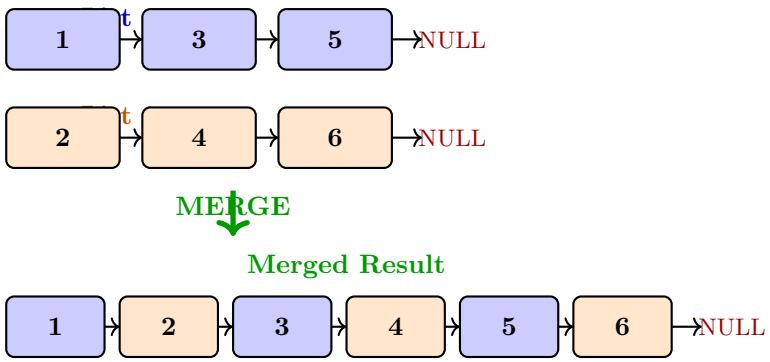

# --- Driver Code ---
ll = LinkedList()
for val in [1, 2, 3, 4, 5]:
    ll.insert_at_tail(val)
mid = ll.find_middle()
print("Middle:", mid.data)    # Output: 3

ll2 = LinkedList()
for val in [1, 2, 3, 4, 5, 6]:
    ll2.insert_at_tail(val)
mid2 = ll2.find_middle()
print("Middle:", mid2.data)  # Output: 3 (first middle for even length)
```

4.4.4 Merge Two Sorted Linked Lists

Merging Two Sorted Lists

Given two sorted linked lists, we want to merge them into a single sorted linked list. We use two pointers, one on each list. We compare the front elements of both lists, pick the smaller one, attach it to our result list, and move that pointer forward. We keep doing this until one list runs out. Then we attach the remaining list directly to the result.



Algorithm: Merge-Sorted-Lists(L1, L2)

if $L1.head = \text{NULL}$ **then return** $L2.head$
end if
if $L2.head = \text{NULL}$ **then return** $L1.head$
end if
$dummy \leftarrow \text{CreateNode}(0)$ ▷ temporary dummy head
$curr \leftarrow dummy$
$p1 \leftarrow L1.head$, $p2 \leftarrow L2.head$
while $p1 \neq \text{NULL}$ **and** $p2 \neq \text{NULL}$ **do**
    if $p1.data \leq p2.data$ **then**
        $curr.next \leftarrow p1$
        $p1 \leftarrow p1.next$
    else
        $curr.next \leftarrow p2$
        $p2 \leftarrow p2.next$
    end if
    $curr \leftarrow curr.next$
end while
if $p1 \neq \text{NULL}$ **then**
    $curr.next \leftarrow p1$
else
    $curr.next \leftarrow p2$
end if
return $dummy.next$ ▷ skip the dummy node

Python implementation: merge-sorted-lists.py

```
def merge_sorted_lists(head1, head2):
    dummy = Node(0)          # dummy node to simplify code
    curr  = dummy

    p1 = head1
    p2 = head2

    while p1 and p2:
        if p1.data <= p2.data:
            curr.next = p1   # pick from list 1
            p1 = p1.next
        else:
            curr.next = p2   # pick from list 2
            p2 = p2.next
        curr = curr.next

    if p1:                   # remaining elements of list 1
        curr.next = p1
    else:                    # remaining elements of list 2
        curr.next = p2

    return dummy.next        # actual merged head


# --- Driver Code ---
ll1 = LinkedList()
for val in [1, 3, 5, 7]:
    ll1.insert_at_tail(val)

ll2 = LinkedList()
for val in [2, 4, 6, 8]:
    ll2.insert_at_tail(val)

merged_head = merge_sorted_lists(ll1.head, ll2.head)

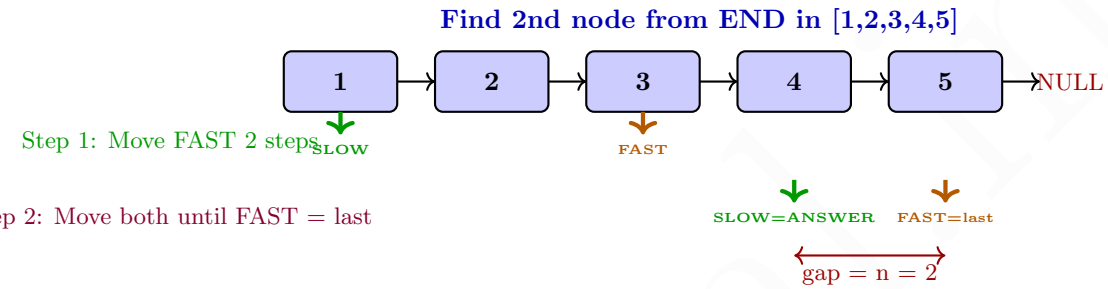
# print merged list
curr = merged_head
while curr:
    print(curr.data, end=" -> ")
```

```
    curr = curr.next
print("NULL")
# Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> NULL
```

4.4.5 Nth Node from the End

Nth Node from the End

To find the nth node from the end in one pass, we use two pointers, **fast** and **slow**. First, we move fast n steps ahead from the head. Then, we move both fast and slow one step at a time, until fast reaches the last node. At this point, slow is exactly n positions from the end. This works because we maintained a gap of exactly n nodes between fast and slow throughout.



Algorithm: Nth-From-End(L, n)

1: $fast \leftarrow L.head$, $slow \leftarrow L.head$
2: **for** $i = 1$ to n **do**
3: **if** $fast =$ NULL **then**
4: **error** "n is larger than list length"
5: **end if**
6: $fast \leftarrow fast.next$
7: **end for**
8: **while** $fast \neq$ NULL **do**
9: $slow \leftarrow slow.next$
10: $fast \leftarrow fast.next$
11: **end while**
12: **return** *slow* ▷ slow is nth from end

Python implementation: nth-from-end.py

```
def nth_from_end(self, n):
    fast = self.head
    slow = self.head

    # move fast n steps ahead
    for i in range(n):
        if fast is None:
            print("n is larger than list length")
            return None
        fast = fast.next

    # move both until fast reaches end
    while fast is not None:
        slow = slow.next
        fast = fast.next

    return slow    # slow is at nth node from end


# --- Driver Code ---
ll = LinkedList()
for val in [1, 2, 3, 4, 5]:
    ll.insert_at_tail(val)
```

```
node = ll.nth_from_end(2)
print("2nd from end:", node.data)    # Output: 4

node = ll.nth_from_end(1)
print("1st from end:", node.data)    # Output: 5

node = ll.nth_from_end(5)
print("5th from end:", node.data)    # Output: 1
```

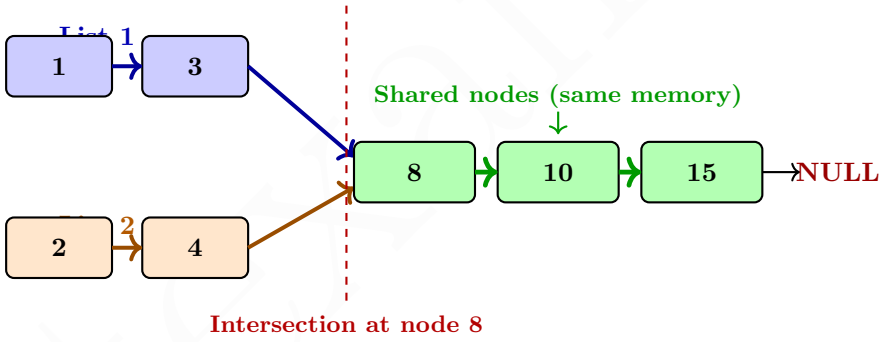
4.4.6 Intersection of Two Linked Lists

Concept: What is Intersection of Linked Lists

Two singly linked lists are said to **intersect** if they share a common node at some point. From that common node onward, both lists follow the exact same nodes all the way to the end. The shared part forms a Y-shape, not an X-shape, because once the lists merge at the intersection node, they become one, and all nodes after it are shared. Think of two rivers flowing separately and then joining together at a point. After that joining point, there is only one river, not two. Similarly, after the intersection node, both lists share the same nodes and the same tail.

Example 207: Analogy: Two Paths Meeting

Imagine two hiking trails starting from different points but eventually joining at a fork. From the fork onward, both trails follow the same path to the summit. The fork is the intersection node. Everything before the fork is separate for each trail. Everything after the fork is shared.



Concept: Important Properties of Intersection

- The intersection is defined by **same memory address**, not same value. Two nodes with value 8 at different memory locations are NOT an intersection.
- Once two lists intersect, they must have the **same tail node**. So, checking if both lists end at the same node is the fastest way to detect intersection.
- The **lengths of the two lists** from head to the intersection node can be different, which is why a naive same-speed traversal misses the intersection.

Example 208: Why Naive Traversal Misses the Intersection

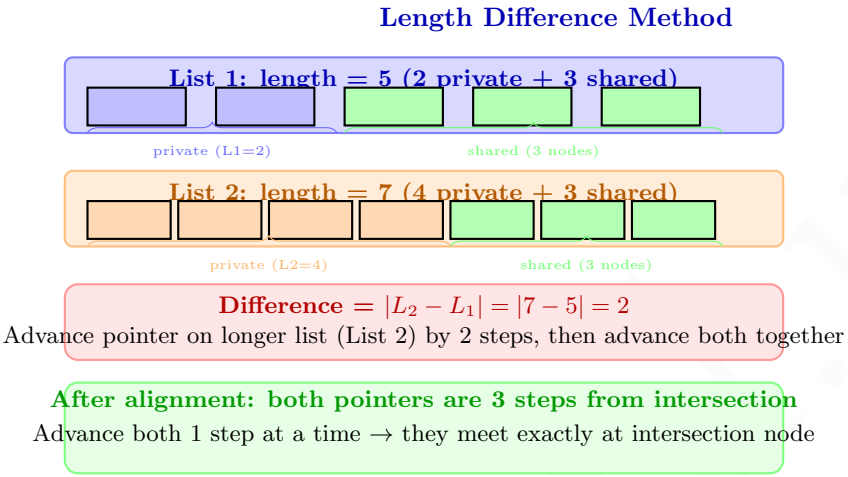
If List 1 has 5 nodes before intersection and List 2 has 3 nodes before intersection, advancing one pointer per step from each head will never land both pointers at the intersection at the same time. They are always out of sync by 2 steps. We need to compensate for this length difference.

Methods to Find Intersection

Method 1: Using Length Difference

Concept: Length Difference Method

The idea is simple. If List 1 has length L_1 and List 2 has length L_2, then the longer list's pointer should start $|L_1 - L_2|$ steps ahead. After this adjustment, both pointers are the same distance from the intersection node. We then advance both one step at a time until they meet. That meeting point is the intersection.



Algorithm: Find-Intersection-Length-Method(L1, L2)

```
Compute len1 ← Length(L1.head)
Compute len2 ← Length(L2.head)
p1 ← L1.head, p2 ← L2.head
if len1 > len2 then
    for i = 1 to len1 − len2 do
        p1 ← p1.next
    end for
else
    for i = 1 to len2 − len1 do
        p2 ← p2.next
    end for
end if
while p1 ≠ p2 do                          ▷ advance both together
    p1 ← p1.next
    p2 ← p2.next
end while
return p1                                  ▷ NULL if no intersection, else intersection node
```

Method 2: Two-Pointer Redirect Method (Floyd-like)

Concept: Two-Pointer Redirect Method

This elegant method uses two pointers p_1 and p_2, starting at the heads of List 1 and List 2. Both advance one step at a time. When a pointer reaches the end (NULL) of its list, it is redirected to the **head of the other list**. If the lists intersect, both pointers will meet at the intersection node. If they do not intersect, both will reach NULL at the same time.

Why does this work? By the time both pointers have been redirected once, each has traveled the full length of both lists combined ($L_1 + L_2$ steps total). The private portions cancel out, and both land at the intersection at the same time.

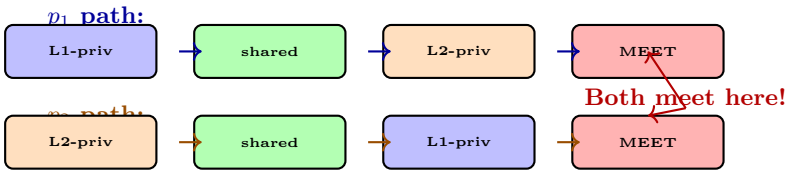
Mathematically: p_1 travels $a + c$ steps in List 1, then b steps in List 2 before intersection: total $= a + c + b$. p_2 travels $b + c$ steps in List 2, then a steps in List 1 before intersection: total $= b + c + a$. Both are equal, so they meet at the intersection.

Two-Pointer Redirect: Why it Works

p_1: List1-private(a steps) + shared(c steps) + List2-private(b steps) = $a + c + b$

p_2: List2-private(b steps) + shared(c steps) + List1-private(a steps) = $b + c + a$

$a + c + b = b + c + a \Rightarrow$ Both pointers meet at intersection after same steps



Algorithm: Find-Intersection-Two-Pointer(L1, L2)

```
p1 ← L1.head, p2 ← L2.head
while p1 ≠ p2 do
    if p1 = NULL then
        p1 ← L2.head                    ▷ redirect to other list's head
    else
        p1 ← p1.next
    end if
    if p2 = NULL then
        p2 ← L1.head                    ▷ redirect to other list's head
    else
        p2 ← p2.next
    end if
end while
return p1                               ▷ NULL means no intersection, else intersection node
```

Example 209: Pseudocode: Two-Pointer Redirect (C-like)

FindIntersection(head1, head2):

1. p1 = head1, p2 = head2
2. while p1 != p2:
3. if p1 == NULL: p1 = head2
4. else: p1 = p1.next
5. if p2 == NULL: p2 = head1
6. else: p2 = p2.next
7. return p1 (NULL means no intersection)

Python implementation: linked-list-intersection.py

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


# ============================================================
# Method 1: Length Difference
# ============================================================
def get_length(head):
    length = 0
    curr   = head
    while curr:
        length += 1
        curr = curr.next
```

```
    return length


def find_intersection_length(head1, head2):
    l1 = get_length(head1)
    l2 = get_length(head2)

    p1 = head1
    p2 = head2

    # advance longer list pointer
    if l1 > l2:
        for _ in range(l1 - l2):
            p1 = p1.next
    else:
        for _ in range(l2 - l1):
            p2 = p2.next

    # advance both together
    while p1 != p2:
        p1 = p1.next
        p2 = p2.next

    return p1   # None if no intersection


# ============================================================
# Method 2: Two-Pointer Redirect (elegant)
# ============================================================
def find_intersection_two_pointer(head1, head2):
    if head1 is None or head2 is None:
        return None

    p1 = head1
    p2 = head2

    while p1 != p2:
        p1 = head2 if p1 is None else p1.next   # redirect on reaching end
        p2 = head1 if p2 is None else p2.next   # redirect on reaching end

    return p1   # None if no intersection, else intersection node


# ============================================================
# Build intersecting lists for testing
# ============================================================
# Shared tail: 8 -> 10 -> 15 -> NULL
shared1 = Node(8)
shared2 = Node(10)
shared3 = Node(15)
shared1.next = shared2
shared2.next = shared3

# List 1: 1 -> 3 -> [8 -> 10 -> 15]
h1      = Node(1)
h1.next = Node(3)
h1.next.next = shared1

# List 2: 2 -> 4 -> 6 -> 7 -> [8 -> 10 -> 15]
h2            = Node(2)
h2.next       = Node(4)
h2.next.next  = Node(6)
h2.next.next.next = Node(7)
h2.next.next.next.next = shared1   # same shared1 node

# Test Method 1
result1 = find_intersection_length(h1, h2)
print("Method 1 - Intersection at node with value:",
```

```
      result1.data if result1 else "None")
# Output: 8

# Test Method 2
result2 = find_intersection_two_pointer(h1, h2)
print("Method 2 - Intersection at node with value:",
      result2.data if result2 else "None")
# Output: 8

# Test with non-intersecting lists
n1      = Node(10)
n1.next = Node(20)
n2      = Node(30)
n2.next = Node(40)

result3 = find_intersection_two_pointer(n1, n2)
print("Non-intersecting:", result3)
# Output: None
```

Comparison: Two Methods to Find Intersection

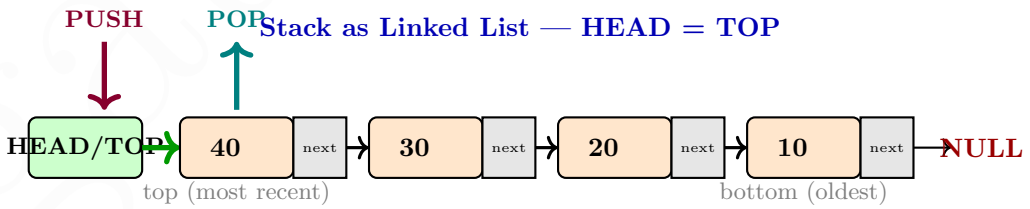
Property	Method 1: Length Difference	Method 2: Two-Pointer Redirect
Time Complexity	$O(L_1 + L_2)$	$O(L_1 + L_2)$
Space Complexity	$O(1)$	$O(1)$
Extra passes needed	3 passes (2 for length + 1 for find)	At most 2 passes combined
Code complexity	Slightly more code	Very clean and short
No intersection result	Both pointers reach NULL	Both become NULL and meet

4.5 Linked List Implementation of Stack and Queue

4.5.1 Stack Using Linked List

Stack Using Linked List

We implement a stack using a singly linked list. The head of the linked list acts as the top of the stack. Push means inserting a new node at the head, which takes $O(1)$ time. Pop means removing the head node, which also takes $O(1)$ time. This approach has no size limit, unlike array-based stacks, because nodes are created dynamically.



Algorithm: Stack-Push-LL(S, x)

1: $new\_node \leftarrow$ CreateNode(x)
2: $new\_node.next \leftarrow S.top$ ▷ new node points to old top
3: $S.top \leftarrow new\_node$ ▷ top is now new node

Algorithm: Stack-Pop-LL(S)

1: **if** $S.top = \text{NULL}$ **then**
2: **error** "Stack Underflow"
3: **end if**
4: $x \leftarrow S.top.data$
5: $S.top \leftarrow S.top.next$ ▷ move top to next node
6: **return** x

Python implementation: stack-linked-list.py

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class StackLL:
    def __init__(self):
        self.top = None     # head of linked list = top of stack

    def is_empty(self):
        return self.top is None

    def push(self, x):
        new_node      = Node(x)
        new_node.next = self.top  # new node points to old top
        self.top      = new_node  # top is now new node

    def pop(self):
        if self.is_empty():
            print("Stack Underflow")
            return -1
        x        = self.top.data
        self.top = self.top.next  # move top to next
        return x

    def peek(self):
        if self.is_empty():
            return -1
        return self.top.data

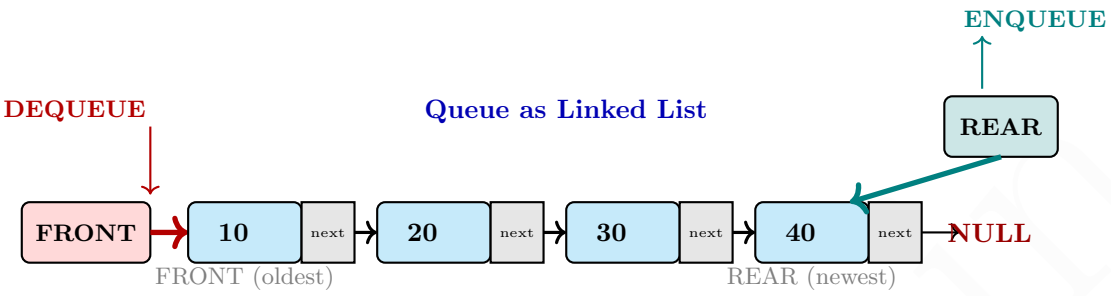
    def display(self):
        curr = self.top
        print("TOP ->", end=" ")
        while curr:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("NULL")


# --- Driver Code ---
s = StackLL()
s.push(10)
s.push(20)
s.push(30)
s.display()               # TOP -> 30 -> 20 -> 10 -> NULL
print("Peek:", s.peek()) # 30
print("Pop:", s.pop())   # 30
print("Pop:", s.pop())   # 20
s.display()               # TOP -> 10 -> NULL
```

4.5.2 Queue Using Linked List

Queue Using Linked List

We implement a queue using a singly linked list. The head of the linked list acts as the FRONT (for dequeue), and we maintain a separate **tail** pointer that always points to the last node, which acts as the REAR (for enqueue). Enqueue adds a new node at the tail in $O(1)$ time. Dequeue removes the head node in $O(1)$ time. Both FRONT and REAR operations are $O(1)$ because we maintain both pointers.



Algorithm: Queue-Enqueue-LL(Q, x)

```
1: new_node ← CreateNode(x)
2: if Q.FRONT = NULL then
3:     Q.FRONT ← new_node
4:     Q.REAR ← new_node
5: else
6:     Q.REAR.next ← new_node                ▷ link new node at end
7:     Q.REAR ← new_node                     ▷ move REAR to new node
8: end if
```

Algorithm: Queue-Dequeue-LL(Q)

```
1: if Q.FRONT = NULL then
2:     error "Queue Underflow"
3: end if
4: x ← Q.FRONT.data
5: Q.FRONT ← Q.FRONT.next                    ▷ move FRONT forward
6: if Q.FRONT = NULL then
7:     Q.REAR ← NULL                         ▷ queue is now empty
8: end if
9: return x
```

Python implementation: queue-linked-list.py

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class QueueLL:
    def __init__(self):
        self.FRONT = None    # head node, for dequeue
        self.REAR  = None    # tail node, for enqueue

    def is_empty(self):
        return self.FRONT is None

    def enqueue(self, x):
        new_node = Node(x)
        if self.is_empty():
            self.FRONT = new_node    # first element
            self.REAR  = new_node
```

```
        else:
            self.REAR.next = new_node    # link at REAR
            self.REAR      = new_node    # move REAR forward

    def dequeue(self):
        if self.is_empty():
            print("Queue Underflow")
            return -1
        x          = self.FRONT.data
        self.FRONT = self.FRONT.next    # move FRONT forward
        if self.FRONT is None:
            self.REAR = None            # queue is now empty
        return x

    def front(self):
        if self.is_empty():
            return -1
        return self.FRONT.data

    def rear(self):
        if self.is_empty():
            return -1
        return self.REAR.data

    def display(self):
        curr = self.FRONT
        print("FRONT ->", end=" ")
        while curr:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("NULL <- REAR")


# --- Driver Code ---
q = QueueLL()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.display()                         # 10 -> 20 -> 30
print("FRONT:", q.front())          # 10
print("REAR:", q.rear())            # 30
print("Dequeue:", q.dequeue())      # 10
print("Dequeue:", q.dequeue())      # 20
q.enqueue(40)
q.display()                         # 30 -> 40
```

Linked List Implementation: Stack vs Queue Summary

Property	Stack (LL)	Queue (LL)
Head acts as	TOP of stack	FRONT (Dequeue end)
Tail acts as	Not needed	REAR (Enqueue end)
Pointers needed	head (top) only	head (FRONT) + tail (REAR)
Insert at	Head in $O(1)$	Tail in $O(1)$
Delete at	Head in $O(1)$	Head in $O(1)$
Overflow possible	No (dynamic)	No (dynamic)
LIFO or FIFO	LIFO	FIFO

4.6 Practice Problems

Problem 112 ***[NAT]*** *The following C-like function operates on a singly linked list:*

```
Node* mystery(Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    Node* prev = NULL;
    Node* curr = head;
    int count = 0;

    while (curr != NULL) {
        Node* next_node = curr->next;
        if (count % 2 == 0) {
            curr->next = prev;
            prev = curr;
        } else {
            prev->next = curr;
        }
        curr = next_node;
        count++;
    }
    return prev;
}
```

For the linked list $1 \to 2 \to 3 \to 4 \to 5$*, what is the value of the head node of the list returned?* _____

Problem 113 ***[MCQ]*** *Consider the following recursive function on a singly linked list:*

Algorithm: Func(node)

1: **if** *node* = NULL **or** *node.next* = NULL **then**
2: **return** *node*
3: **end if**
4: *rest* ← Func(*node.next*)
5: *node.next.next* ← *node*
6: *node.next* ← NULL
7: **return** *rest*

If this function is called with the head of the list $1 \to 2 \to 3 \to 4$*, the resulting list from head to tail is:*

(A) $4 \to 3 \to 2 \to 1$

(B) $1 \to 2 \to 3 \to 4$

(C) $4 \to 1 \to 2 \to 3$

(D) $2 \to 1 \to 4 \to 3$

Problem 114 ***[NAT]*** *Consider a singly linked list of* n *nodes. The following operations are performed: insert at head* (n *times), then delete by position (deleting position 1 exactly* k *times). After all operations, the head contains the value that was inserted in which insertion step (1-indexed from first insert)? For* $n = 7$ *and* $k = 3$*, the head contains the value inserted at step number* _____*.*

Problem 115 ***[MSQ]*** *Which of the following operations on a singly linked list take* $O(1)$ *time in the worst case?*

(A) Insert at head (given only head pointer)

(B) Insert at tail (given only head pointer, no tail pointer)

(C) Delete head node (given only head pointer)

(D) Delete tail node (given only head pointer)

Problem 116 ***[MCQ]*** *Consider the following Python code on a linked list* $1 \to 2 \to 3 \to 4 \to 5$*:*

```
Node* transform(Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    slow->next = NULL;
    return head;
}
```

What does the list look like from head to tail after calling `transform(head)`*?*

(A) $1 \to 2 \to 3$

(B) $1 \to 2 \to 3 \to 4 \to 5$

(C) $1 \to 2$

(D) $3 \to 4 \to 5$

Problem 117 ***[NAT]*** *In a singly linked list, deleting the kth node from the end (without knowing the length) using the two-pointer technique requires moving the fast pointer exactly how many steps ahead of slow before starting the simultaneous movement? For* $k = 4$*, the fast pointer moves _____ steps ahead.*

Problem 118 ***[MCQ]*** *A singly linked list has n nodes. The following algorithm is run on it:*

Algorithm: Alter(head)

1: *curr* ← *head*
2: **while** *curr* ≠ NULL **and** *curr.next* ≠ NULL **do**
3: *temp* ← *curr.next*
4: *curr.next* ← *temp.next*
5: *temp.next* ← NULL
6: *curr* ← *curr.next*
7: **end while**

For the list $1 \to 2 \to 3 \to 4 \to 5 \to 6$*, what is the resulting list?*

(A) $1 \to 3 \to 5$

(B) $2 \to 4 \to 6$

(C) $1 \to 2 \to 4 \to 6$

(D) $1 \to 3 \to 5 \to 6$

Problem 119 ***[NAT]*** *The following function is called on a linked list* $1 \to 2 \to 3 \to 4 \to 5$*:*

```
Node* partial_reverse(Node* head, int k) {
    Node* prev = NULL;
    Node* curr = head;
    int count = 0;

    while (curr != NULL && count < k) {
        Node* next_node = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next_node;
        count++;
    }

    if (head != NULL) {
        head->next = curr;
    }

    return prev;
}
```

After calling `partial_reverse(head, 3)`*, what is the value of the* ***second*** *node from the head?* _____

Problem 120 ***[MSQ]*** *Which of the following statements about reversing a singly linked list are TRUE?*

(A) The iterative reversal requires exactly 3 pointer variables: prev, curr, and next.

(B) The recursive reversal has a call stack depth of $O(n)$, where n is the number of nodes.

(C) Both iterative and recursive reversal take $O(n)$ time and $O(1)$ extra space.

(D) After recursive reversal, the original head node's `next` *must be explicitly set to NULL.*

Problem 121 ***[MCQ]*** *Consider the following incorrect attempt at iterative reversal:*

```
Node* wrong_reverse(Node* head) {
    Node* prev = NULL;
    Node* curr = head;
    while (curr != NULL) {
        curr->next = prev;
        prev = curr;
        curr = curr->next;
    }
    return prev;
}
```

For list $1 \to 2 \to 3 \to 4$, this function returns a list containing how many nodes?

(A) 4

(B) 3

(C) 2

(D) 1

Problem 122 ***[MCQ]*** *Consider a linked list $1 \to 2 \to 3 \to 4 \to 5 \to 6$, where node 6's next pointer points back to node 3. Using Floyd's algorithm, at which node do the slow and fast pointers first meet?*

(A) Node with value 3

(B) Node with value 4

(C) Node with value 5

(D) Node with value 6

Problem 123 ***[NAT]*** *In Floyd's cycle detection algorithm, slow moves 1 step and fast moves 2 steps per iteration. Consider a linked list where the linear part (before cycle) has $F = 3$ nodes and the cycle has $C = 4$ nodes. The slow and fast pointers start at the head. The number of iterations (steps taken by slow pointer) until they first meet inside the cycle is* _____.

Problem 124 ***[MCQ]*** *The slow-fast pointer technique is used to find the middle of a linked list. For a list of length n, when the fast pointer terminates (reaches NULL or the last node), slow is at the middle. For an even-length list $1 \to 2 \to 3 \to 4$ using the condition* `while fast.next and fast.next.next`*, the middle node returned is:*

(A) Node with value 2

(B) Node with value 3

(C) Node with value 1

(D) Node with value 4

Problem 125 ***[NAT]*** *Consider the following function:*

```
def middle_value(head):
    slow = head
    fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow.data
```

What value is returned for the list $10 \to 20 \to 30 \to 40 \to 50 \to 60$*?* _____

Problem 126 ***[MCQ]*** *Two sorted linked lists* $L_1 = 1 \to 4 \to 7 \to 9$ *and* $L_2 = 2 \to 3 \to 8 \to 10$ *are merged using the standard dummy-node approach. How many pointer assignment operations (of the form* `curr.next = ...`*) are performed in total during the merge?*

(A) 7

(B) 8

(C) 9

(D) 6

Problem 127 ***[MCQ]*** *Consider the following recursive function for merging two sorted linked lists:*

Algorithm: Merge(h1, h2)

```
1: if h1 = NULL then return h2
2: end if
3: if h2 = NULL then return h1
4: end if
5: if h1.data ≤ h2.data then
6:     h1.next ← Merge(h1.next, h2)
7:     return h1
8: else
9:     h2.next ← Merge(h1, h2.next)
10:    return h2
11: end if
```

For $h_1 = 1 \to 3 \to 5$ *and* $h_2 = 2 \to 4 \to 6$*, the maximum depth of the recursion stack during this call is:*

(A) 3

(B) 6

(C) 7

(D) 4

Problem 128 ***[MSQ]*** *Which of the following are TRUE about merging two sorted singly linked lists of sizes* m *and* n *respectively?*

(A) The time complexity of the iterative merge is $\Theta(m + n)$*.*

(B) The recursive merge uses $O(m + n)$ *stack space in the worst case.*

(C) No new nodes need to be created; only pointer reassignments are needed.

(D) The iterative dummy-node approach requires $O(m + n)$ *extra space for the dummy node.*

Problem 129 ***[MSQ]*** *A student tries to delete the* k*th node from the end of a singly linked list using two pointers, without finding the length. Which of the following are TRUE about this approach?*

(A) We need to maintain a `prev` *pointer for the slow pointer to perform deletion in* $O(1)$ *after finding the node.*

(B) Moving fast $k + 1$ *steps ahead and then both together allows deletion without a separate prev pointer.*

(C) Moving fast k *steps ahead and then both together makes slow land exactly on the node to delete.*

(D) This approach handles deletion of the head node (when k *equals list length) without special casing.*

Problem 130 ***[MCQ]*** *In a doubly linked list, inserting a new node* `new` *between nodes* `p` *and* `q` *(where* `p.next = q`*) requires pointer assignments in what minimum number of steps?*

(A) 2

(B) 3

(C) 4

(D) 5

Problem 131 ***[MCQ]*** *Consider a doubly linked list where we want to delete a node* `X` *that is neither the head nor the tail. The minimum number of pointer assignments needed is:*

(A) 1

(B) 2

(C) 3

(D) 4

Problem 132 ***[MSQ]*** *Which of the following are TRUE about a circular singly linked list compared to a linear singly linked list?*

(A) Traversal must check whether the current node equals the head to detect the end, not check for NULL.

(B) Inserting at the tail is $O(1)$ if we maintain a tail pointer, because we can reach head from tail.

(C) Detecting whether a list is circular requires Floyd's algorithm or explicit tracking; checking `next == NULL` *is insufficient.*

(D) Deleting the head requires updating the tail's `next` *pointer as well.*

Problem 133 ***[MSQ]*** *A stack is implemented using a singly linked list (head = top). Which of the following are TRUE compared to an array-based stack of fixed size n?*

(A) The linked list stack never overflows, since nodes are created dynamically.

(B) Push and Pop are both $O(1)$ in the linked list stack.

(C) The linked list stack uses less total memory than an array-based stack for n elements.

(D) The linked list stack requires more memory per element because of the extra pointer field.

Problem 134 ***[MCQ]*** *Consider a queue implemented using a linked list. The FRONT pointer is lost (becomes NULL by mistake) while the queue still has elements. Which operation can still be performed correctly using only the REAR pointer?*

(A) Dequeue (remove from FRONT)

(B) Enqueue (add at REAR)

(C) Peek FRONT element

(D) Check if the queue is empty

Problem 135 ***[MCQ]*** *A student implements a queue using a singly linked list, but maintains only a FRONT pointer (no REAR pointer). What is the time complexity of Enqueue (insertion at tail) in this implementation?*

(A) $O(1)$

(B) $O(\log n)$

(C) $O(n)$

(D) $O(n^2)$

Problem 136 ***[MCQ]*** *Consider the following C function on a linked list:*

```
void func(Node* head) {
    if (head == NULL || head->next == NULL) {
        return;
    }

    Node* slow = head;
    Node* fast = head;
    Node* prev_slow = NULL;

    while (fast != NULL && fast->next != NULL) {
        prev_slow = slow;
```

```
        slow = slow->next;
        fast = fast->next->next;
    }

    if (prev_slow != NULL) {
        prev_slow->next = NULL;
    }

    Node* second_half = slow;
    second_half = reverse(second_half); // standard reversal

    Node* curr1 = head;
    Node* curr2 = second_half;

    while (curr1 != NULL && curr2 != NULL) {
        printf("%s ", (curr1->data == curr2->data) ? "True" : "False");
        curr1 = curr1->next;
        curr2 = curr2->next;
    }
}
```

For input $1 \to 2 \to 3 \to 2 \to 1$*, what does this function check?*

(A) Whether the list is sorted

(B) Whether the list is a palindrome

(C) Whether the list has a cycle

(D) Whether the list has all unique elements

Problem 137 ***[MCQ]*** *Consider two linked list implementations, one using an array and one using a singly linked list. A program performs n insertions at random positions (uniformly distributed). The expected time complexity for all insertions combined is:*

(A) $O(n)$ *for array,* $O(n)$ *for linked list*

(B) $O(n^2)$ *for array (due to shifts),* $O(n^2)$ *for linked list (due to traversal)*

(C) $O(n \log n)$ *for array,* $O(n^2)$ *for linked list*

(D) $O(n^2)$ *for array,* $O(n)$ *for linked list*

Problem 138 ***[MCQ]*** *Consider the following function applied to list :*

```
bool mystery(Node* head) {
    Stack stack;
    Node* curr = head;

    while (curr != NULL) {
        push(&stack, curr->data);
        curr = curr->next;
    }

    curr = head;
    while (curr != NULL) {
        if (curr->data != pop(&stack)) {
            return false;
        }
        curr = curr->next;
    }

    return true;
}
```

What does this function check, and what does it return for input $1 \to 2 \to 3 \to 2 \to 1$*?*

(A) Checks if sorted; returns False

(B) Checks if palindrome; returns True

(C) Checks if all elements distinct; returns True

(D) Checks if palindrome; returns False

Problem 139 ***[NAT]*** *Consider the following algorithm on a singly linked list of length n:*

Algorithm: CountOps(head)

```
1: ops ← 0
2: curr ← head
3: while curr ≠ NULL do
4:     inner ← curr.next
5:     while inner ≠ NULL do
6:         if inner.data = curr.data then
7:             ops ← ops + 1
8:         end if
9:         inner ← inner.next
10:    end while
11:    curr ← curr.next
12: end while
13: return ops
```

For the list $1 \to 2 \to 1 \to 3 \to 2 \to 1$, the value returned is _____.

Problem 140 ***[MSQ]*** *A function is defined recursively on linked lists as:*

$$f(NULL) = 0, \quad f(node) = f(node.next) + node.data$$

For the list $3 \to -1 \to 4 \to -2 \to 5$, which of the following are TRUE?

(A) $f(head) = 9$

(B) The function computes the sum by visiting nodes in reverse order (tail to head due to recursion).

(C) The recursion depth for a list of length 5 is 6 (including the NULL base case).

(D) $f(head) = f(head.next) + 3$

Problem 141 *Which of the following operations is performed more efficiently by doubly linked list than by linear linked list?*

A. Deleting a node whose location is given

B. Searching an unsorted list for a given item

C. Inserting a node after the node with a given location

D. Traversing the list to process each node

Problem 142 *A doubly linked list is declared as:*

```
struct Node {
    int Value;
    struct Node *Fwd;
    struct Node *Bwd;
};
```

Where `Fwd` and `Bwd` represent forward and backward link to the adjacent elements of the list. Which of the following segment of code deletes the node pointed to by `X` from the doubly linked list, if it is assumed that `X` points to neither the first nor the last node of the list?

A. `X->Bwd->Fwd = X->Fwd; X->Fwd->Bwd = X->Bwd;`

B. `X->Bwd.Fwd = X->Fwd; X.Fwd->Bwd = X->Bwd;`

C. `X.Bwd->Fwd = X.Bwd; X->Fwd.Bwd = X.Bwd;`

D. `X->Bwd->Fwd = X->Bwd; X->Fwd->Bwd = X->Fwd;`

Problem 143 *Consider a singly linked list of the form where F is a pointer to the first element in the linked list and L is the pointer to the last element in the list. The time of which of the following operations depends on the length of the list?*

A. Delete the last element of the list

B. Delete the first element of the list

C. Add an element after the last element of the list

D. Interchange the first two elements of the list

Problem 144 *The following steps in a linked list:*

```
p = getnode()
info(p) = 10
next(p) = list
list = p
```

result in which type of operation?

A. Pop operation in stack

B. Removal of a node

C. Inserting a node

D. Modifying an existing node

gateaim.in

4.7 GATE PYQs

GATEPYQ 65 (GATE CSE 2026) *Consider the following code snippet in C language that computes the number of nodes in a non-empty singly linked list pointed to by the pointer variable* `head`.

```
struct node {
  int elt;
  struct node *next;
};

int getListSize (struct node *head)
{
  if ( E1 ) return 1;
  return E2 ;
}
```

Which one of the following options gives the correct replacements for the expressions `E1` *and* `E2`*?*

A. `E1: head == NULL`
   `E2: 1 + getListSize(head)`

B. `E1: head->next == NULL`
   `E2: 1 + getListSize(head->next)`

C. `E1: head == NULL`
   `E2: 1 + getListSize(head->next)`

D. `E1: head->next == NULL`
   `E2: 1 + getListSize(head)`

GATEPYQ 66 (GATE CSE 2025) *Consider an unordered list of N distinct integers. What is the minimum number of element comparisons required to find an integer in the list that is NOT the largest in the list?*

A. 1

B. $N - 1$

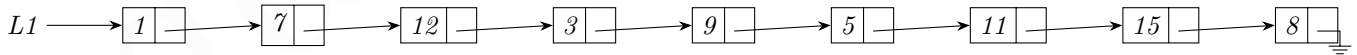
C. N

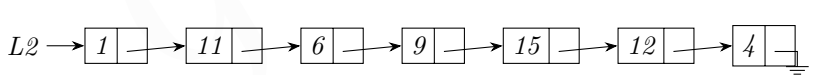
D. $2N - 1$

GATEPYQ 67 (GATE CSE 2025) *Let* `LIST` *be a datatype for an implementation of linked list defined as follows:*

```
typedef struct list {
    int data;
    struct list *next;
} LIST;
```

Suppose the program has created two linked lists, `L1` *and* `L2`*, whose contents are given in the question context (code for creating* `L1` *and* `L2` *is not provided here).* `L1` *contains 9 nodes, and* `L2` *contains 7 nodes.*





Consider the following C program segment that modifies the list `L1`*. The number of nodes in* `L1` *after the execution of the code segment is* _____*. (Answer in integer)*

```
int find (int query, LIST *list) {
    while (list != NULL){
        if(list->data == query) return 1;
        list = list->next;
    }
    return 0;
}

int main () {
    ... ... ...
```

```
    ptr1=L1; ptr2=L2;
    while (ptr1->next != NULL) {
        query = ptr1->next->data;
        if (find (query, L2))
            ptr1->next = ptr1->next->next;
        else ptr1 = ptr1->next;
    }
    ... ... ...
    return 0;
}
```

GATEPYQ 68 (GATE CSE 2023) *Let* `SLLdel` *be a function that deletes a node in a singly-linked list given a pointer to the node and a pointer to the head of the list. Similarly, let* `DLLdel` *be another function that deletes a node in a doubly-linked list given a pointer to the node and a pointer to the head of the list.*
Let n denote the number of nodes in each of the linked lists. Which one of the following choices is TRUE about the worst-case time complexity of `SLLdel` *and* `DLLdel`*?*

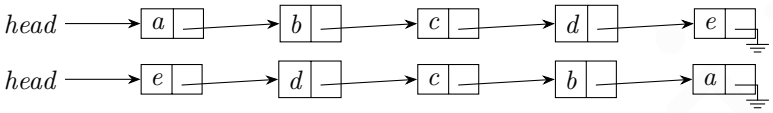
A. `SLLdel` *is* $O(1)$ *and* `DLLdel` *is* $O(n)$

B. *Both* `SLLdel` *and* `DLLdel` *are* $O(\log n)$

C. *Both* `SLLdel` *and* `DLLdel` *are* $O(1)$

D. `SLLdel` *is* $O(n)$ *and* `DLLdel` *is* $O(1)$

GATEPYQ 69 (GATE CSE 2022) *Consider the problem of reversing a singly linked list. To take an example, given a linked list, the reversed linked list should point in the opposite direction.*



Which one of the following statements is TRUE about the time complexity of algorithms that solve the above problem in $O(1)$ space?

A. *The best algorithm for the problem takes $\Theta(n)$ time in the worst case.*

B. *The best algorithm for the problem takes $\Theta(n \log n)$ time in the worst case.*

C. *The best algorithm for the problem takes $\Theta(n^2)$ time in the worst case.*

D. *It is not possible to reverse a singly linked list in $O(1)$ space.*

GATEPYQ 70 (GATE CSE 2020) *What is the worst case time complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order?*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(1)$

GATEPYQ 71 (GATE CSE 2017) *A circular queue has been implemented using a single linked list where each node consists of a value and a single pointer pointing to the next node. We maintain exactly two external pointers FRONT and REAR pointing to the front node and the rear node of the queue, respectively. Which of the following statements is/are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in $O(1)$ time?*

I. *Next pointer of front node points to the rear node.*

II. *Next pointer of rear node points to the front node.*

A. *I only*

B. *II only*

C. *Both I and II*

D. *Neither I nor II*

GATEPYQ 72 (GATE CSE 2017) *Consider the C code fragment given below:*

```
typedef struct node {
    int data;
    struct node* next;
} node;

void join (node* m, node* n) {
    node* p = n;
    while (p->next != NULL) {
        p = p->next;
    }
    p->next = m;
}
```

Assuming that `m` *and* `n` *point to valid* `NULL`*-terminated linked lists, invocation of* `join` *will:*

A. *append list* `m` *to the end of list* `n` *for all inputs.*

B. *either cause a null pointer dereference or append list* `m` *to the end of list* `n`*.*

C. *cause a null pointer dereference for all inputs.*

D. *append list* `n` *to the end of list* `m` *for all inputs.*

GATEPYQ 73 (GATE CSE 2016) *N items are stored in a sorted doubly linked list. For a delete operation, a pointer is provided to the record to be deleted. For a decrease-key operation, a pointer is provided to the record on which the operation is to be performed.*
An algorithm performs the following operations on the list in this order: $\Theta(N)$ delete, $O(\log N)$ insert, $O(\log N)$ find, and $\Theta(N)$ decrease-key. What is the time complexity of all these operations put together?

A. $O(\log^2 N)$

B. $O(N)$

C. $O(N^2)$

D. $\Theta(N^2 \log N)$

GATEPYQ 74 (GATE CSE 2015) *An unordered list contains n distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is:*

A. $\Theta(n \log n)$

B. $\Theta(n)$

C. $\Theta(\log n)$

D. $\Theta(1)$

GATEPYQ 75 (GATE CSE 2010) *The following C function takes a singly-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank.*

```
typedef struct node {
     int value;
     struct node *next;
} Node;

Node *move_to_front(Node *head) {
       Node *p, *q;
       if ((head == NULL) || (head->next == NULL)) return head;
       q = NULL; p = head;
       while (p->next != NULL) {
             q = p;
             p = p->next;
       }
       ------------------------------
       return head;
}
```

Choose the correct alternative to replace the blank line.

A. `q = NULL; p->next = head; head = p;`

B. `q->next = NULL; head = p; p->next = head;`

C. `head = p; p->next = q; q->next = NULL;`

D. `q->next = NULL; p->next = head; head = p;`

GATEPYQ 76 (GATE CSE 2008) *The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?*

```
struct node {
   int value;
   struct node *next;
};

void rearrange(struct node *list) {
   struct node *p, *q;
   int temp;
   if (!list || !list->next) return;
   p = list; q = list->next;
   while (q)
   {
      temp = p->value;
      p->value = q->value;
      q->value = temp;
      p = q->next;
      q = p ? p->next : 0;
   }
}
```

A. *1, 2, 3, 4, 5, 6, 7*

B. *2, 1, 4, 3, 6, 5, 7*

C. *1, 3, 2, 5, 4, 7, 6*

D. *2, 3, 4, 5, 6, 7, 1*

GATEPYQ 77 (GATE IT 2004) *Let P be a singly linked list. Let Q be the pointer to an intermediate node x in the list. What is the worst-case time complexity of the best-known algorithm to delete the node x from the list?*

A. $O(n)$

B. $O(\log_2 n)$

C. $O(\log n)$

D. $O(1)$

GATEPYQ 78 (GATE CSE 2004) *Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership, cardinality will be the slowest?*

A. *union only*

B. *intersection, membership*

C. *membership, cardinality*

D. *union, intersection*

GATEPYQ 79 (GATE CSE 2004) *A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time?*

A. *rear node*

B. *not possible with a single pointer*

C. front node

D. node next to front

GATEPYQ 80 (GATE CSE 2003) *Consider the function `f` defined below:*

```
struct item
{
    int data;
    struct item *next;
};

int f(struct item *p)
{
    return ((p == NULL) || (p->next == NULL) ||
            ((p->data <= p->next->data) && f(p->next)));
}
```

For a given linked list `p`, the function `f` returns 1 if and only if:

A. the list is empty or has exactly one element

B. the elements in the list are sorted in non-decreasing order of data value

C. the elements in the list are sorted in non-increasing order of data value

D. not all elements in the list have the same data value

GATEPYQ 81 (GATE CSE 2002) *In the worst case, the number of comparisons needed to search a single linked list of length n for a given element is:*

A. $\log n$

B. $n/2$

C. $\log_2 n - 1$

D. n

GATEPYQ 82 (GATE CSE 1997) *The concatenation of two lists is to be performed in $O(1)$ time. Which of the following implementations of a list should be used?*

A. Singly linked list

B. Doubly linked list

C. Circular doubly linked list

D. Array implementation of list

GATEPYQ 83 (GATE CSE 1995) *For merging two sorted lists of sizes m and n into a sorted list of size $m + n$, we require comparisons of:*

A. $O(m)$

B. $O(n)$

C. $O(m + n)$

D. $O(\log m + \log n)$

GATEPYQ 84 (GATE CSE 1987) *In a circular linked list organization, insertion of a record involves modification of:*

A. One pointer.

B. Two pointers.

C. Multiple pointers.

D. No pointer.

4.8 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
29	Introduction to Linked Lists	https://www.youtube.com/watch?v=J3wzk3vPb6g	
30	Singly Linked List Insertion: Head, Tail & Middle	https://www.youtube.com/watch?v=OILBBbBHpQE	
31	Singly Linked List Deletion: Head, Value & Position	https://www.youtube.com/watch?v=KlDJv6HTvK8	
32	Singly Linked List Reversal: Iterative & Recursive	https://www.youtube.com/watch?v=G-D2Na8p4dY	
33	Doubly Linked List Insertion: Head, Tail & Position	https://www.youtube.com/watch?v=9v79SfC54WY	
34	Doubly Linked List Deletion: Head, Value & Position	https://www.youtube.com/watch?v=QfBNTqB_rrI	
35	Circular Linked List Insertion: Head, Tail & Position	https://www.youtube.com/watch?v=yrOnqWSApFQ	
36	Circular Linked List Deletion: Head, Tail & Position	https://www.youtube.com/watch?v=ic1UqR2t_98	

Lec. No	Title	URL	QR Code
37	Classic Problems in Linked Lists	https://www.youtube.com/watch?v=T_GWv4Es-oA	
38	GATE PYQs on Linked Lists	https://www.youtube.com/watch?v=_thOnD3Wjmc	
39	Problem Solving on Linked Lists	https://www.youtube.com/watch?v=da6Qz7zD-aA	

gateaim

Chapter 5

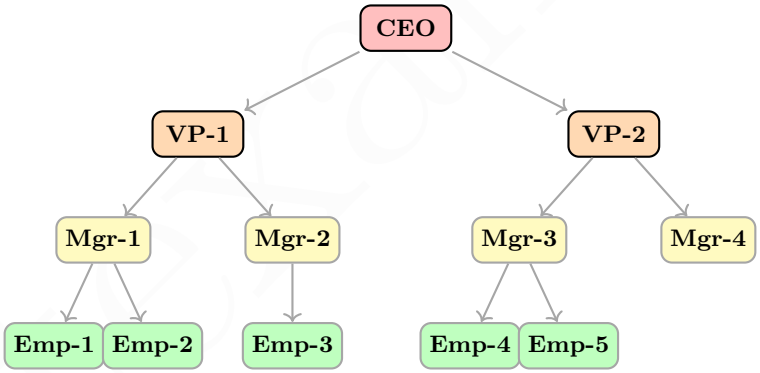
Trees

Concept: What is a Tree

A tree is a non-linear data structure where data is organized in a hierarchical way, like a family tree. Unlike a linked list where nodes are connected in a straight line, a tree has one starting point at the top, and it branches out downward. Each node can have zero or more children, and every node except the starting one has exactly one parent. There are no loops or cycles in a tree.

Example 210: Analogy: A Company Hierarchy

Think of a company. The CEO is at the top. Below the CEO are Vice Presidents. Below each VP are Managers. Below each Manager are Employees. This structure is exactly a tree. The CEO is the starting point, each person reports to exactly one person above them, and no one reports to themselves in a loop.



5.1 Terminology and Structure

Concept: Root Node

The root is the topmost node of the tree. It is the starting point. There is exactly one root in any tree. It has no parent. In the company analogy, the CEO is the root.

Concept: Leaf Node

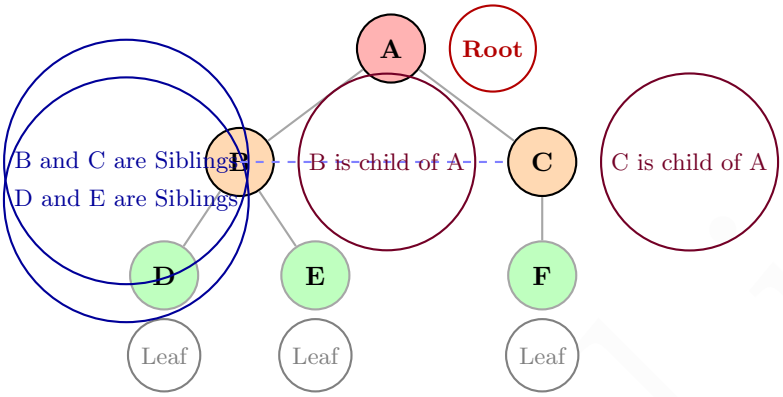
A leaf node is a node that has no children. It is at the bottom of the tree. In the company analogy, employees who manage nobody are the leaves.

Concept: Internal Node

An internal node is any node that has at least one child. It is not a leaf. The root is also an internal node (unless the tree has only one node). All managers and VPs are internal nodes in our analogy.

Concept: Parent, Child, and Sibling

- **Parent:** The node directly above a given node is its parent. Every node except the root has exactly one parent.
- **Child:** Nodes directly below a given node are its children. A node can have zero or more children.
- **Sibling:** Two nodes are siblings if they share the same parent. Brothers and sisters in a family tree.



Concept: Depth of a Node

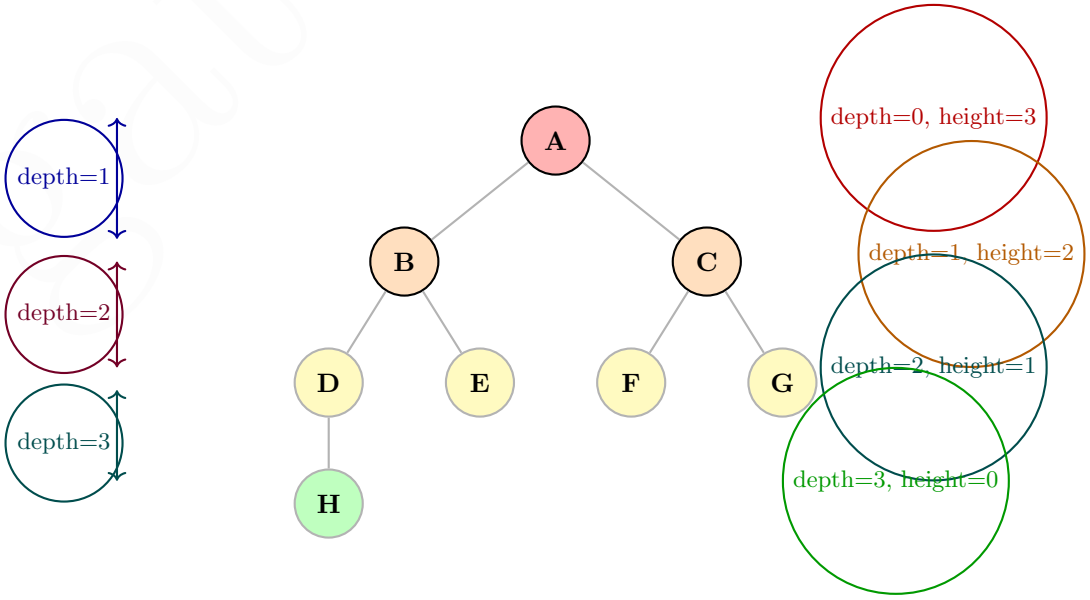
The depth of a node is the number of edges from the root to that node. The root has depth 0. A child of the root has depth 1, and so on. Depth tells us how far down a node is from the top.

Concept: Height of a Node and Height of a Tree

The height of a node is the number of edges on the longest path from that node down to a leaf. A leaf node has height 0. The height of the tree is the height of the root node, which equals the maximum depth of any leaf.

Example 211: Height and Depth: Easy to Remember

Think of a building. Depth is like the floor number counting from the ground floor. Ground floor (root) is floor 0. Going deeper means going higher floor number. Height of a floor is how many floors are still above it (looking downward from that floor). Height of the building is the total number of floors above ground.



Concept: Degree of a Node

The degree of a node is the number of children that node has. A leaf node has degree 0. In a binary tree, degree can be 0, 1, or 2 at most. The degree of the tree is the maximum degree of any node in the tree.

5.1.1 Full, Complete, and Perfect Binary Tree

Concept: Full Binary Tree

A full binary tree is a binary tree where every node has either 0 children or exactly 2 children. No node has exactly 1 child. Think of it as, every internal node must be completely paired, like a bracket tournament where every match has exactly two players.

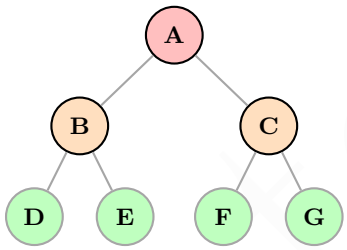
Concept: Complete Binary Tree

A complete binary tree is a binary tree where all levels are completely filled, except possibly the last level. The last level must be filled from left to right, with no gaps in the middle. Think of filling seats in a cinema row by row, from left to right, without leaving any empty seat between occupied ones.

Concept: Perfect Binary Tree

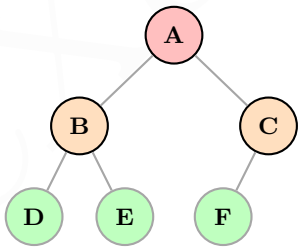
A perfect binary tree is a binary tree where all internal nodes have exactly 2 children, and all leaf nodes are at the same level. It is the most regular shape. A perfect binary tree with height h has exactly $2^{h+1} - 1$ total nodes.

Full Binary Tree



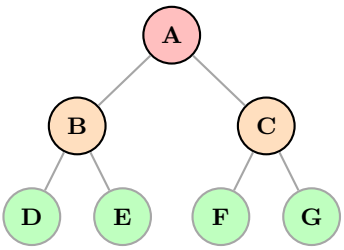
every node: 0 or 2 children

Complete Binary Tree



last level filled left to right

Perfect Binary Tree



all leaves at same level

Concept: Relation Between Leaves and Internal Nodes in a Full Binary Tree

In a full binary tree, if the number of internal nodes is I and the number of leaf nodes is L, then

$$L = I + 1$$

This is a very useful and frequently tested property. No matter what shape the full binary tree is, the number of leaves is always exactly one more than the number of internal nodes. Intuition: every time we add an internal node (which contributes 2 children but removes itself from being a leaf), the leaf count increases by exactly 1.

Example 212: Verifying: $L = I + 1$

In the full binary tree above with nodes A, B, C, D, E, F, G:

- Internal nodes (have children): A, B, C. So $I = 3$.
- Leaf nodes (no children): D, E, F, G. So $L = 4$.
- Check: $L = I + 1 \Rightarrow 4 = 3 + 1$. Correct.

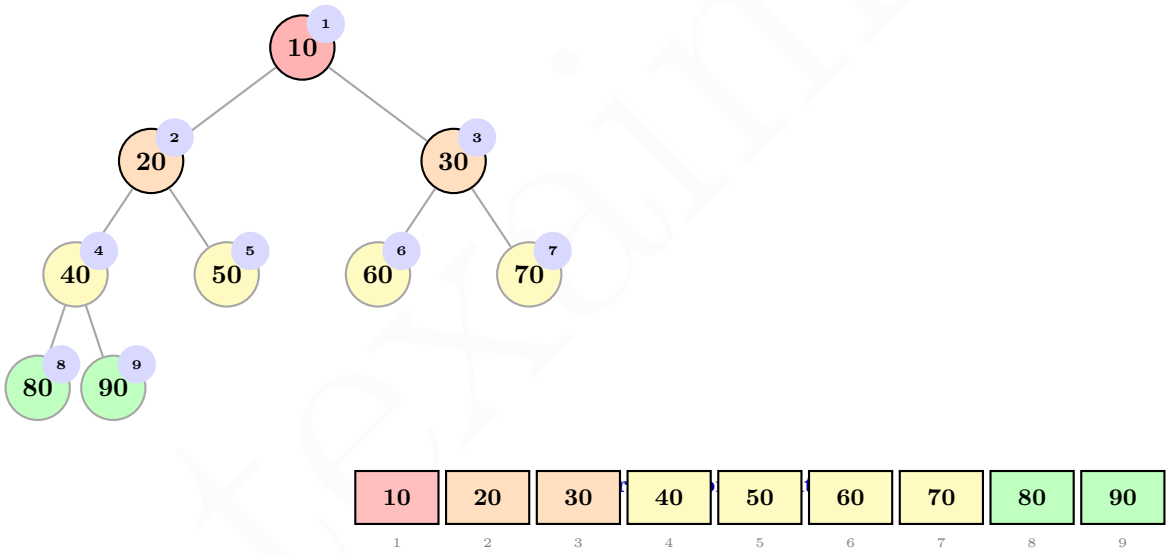
5.1.2 Array Representation of a Binary Tree

Concept: Array Representation (1-Indexed)

A binary tree can be stored in a plain array without using any pointers. We number the nodes level by level, left to right, starting from index 1. The root is at index 1. For any node at index i:

- Its **parent** is at index $\lfloor i/2 \rfloor$
- Its **left child** is at index $2i$
- Its **right child** is at index $2i + 1$

This is exactly the structure used in heaps, and it works best for complete or perfect binary trees, because sparse trees waste many array slots.



Example 213: Verifying Array Index Formulas

For node at index 4 (value 40):

- Parent: $\lfloor 4/2 \rfloor = 2 \rightarrow$ index 2 has value 20. Correct, 40's parent is 20.
- Left child: $2 \times 4 = 8 \rightarrow$ index 8 has value 80. Correct.
- Right child: $2 \times 4 + 1 = 9 \rightarrow$ index 9 has value 90. Correct.

Python implementation: tree-array-representation.py

```
class BinaryTreeArray:
    def __init__(self, maxsize):
        self.arr     = [None] * (maxsize + 1)  # 1-indexed
        self.maxsize = maxsize

    def get_parent(self, i):
```

```
        if i <= 1:
            return None          # root has no parent
        return self.arr[i // 2]

    def get_left_child(self, i):
        if 2 * i > self.maxsize:
            return None
        return self.arr[2 * i]

    def get_right_child(self, i):
        if 2 * i + 1 > self.maxsize:
            return None
        return self.arr[2 * i + 1]


# --- Driver Code ---
t = BinaryTreeArray(9)
vals = [None, 10, 20, 30, 40, 50, 60, 70, 80, 90]
for i in range(1, 10):
    t.arr[i] = vals[i]

print("Parent of index 4:", t.get_parent(4))          # 20
print("Left child of index 2:", t.get_left_child(2)) # 40
print("Right child of index 2:", t.get_right_child(2)) # 50
```

5.2 Binary Tree Formulas

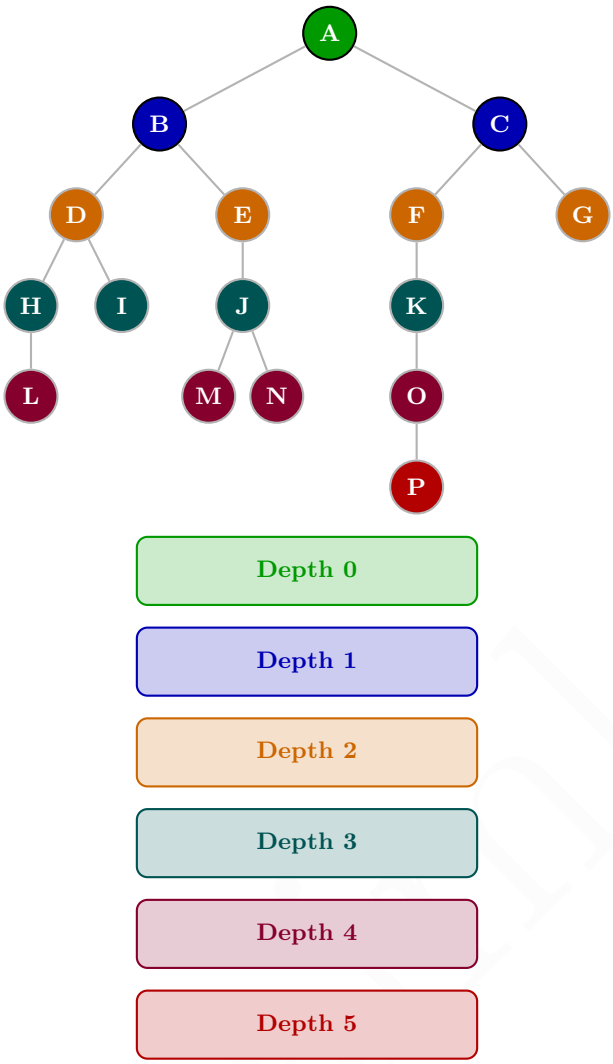
Concept: Depth of a Node

The **depth** of a node is the number of edges from the root down to that node. The root itself has depth 0. A child of the root has depth 1. Each step downward increases depth by 1.
Simple rule: depth of a node = number of edges you travel starting from the root to reach that node.

Example 214: Depth Examples on the Tree Below

Consider the tree with root A, children B and C, where B has children D and E, C has children F and G, D has children H and I, E has child J, F has child K, J has children M and N, H has child L, K has child O, O has child P.

- A (root): depth = 0
- B, C: depth = 1
- D, E, F, G: depth = 2
- H, I, J, K: depth = 3
- L, M, N, O: depth = 4
- P: depth = 5



Concept: Level of a Node

The **level** of a node is the same as its depth. A level of a tree consists of all nodes that share the same depth value.

- Root is at level 0.
- All direct children of the root are at level 1.
- Their children are at level 2, and so on.

The terms depth and level are used interchangeably for nodes in a binary tree.

Concept: Height of a Node and Height of a Tree

The **height** of a node is the number of edges on the **longest path going downward** from that node to any leaf below it.

- A leaf node has height = 0 (no edges going down from it).
- The **height of the tree** is the height of the root node.
- Height of tree = maximum depth of any leaf in the tree.

Think of it this way: depth measures how far down you are from the top. Height measures how far down you can still go from a given node.

Example 215: Height Examples on the Same Tree

Using the same tree as before:

- P (a leaf): height = 0

- O (parent of P): height = 1
- K (parent of O): height = 2
- F (parent of K): height = 3
- C (parent of F and G): height = 4
- A (root): height = 5 (longest path: A → C → F → K → O → P)

Height of this tree = 5.

Concept: Number of Levels vs Height

If a tree has height h, then the root is at level 0 and the deepest leaf is at level h. So the total number of levels is:

$$\text{Total levels } L = h + 1$$

Equivalently, if we know the number of levels L, then the height is $h = L - 1$.

Example 216: Levels and Height

The tree above has height $h = 5$. Total levels $= 5 + 1 = 6$ (levels 0, 1, 2, 3, 4, 5). If someone says a tree has 8 levels, the height is $8 - 1 = 7$.

Concept: Maximum Nodes at a Given Level

In a binary tree, each node can have at most 2 children. So the number of nodes at each level can at most double compared to the level above it.

- Level 0 (root): at most $1 = 2^0$ node
- Level 1: at most $2 = 2^1$ nodes
- Level 2: at most $4 = 2^2$ nodes
- Level l: at most 2^l nodes

$$\boxed{\text{Maximum nodes at level } l = 2^l}$$

Example 217: Max Nodes at Level Examples

- At level 0: max $2^0 = 1$ node (the root).
- At level 3: max $2^3 = 8$ nodes.
- At level 10: max $2^{10} = 1024$ nodes.

Concept: Maximum Total Nodes for Height h

To get the maximum total nodes in a binary tree of height h, we sum the maximum nodes over all levels from 0 to h:

$$\sum_{l=0}^{h} 2^l = 2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$$

$$\boxed{\text{Maximum total nodes for height } h = 2^{h+1} - 1}$$

If the total number of levels is $L = h + 1$, this can also be written as $2^L - 1$.
This maximum is achieved only by a **perfect binary tree**, where every level is completely filled.

Example 218: Max Total Nodes Example

- Height $h = 2$: max nodes $= 2^{2+1} - 1 = 7$.
- Height $h = 4$: max nodes $= 2^{4+1} - 1 = 31$.
- Height $h = 10$: max nodes $= 2^{11} - 1 = 2047$.

Concept: Minimum Total Nodes for Height h

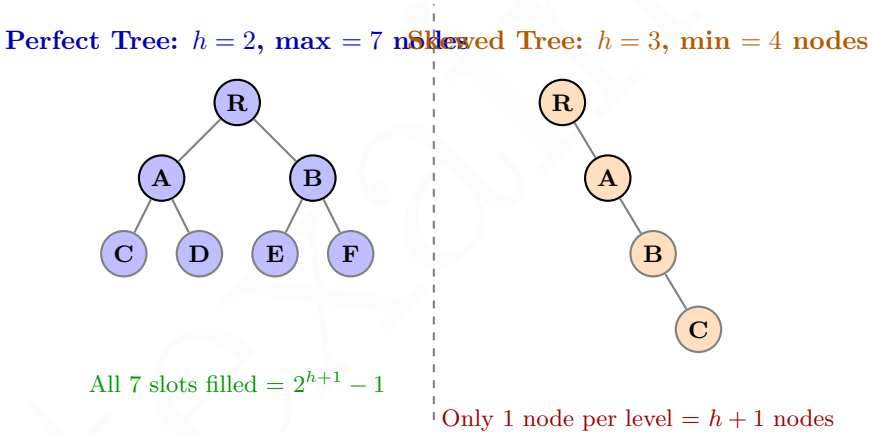
The minimum number of nodes in a binary tree of height h occurs when the tree is as sparse as possible. This happens in a **skewed (degenerate) tree**, where every internal node has exactly one child, making the tree look like a chain. In this case, each level has exactly one node, so:

$$\boxed{\text{Minimum total nodes for height } h = h + 1}$$

Example 219: Min Total Nodes Example

- Height $h = 3$: min nodes $= 3 + 1 = 4$ (a straight chain of 4 nodes).
- Height $h = 7$: min nodes $= 7 + 1 = 8$.

A right-skewed tree with 4 nodes: root → child → grandchild → great-grandchild. Each node has only one child. Height = 3, nodes = 4.



Concept: Height Range for n Nodes

Given n nodes in a binary tree, the height h is constrained from both sides:
Minimum height (achieved by a complete/perfect-shaped tree, packed as wide as possible):

$$h_{\min} = \lceil \log_2(n+1) \rceil - 1$$

Maximum height (achieved by a fully skewed tree, one node per level):

$$h_{\max} = n - 1$$

So for any binary tree with n nodes:

$$\lceil \log_2(n+1) \rceil - 1 \;\leq\; h \;\leq\; n - 1$$

Example 220: Height Range for $n = 7$ Nodes

- $h_{\min} = \lceil \log_2(7+1) \rceil - 1 = \lceil 3 \rceil - 1 = 2$ (a perfect tree with 7 nodes has height 2).
- $h_{\max} = 7 - 1 = 6$ (a skewed tree with 7 nodes has height 6).

So for 7 nodes: $2 \leq h \leq 6$.

Summary: Key Binary Tree Formulas

Quantity	Formula
Max nodes at level l (root = level 0)	2^l
Max total nodes for height h	$2^{h+1} - 1$
Max total nodes for L levels	$2^L - 1$
Min total nodes for height h	$h + 1$
Min height for n nodes	$\lceil \log_2(n+1) \rceil - 1$
Max height for n nodes	$n - 1$
Total levels from height h	$h + 1$
Height from total levels L	$L - 1$

Formula Practice Problems

Example 221: Q1: Max Nodes at a Level

In a binary tree (root at level 0), level 5 is completely full. How many nodes are present at level 5 alone?
Solution: Maximum nodes at level $l = 2^l$. So at level 5: $2^5 = 32$ nodes.

Example 222: Q2: Max Total Nodes for a Given Height

What is the maximum number of nodes possible in a binary tree of height 6?
Solution: Max total nodes $= 2^{h+1} - 1 = 2^{6+1} - 1 = 2^7 - 1 = 128 - 1 = 127$ nodes. This is achieved only by a perfect binary tree.

Example 223: Q3: Min Total Nodes for a Given Height

A binary tree has height 5. What is the minimum number of nodes it can have?
Solution: Min total nodes $= h + 1 = 5 + 1 = 6$ nodes. This occurs in a fully skewed tree where each internal node has exactly one child.

Example 224: Q4: Min Height for a Given Node Count

A complete binary tree has 511 nodes. What is its height?
Solution:

$$h_{\min} = \lceil \log_2(511+1) \rceil - 1 = \lceil \log_2(512) \rceil - 1 = \lceil 9 \rceil - 1 = 9 - 1 = 8$$

We can verify: $2^{8+1} - 1 = 2^9 - 1 = 511$. So 511 nodes form a perfect binary tree of height 8.

Example 225: Q5: Max Height for a Given Node Count

A binary tree has 12 nodes. What is the maximum possible height of this tree?
Solution: Max height $= n - 1 = 12 - 1 = 11$. This occurs in a fully skewed tree where every node has only one child, forming a chain of 12 nodes.

Example 226: Q6: Total Levels and Max Nodes

A binary tree has exactly 7 levels (root at level 0 through level 6). What is its height, and what is the maximum number of nodes it can contain?
Solution:

- Height $= L - 1 = 7 - 1 = 6$.
- Max nodes $= 2^L - 1 = 2^7 - 1 = 128 - 1 = 127$.

Example 227: Q7: Bounds on Height

A binary tree has $n = 100$ nodes. Which of the following statements about its height h is always true?
(A) $h = 99$ always (B) $6 \leq h \leq 99$ (C) $h \geq 100$ (D) $h \leq 6$ always
Solution:

- $h_{\min} = \lceil \log_2(101) \rceil - 1 = \lceil 6.66 \rceil - 1 = 7 - 1 = 6$.
- $h_{\max} = n - 1 = 99$.

So height h satisfies $6 \leq h \leq 99$. The correct answer is **(B)**.
Option A is wrong because $h = 99$ is only the maximum, not always true. Option D is wrong because h can be much larger than 6. Option C is impossible since h can never exceed $n - 1 = 99$.

5.3 Theorems and Structural Properties of Binary Trees

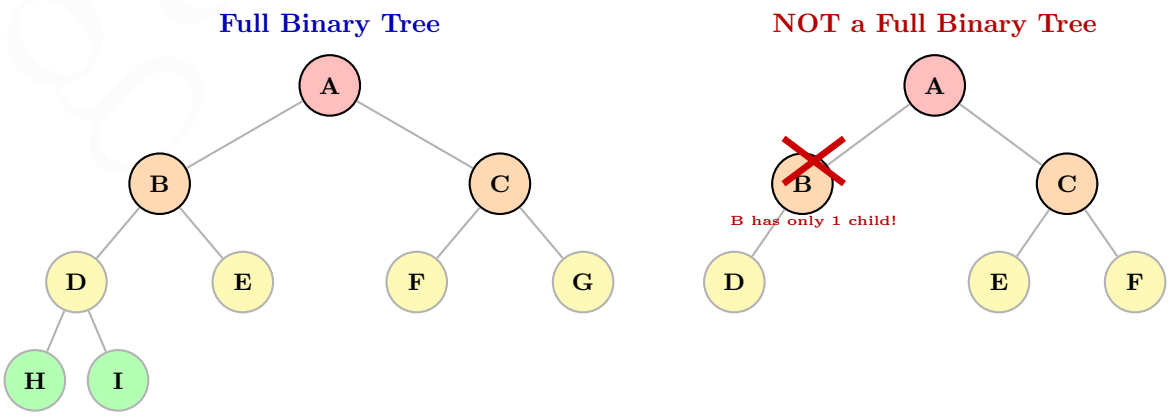
Concept: Notation Used Throughout This Section

- **Level (l):** Distance from the root node. Root is at level $l = 0$.
- **Height (h):** Maximum level of any leaf node in the tree. $h = L - 1$.
- **Levels Count (L):** Total number of levels present. $L = h + 1$.
- **n:** Total number of nodes in the tree.
- **I:** Number of internal nodes (nodes with at least one child).
- **n_{leaf}:** Number of leaf nodes (nodes with no children).
- Always: $n = I + n_{\text{leaf}}$.

5.3.1 Full Binary Tree

Concept: What is a Full Binary Tree

A **full binary tree** (also called a strictly binary tree or proper binary tree) is a binary tree where every node has either **exactly 0 children** or **exactly 2 children**. No node in a full binary tree has exactly 1 child.
Think of a tournament bracket: every match either ends (leaf) or produces two more matches (internal node). There is never a match with only one continuation.



Derivation of Full Binary Tree Formulas

Theorem: Leaf Count and Internal Node Count

In any full binary tree: $n_{\text{leaf}} = I + 1$.
Proof using handshaking on edges: In a binary tree with n nodes, there are $n-1$ edges total (tree property). Each internal node contributes exactly 2 edges going downward (since every internal node in a full binary tree has exactly 2 children). So total edges $= 2I$. But total edges $= n-1$. Therefore $2I = n-1$, which gives $n = 2I+1$. Since $n = I + n_{\text{leaf}}$, we get $I + n_{\text{leaf}} = 2I + 1$, so $n_{\text{leaf}} = I + 1$.

Example 228: Verifying $n_{\text{leaf}} = I + 1$

In the full binary tree above: nodes A, B, C, D are internal ($I = 4$). Nodes E, F, G, H, I are leaves ($n_{\text{leaf}} = 5$). Check: $n_{\text{leaf}} = I + 1 \Rightarrow 5 = 4 + 1$. Correct. Also $n = 2I + 1 = 9$ and $I = (n-1)/2 = 4$.

Formula: Nodes in terms of Levels (L)

For a full binary tree with L levels:

- **Minimum nodes:** When the tree is a skewed-but-full structure. The minimum has 1 internal node per level except the last, giving a chain of paired leaves. The minimum is $n_{\min} = 2L - 1$. (Each internal level adds 1 internal node and we have $L-1$ internal levels, plus L leaf nodes... actually derived as: $L-1$ internal nodes plus L leaves $= 2L - 1$.)
- **Maximum nodes:** When the tree is a perfect binary tree. $n_{\max} = 2^L - 1$.

$$2L - 1 \;\leq\; n \;\leq\; 2^L - 1$$

Formula: Nodes in terms of Height (h)

Since $L = h + 1$:

$$2(h+1) - 1 \;\leq\; n \;\leq\; 2^{h+1} - 1 \quad \Rightarrow \quad 2h + 1 \;\leq\; n \;\leq\; 2^{h+1} - 1$$

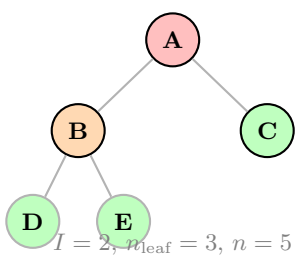
- Minimum total nodes $= 2h + 1$ (minimum full binary tree: a left-skewed full tree where each internal node has two children but only the rightmost path is long).
- Maximum total nodes $= 2^{h+1} - 1$ (perfect binary tree).

Formula: Internal Nodes in terms of L and h

- Since $n_{\text{leaf}} = I + 1$ and $n = 2I + 1$:
- When $n = 2L - 1$ (minimum): $I = L - 1$ (minimum internal nodes).
- When $n = 2^L - 1$ (maximum): $I = 2^{L-1} - 1$ (maximum internal nodes).

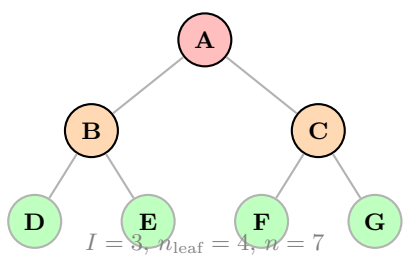
$$L - 1 \;\leq\; I \;\leq\; 2^{L-1} - 1 \qquad \text{or equivalently} \qquad h \;\leq\; I \;\leq\; 2^h - 1$$

Min Full BT: $h = 2$, $n = 2h + 1 = 5$



$I = 2$, $n_{\text{leaf}} = 3$, $n = 5$

Max Full BT: $h = 2$, $n = 2^{h+1} - 1 = 7$



$I = 3$, $n_{\text{leaf}} = 4$, $n = 7$

Full Binary Tree: Formula Summary

Property	Formula
Leaves in terms of internal nodes	$n_{\textbf{leaf}} = I + 1$
Total nodes in terms of I	$n = 2I + 1$
Internal nodes in terms of n	$I = \dfrac{n-1}{2}$ **(odd n only)**
Leaves in terms of n	$n_{\textbf{leaf}} = \dfrac{n+1}{2}$
n in terms of levels L	$2L - 1 \leq n \leq 2^L - 1$
n in terms of height h	$2h + 1 \leq n \leq 2^{h+1} - 1$
I in terms of levels L	$L - 1 \leq I \leq 2^{L-1} - 1$
I in terms of height h	$h \leq I \leq 2^h - 1$
Max leaves at height h	2^h

Example 229: Full Binary Tree: Worked Examples

Example 1: A full binary tree has 15 nodes. Find I and n_{leaf}.

$$I = \frac{n-1}{2} = \frac{15-1}{2} = 7, \qquad n_{\text{leaf}} = I + 1 = 8$$

Example 2: A full binary tree has height $h = 3$. What are the minimum and maximum possible total nodes?

$$n_{\min} = 2h + 1 = 7, \qquad n_{\max} = 2^{h+1} - 1 = 15$$

Example 3: A full binary tree has 9 leaf nodes. How many internal nodes does it have?

$$n_{\text{leaf}} = I + 1 \Rightarrow 9 = I + 1 \Rightarrow I = 8, \qquad n = 2I + 1 = 17$$

5.3.2 Perfect Binary Tree

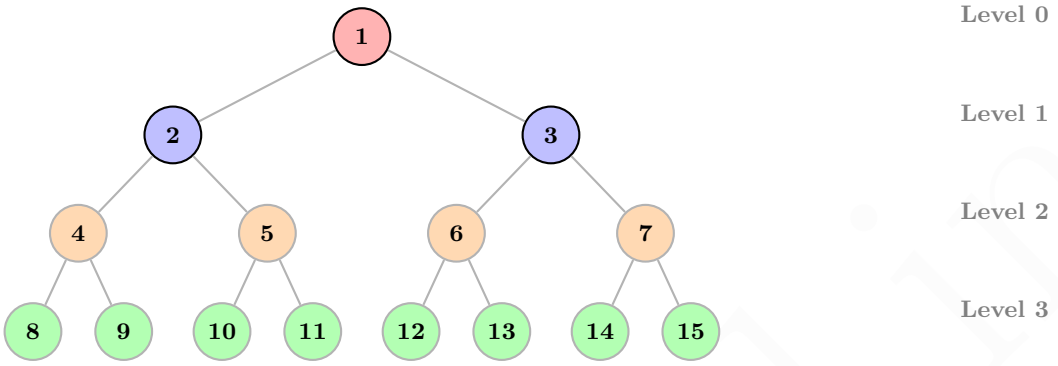
Concept: What is a Perfect Binary Tree

A **perfect binary tree** is a binary tree where:

- Every internal node has **exactly 2 children** (full condition), AND
- All leaf nodes are at the **same level** (complete and symmetric).

A perfect binary tree of height h has exactly $h+1$ levels, all levels are completely filled. It is the most regular and most densely packed binary tree shape.

Perfect BT: $h = 3$, $n = 15$, $n_{\text{leaf}} = 8$, $I = 7$



Derivation of Perfect Binary Tree Formulas

Theorem: Total Nodes in Perfect Binary Tree

In a perfect binary tree of height h (or L levels), every level is full. The number of nodes at level l is exactly 2^l. Summing all levels:

$$n = \sum_{l=0}^{h} 2^l = 2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$$

Or equivalently in terms of levels $L = h + 1$:

$$\boxed{n = 2^{h+1} - 1 = 2^L - 1}$$

Theorem: Leaf Nodes in Perfect Binary Tree

All leaf nodes are at the last level h. Number of nodes at level h is 2^h.

$$\boxed{n_{\text{leaf}} = 2^h = 2^{L-1}}$$

Since $n = 2^{h+1} - 1$ and $n_{\text{leaf}} = 2^h$:

$$n_{\text{leaf}} = \frac{n+1}{2}$$

Theorem: Internal Nodes in Perfect Binary Tree

$I = n - n_{\text{leaf}} = (2^{h+1} - 1) - 2^h = 2^h - 1$. Also from the full binary tree result ($n = 2I + 1$):

$$I = \frac{n-1}{2} = \frac{2^{h+1} - 2}{2} = 2^h - 1 = 2^{L-1} - 1$$

$$\boxed{I = 2^h - 1 = 2^{L-1} - 1}$$

Theorem: Height from n in Perfect Binary Tree

From $n = 2^{h+1} - 1$, solving for h:

$$n + 1 = 2^{h+1} \Rightarrow h + 1 = \log_2(n+1) \Rightarrow h = \log_2(n+1) - 1$$

This is only valid when $n + 1$ is a power of 2 (i.e., the tree is perfect).

Perfect Binary Tree: Formula Summary

Property	Formula
Total nodes	$n = 2^{h+1} - 1 = 2^L - 1$
Leaf nodes	$n_{\text{leaf}} = 2^h = 2^{L-1} = \frac{n+1}{2}$
Internal nodes	$I = 2^h - 1 = 2^{L-1} - 1 = \frac{n-1}{2}$
Leaves in terms of internal nodes	$n_{\text{leaf}} = I + 1$
Height from n	$h = \log_2(n+1) - 1$
Nodes at level l	**exactly** 2^l
Total levels	$L = h + 1 = \log_2(n+1)$
Max leaves at height h	2^h **(all at last level)**
Relation to full BT	**Every perfect BT is also a full BT**

Example 230: Perfect Binary Tree: Worked Examples

Example 1: A perfect binary tree has height $h = 4$. Find n, I, n_{leaf}.

$$n = 2^{4+1} - 1 = 31, \qquad n_{\text{leaf}} = 2^4 = 16, \qquad I = 2^4 - 1 = 15$$

Verify: $n_{\text{leaf}} = I + 1 \Rightarrow 16 = 15 + 1$ ✓, $n = I + n_{\text{leaf}} = 15 + 16 = 31$ ✓

Example 2: A perfect binary tree has 63 nodes. What is its height?

$$h = \log_2(63+1) - 1 = \log_2(64) - 1 = 6 - 1 = 5$$

Example 3: A perfect binary tree has 5 levels ($L = 5$). Find n, n_{leaf}, I.

$$n = 2^5 - 1 = 31, \qquad n_{\text{leaf}} = 2^{5-1} = 16, \qquad I = 2^{5-1} - 1 = 15$$

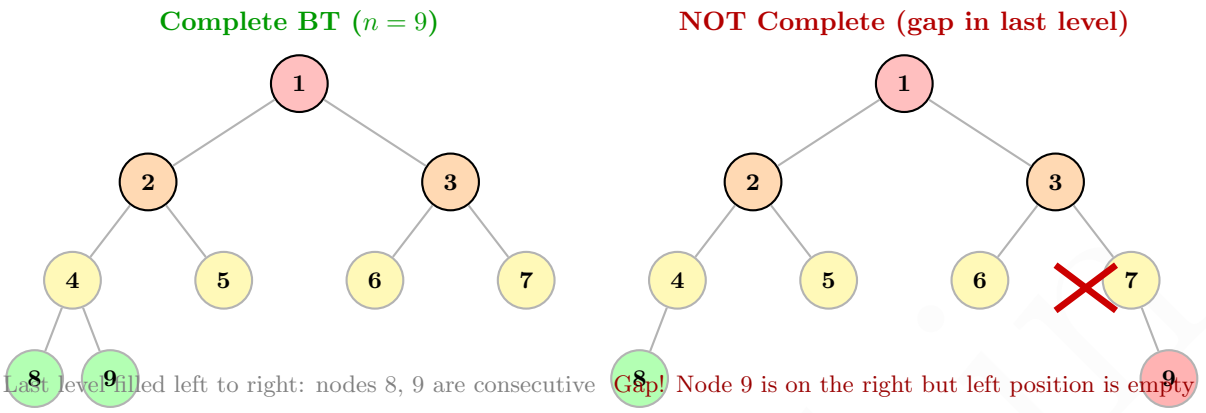
5.3.3 Complete Binary Tree

Concept: What is a Complete Binary Tree

A **complete binary tree** is a binary tree where:

- All levels except possibly the last level are **completely filled**.
- The last level has nodes filled from **left to right** with no gaps in between.

A perfect binary tree is always also a complete binary tree. But a complete binary tree is not always perfect (the last level may be partially filled). Complete binary trees are important because they are efficiently stored in arrays using index formulas.



Derivation of Complete Binary Tree Formulas

Theorem: Range of Total Nodes for Height h

A complete binary tree of height h has all levels 0 to $h-1$ completely filled ($2^h - 1$ nodes in them), and level h has at least 1 node and at most 2^h nodes. Therefore:

$$\underbrace{(2^h - 1) + 1}_{= 2^h} \leq n \leq \underbrace{(2^h - 1) + 2^h}_{= 2^{h+1}-1}$$

$$\boxed{2^h \leq n \leq 2^{h+1} - 1}$$

Equivalently in terms of levels L:

$$\boxed{2^{L-1} \leq n \leq 2^L - 1}$$

Theorem: Height from n in Complete Binary Tree

From $2^h \leq n \leq 2^{h+1} - 1$, taking $\log_2$ of the lower bound:

$$h = \lfloor \log_2 n \rfloor$$

This is the exact height formula for any complete binary tree with n nodes.

Theorem: Leaf Nodes and Internal Nodes in Complete Binary Tree

For a complete binary tree with n nodes:

$$\boxed{n_{\text{leaf}} = \left\lceil \frac{n}{2} \right\rceil}$$

$$\boxed{I = \left\lfloor \frac{n}{2} \right\rfloor}$$

Derivation: Using the array representation (1-indexed), nodes $\lfloor n/2 \rfloor + 1$ through n are leaves (they have no children since their left child would be at index $> n$). That is $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ leaves. The remaining $\lfloor n/2 \rfloor$ nodes are internal.

Checking the formula:

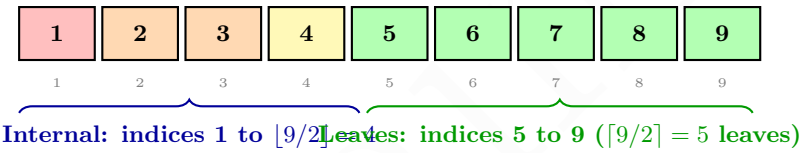
- If $n = 9$: $n_{\text{leaf}} = \lceil 9/2 \rceil = 5$, $I = \lfloor 9/2 \rfloor = 4$.
- If $n = 10$: $n_{\text{leaf}} = \lceil 10/2 \rceil = 5$, $I = \lfloor 10/2 \rfloor = 5$.

Example 231: Why $n_{\text{leaf}} = \lceil n/2 \rceil$ in Complete Binary Tree

In a complete binary tree stored in a 1-indexed array of size n:

- Node at index i has left child at $2i$ and right child at $2i + 1$.
- A node at index i is a leaf if $2i > n$ (its left child doesn't exist).
- Largest non-leaf index: $\lfloor n/2 \rfloor$.
- Indices $\lfloor n/2 \rfloor + 1$ to n are all leaves: that is $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ leaf nodes.

Complete BT ($n = 9$): Array Representation



Formula: Internal Nodes in Terms of Levels

All levels 0 to $L - 2$ are completely full in a complete binary tree. So the minimum number of internal nodes includes all of level $L - 2$ and above:

- Minimum I: when the last level has only 1 node (leftmost), then $I = (2^{L-1} - 1)$ from filled levels... actually the range comes from the range of n:
- Since $I = \lfloor n/2 \rfloor$ and $2^{L-1} \leq n \leq 2^L - 1$:

$$\left\lfloor \frac{2^{L-1}}{2} \right\rfloor \leq I \leq \left\lfloor \frac{2^L - 1}{2} \right\rfloor \quad \Rightarrow \quad 2^{L-2} \leq I \leq 2^{L-1} - 1$$

In terms of height h:

$$\boxed{2^{h-1} \leq I \leq 2^h - 1}$$

Complete Binary Tree: Formula Summary

Property	Formula
Total nodes range (height h)	$2^h \le n \le 2^{h+1} - 1$
Total nodes range (L levels)	$2^{L-1} \le n \le 2^L - 1$
Height from n	$h = \lfloor \log_2 n \rfloor$
Leaf nodes	$n_{\textbf{leaf}} = \lceil n/2 \rceil$
Internal nodes	$I = \lfloor n/2 \rfloor$
Internal nodes range (height h)	$2^{h-1} \le I \le 2^h - 1$
Internal nodes range (L levels)	$2^{L-2} \le I \le 2^{L-1} - 1$
Max leaves at height h	2^h **(when tree is perfect)**
Array: parent of node at index i	$\lfloor i/2 \rfloor$ **(1-indexed)**
Array: children of node at index i	**Left** $= 2i$, **Right** $= 2i + 1$

Example 232: Complete Binary Tree: Worked Examples

Example 1: A complete binary tree has 20 nodes. Find h, n_{leaf}, I.

$$h = \lfloor \log_2 20 \rfloor = \lfloor 4.32 \rfloor = 4$$

$$n_{\text{leaf}} = \lceil 20/2 \rceil = 10, \qquad I = \lfloor 20/2 \rfloor = 10$$

Verify: $n_{\text{leaf}} + I = 10 + 10 = 20 = n$ ✓

Example 2: A complete binary tree has height $h = 3$. What are the minimum and maximum nodes?

$$n_{\min} = 2^3 = 8, \qquad n_{\max} = 2^{3+1} - 1 = 15$$

Example 3: A complete binary tree has 100 nodes. Find n_{leaf} and I.

$$n_{\text{leaf}} = \lceil 100/2 \rceil = 50, \qquad I = \lfloor 100/2 \rfloor = 50$$

Example 4: A complete binary tree has 7 nodes (hence perfect). Find n_{leaf} using the complete tree formula.

$$n_{\text{leaf}} = \lceil 7/2 \rceil = 4 = 2^2 = 2^h \text{ (since } h = \lfloor \log_2 7 \rfloor = 2)$$

Both complete and perfect formulas agree here. ✓

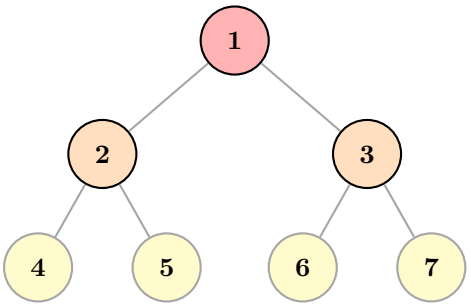
Comparison: Full vs Perfect vs Complete Binary Tree

Property / Formula	Full BT	Perfect BT	Complete BT
n_{leaf} in terms of I	$n_{\textbf{leaf}} = I + 1$	$n_{\textbf{leaf}} = I + 1$	$n_{\textbf{leaf}} = \lceil n/2 \rceil$
n in terms of I	$n = 2I + 1$	$n = 2I + 1$	**Level-dependent**
I in terms of n	$I = \frac{n-1}{2}$	$I = \frac{n-1}{2}$	$I = \lfloor n/2 \rfloor$
n_{leaf} in terms of n	$\frac{n+1}{2}$	$\frac{n+1}{2}$	$\lceil n/2 \rceil$
n in terms of L	$2L - 1 \le n \le 2^L - 1$	$n = 2^L - 1$	$2^{L-1} \le n \le 2^L - 1$
I in terms of L	$L - 1 \le I \le 2^{L-1} - 1$	$I = 2^{L-1} - 1$	$2^{L-2} \le I \le 2^{L-1} - 1$
Max leaves at L levels	2^{L-1}	2^{L-1}	2^{L-1}
n in terms of h	$2h + 1 \le n \le 2^{h+1} - 1$	$n = 2^{h+1} - 1$	$2^h \le n \le 2^{h+1} - 1$
I in terms of h	$h \le I \le 2^h - 1$	$I = 2^h - 1$	$2^{h-1} \le I \le 2^h - 1$
Max leaves at h	2^h	2^h **(all leaves at level h)**	2^h

5.4 Binary Tree Traversals

Concept: What is Traversal

Traversal means visiting every node of the tree exactly once, in some specific order. Unlike arrays where we always go from index 0 to $n-1$, trees can be traversed in many different orders. The four common traversals are Inorder, Preorder, Postorder, and Level-order. Each gives a different sequence of nodes, and each is useful for different purposes.

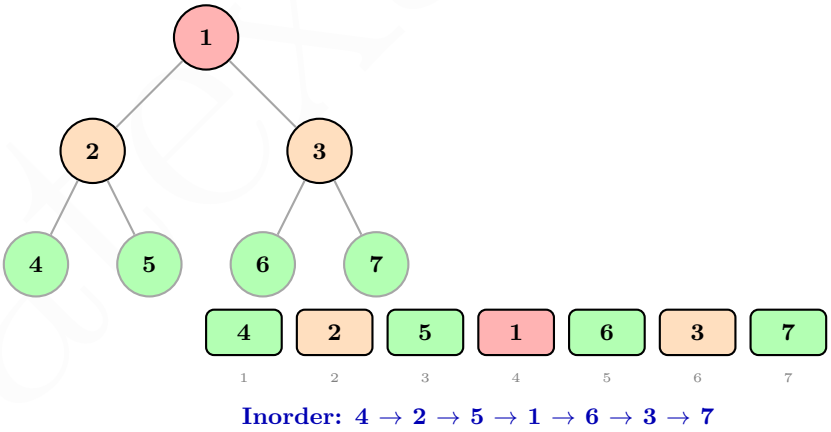


Inorder: 4, 2, 5, 1, 6, 3, 7
Preorder: 1, 2, 4, 5, 3, 6, 7
Postorder: 4, 5, 2, 6, 7, 3, 1
Level-order: 1, 2, 3, 4, 5, 6, 7

5.4.1 Inorder Traversal (Left, Root, Right)

Concept: Inorder Traversal

In inorder traversal, we visit nodes in the order: Left subtree, then Root, then Right subtree, recursively. Think of it as reading a book: go all the way to the leftmost page first, read it, come back, read the current chapter, then go to the right. For a BST, inorder traversal gives nodes in sorted order, which is a very important property.



Inorder: 4 → 2 → 5 → 1 → 6 → 3 → 7

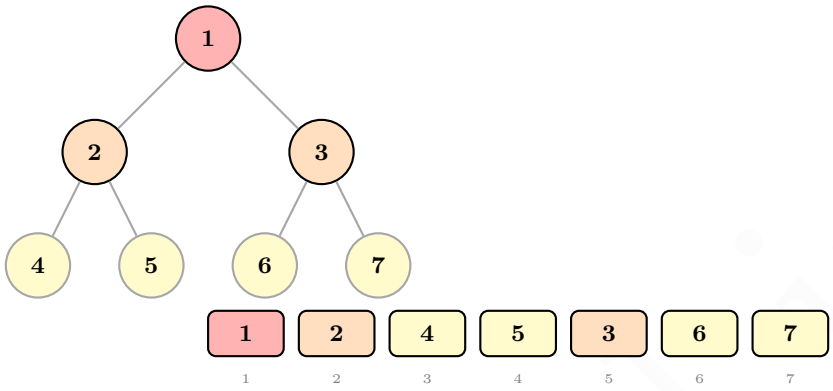
Algorithm: Inorder(node)

```
1: if node ≠ NULL then
2:     Inorder(node.left)
3:     print node.data
4:     Inorder(node.right)
5: end if
```

5.4.2 Preorder Traversal (Root, Left, Right)

Concept: Preorder Traversal

In preorder traversal, we visit nodes in the order: Root first, then Left subtree, then Right subtree. Think of it as copying a directory structure: you first note down the current folder name, then copy everything inside the left subfolder, then copy everything inside the right subfolder. Preorder is used to create a copy of the tree.



Preorder: 1 → 2 → 4 → 5 → 3 → 6 → 7

5.4.3 Postorder Traversal (Left, Right, Root)

Concept: Postorder Traversal

In postorder traversal, we visit nodes in the order: Left subtree, then Right subtree, then Root last. Think of it as cleaning up: you clean all the stuff inside a folder first, and only then delete the folder itself. Postorder is used to delete a tree, or to evaluate expression trees where operators come after operands.

Algorithm: Postorder(node)

```
1: if node ≠ NULL then
2:     Postorder(node.left)
3:     Postorder(node.right)
4:     print node.data
5: end if
```

5.4.4 Reconstructing a Tree from Two Traversals

Concept: Tree Reconstruction

Given two traversals, we can reconstruct the unique binary tree, but only if one of them is Inorder. This is because Inorder tells us which nodes are on the left and which are on the right of each root. The other traversal (Preorder or Postorder) tells us which node is the root.

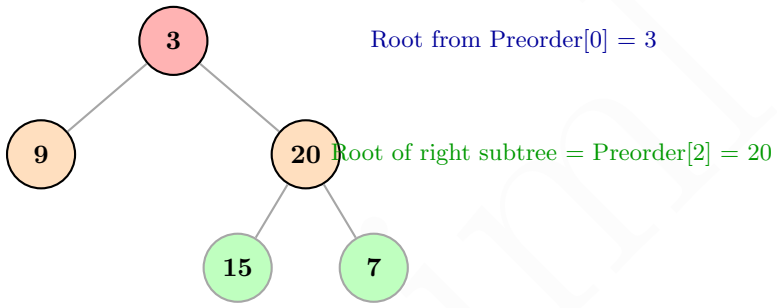
- **Inorder + Preorder** → can reconstruct unique tree.
- **Inorder + Postorder** → can reconstruct unique tree.
- **Preorder + Postorder alone** → cannot always reconstruct unique tree.

Example 233: Step by Step: Reconstruct from Inorder + Preorder (Example 1)

Given:

- Preorder: [3, 9, 20, 15, 7]
- Inorder: [9, 3, 15, 20, 7]

Step 1: First element of Preorder is always the root. So root = 3.
Step 2: Find root (3) in Inorder. Everything to its left in Inorder ([9]) is in the left subtree. Everything to its right ([15, 20, 7]) is in the right subtree.
Step 3: Left subtree has 1 node (9). Right subtree has 3 nodes (15, 20, 7). Take next 1 element from Preorder for left subtree: [9]. Take next 3 for right subtree: [20, 15, 7].
Step 4: Recurse. For right subtree: Preorder [20, 15, 7], Inorder [15, 20, 7]. Root = 20. Left of 20 in Inorder = [15]. Right = [7].
Final Tree:

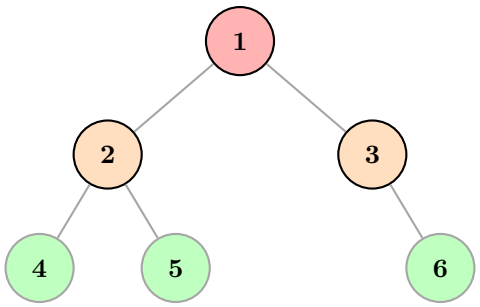


Example 234: Step by Step: Reconstruct from Inorder + Preorder (Example 2)

Given:

- Preorder: [1, 2, 4, 5, 3, 6]
- Inorder: [4, 2, 5, 1, 3, 6]

Step 1: Preorder[0] = 1 is the main root.
Step 2: In Inorder, elements left of 1 are [4, 2, 5] (Left subtree), and right of 1 are [3, 6] (Right subtree).
Step 3: For Left subtree: Preorder is [2, 4, 5] and Inorder is [4, 2, 5]. Root = 2. Left of 2 is [4], right of 2 is [5].
Step 4: For Right subtree: Preorder is [3, 6] and Inorder is [3, 6]. Root = 3. Left of 3 is empty, right of 3 is [6].
Final Tree:

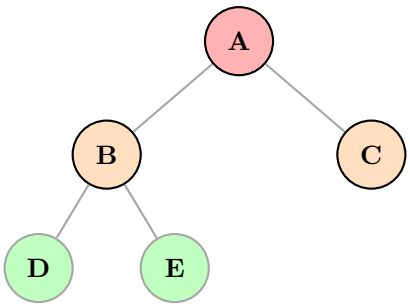


Example 235: Step by Step: Reconstruct from Inorder + Preorder (Example 3)

Given:

- Preorder: [A, B, D, E, C]
- Inorder: [D, B, E, A, C]

Step 1: Preorder[0] = A is the root.
Step 2: Inorder splits into Left: [D, B, E] and Right: [C].
Step 3: For Left subtree: Preorder is [B, D, E], Inorder is [D, B, E]. Root = B. Left of B is [D], right of B is [E].
Step 4: For Right subtree: Only node [C] remains as the right child of A.
Final Tree:

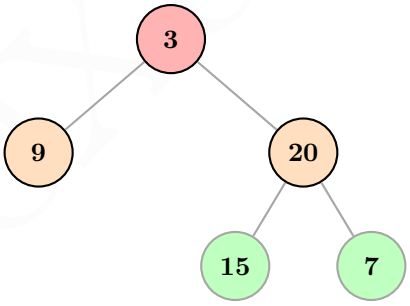


Example 236: Step by Step: Reconstruct from Inorder + Postorder (Example 1)

Given:

- Inorder: [9, 3, 15, 20, 7]
- Postorder: [9, 15, 7, 20, 3]

Step 1: Last element of Postorder is always the root. So root = 3.
Step 2: Locate 3 in Inorder. Left subtree nodes = [9], Right subtree nodes = [15, 20, 7].
Step 3: Process Postorder from right to left. Left subtree gets Postorder [9], Right subtree gets Postorder [15, 7, 20].
Step 4: For Right subtree: Last element in Postorder [15, 7, 20] is 20. Root = 20. Left of 20 in Inorder = [15], Right = [7].
Final Tree:

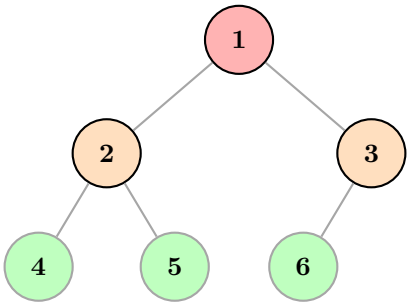


Example 237: Step by Step: Reconstruct from Inorder + Postorder (Example 2)

Given:

- Inorder: [4, 2, 5, 1, 6, 3]
- Postorder: [4, 5, 2, 6, 3, 1]

Step 1: Postorder[-1] = 1 is the main root.
Step 2: Inorder splits into Left: [4, 2, 5] and Right: [6, 3].
Step 3: For Right subtree ([6, 3]): Postorder slice is [6, 3]. Last element = 3 (Right child of 1). In Inorder, 6 is to the left of 3.
Step 4: For Left subtree ([4, 2, 5]): Postorder slice is [4, 5, 2]. Last element = 2 (Left child of 1). Left of 2 is [4], Right of 2 is [5].
Final Tree:

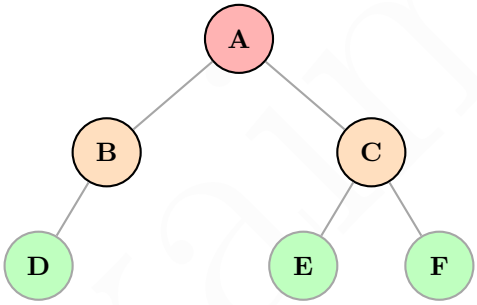


Example 238: Step by Step: Reconstruct from Inorder + Postorder (Example 3)

Given:

- Inorder: [D, B, A, E, C, F]
- Postorder: [D, B, E, F, C, A]

Step 1: Postorder[-1] = A is the root.
Step 2: Inorder splits into Left: [D, B] and Right: [E, C, F].
Step 3: For Left subtree: Postorder is [D, B]. Root = B. Inorder shows D is on the left of B.
Step 4: For Right subtree: Postorder is [E, F, C]. Root = C. Inorder shows E is to the left of C, and F is to the right of C.
Final Tree:



Concept: Why Preorder + Postorder Cannot Guarantee Uniqueness

When given Preorder and Postorder traversals without Inorder, we can identify the main root (first element of Preorder, last element of Postorder). However, whenever a node has **only one child**, it is impossible to determine whether that child is a **left child** or a **right child**.
General Attempt Method:

- Preorder[0] (or Postorder[-1]) is the root.
- Preorder[1] is assumed to be the root of the left subtree.
- We locate Preorder[1] in Postorder to find the boundary of the left subtree.
- **Ambiguity:** If a parent node has only one child, that child appears immediately after the parent in Preorder and right before it in Postorder, giving no boundary information to distinguish left from right.

Formula for Number of Possible Trees:

$$\text{Number of Trees} = 2^k$$

where k is the number of internal nodes with **exactly one child**. Since each single-child node provides 2 independent choices (Left or Right), k single-child nodes yield 2^k possible tree topologies.
Example: Given Preorder: [1, 2, 3] and Postorder: [3, 2, 1].

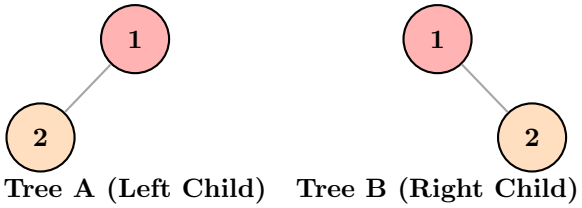
- Node 1 has only child 2 $\rightarrow$ Choice of Left or Right.
- Node 2 has only child 3 $\rightarrow$ Choice of Left or Right.
- Here $k = 2$ single-child nodes, so total trees $= 2^2 = 4$ distinct topologies.

Example 239: Ambiguity Case 1: Two-Node Tree

Given:

- Preorder: [1, 2]
- Postorder: [2, 1]

Both structures below generate identical Preorder [1, 2] and Postorder [2, 1] sequences:



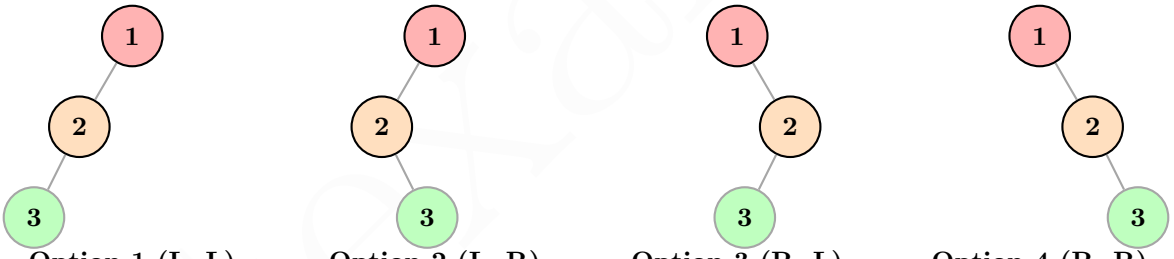
Tree A (Left Child) **Tree B (Right Child)**

Example 240: Ambiguity Case 2: Three-Node Skewed vs Mixed Trees

Given:

- Preorder: [1, 2, 3]
- Postorder: [3, 2, 1]

Multiple distinct tree topologies satisfy these exact sequences:



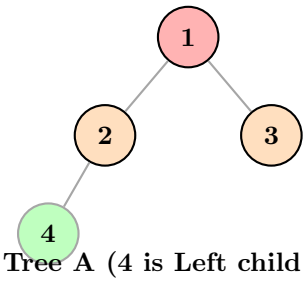
Option 1 (L, L) **Option 2 (L, R)** **Option 3 (R, L)** **Option 4 (R, R)**

Example 241: Ambiguity Case 3: Four-Node Internal Single-Child Tree

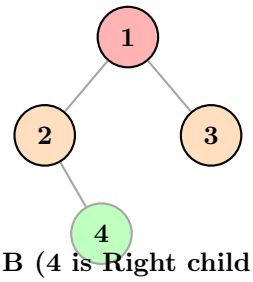
Given:

- Preorder: [1, 2, 4, 3]
- Postorder: [4, 2, 3, 1]

Root is 1. Left subtree contains 2 and 4; Right subtree contains 3. However, whether node 4 is a left child or right child of node 2 cannot be determined:



Tree A (4 is Left child of 2)



Tree B (4 is Right child of 2)

Concept: Reconstruction Under Special Binary Tree Constraints

If additional structural constraints are known, Preorder + Postorder **can** uniquely reconstruct the tree:

- **Full / Strict Binary Tree ($k = 0$):** Since every internal node has 2 children, no single-child ambiguity exists.

$$\text{Number of Trees} = 2^0 = 1 \quad \text{(Unique)}$$

- **Perfect Binary Tree:** A restricted subset of Full Binary Trees.

$$\text{Number of Trees} = 1 \quad \text{(Unique)}$$

- **Complete Binary Tree:** Structural position rules force any degree-1 node's single child to be on the left.

$$\text{Number of Trees} = 1 \quad \text{(Unique)}$$

5.5 Binary Search Tree (BST)

Concept: BST Property

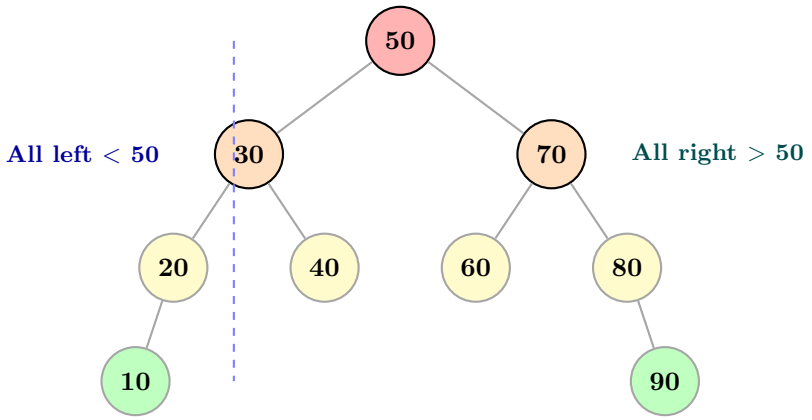
A Binary Search Tree is a binary tree with a very useful ordering property. For every node in the tree:

- All values in the **left subtree** are **strictly less than** the node's value.
- All values in the **right subtree** are **strictly greater than** the node's value.

This property holds recursively for every node, not just the root. This ordering makes searching, insertion, and deletion much faster than in a plain binary tree, in the average case.

Example 242: Analogy: Dictionary Search

Think of searching for a word in a dictionary. You open to the middle page. If your word comes before the middle word alphabetically, you look in the left half. Otherwise, you look in the right half. A BST works the same way. At each node, you decide to go left (smaller) or right (larger), eliminating half the remaining nodes each time.



Inorder gives sorted order: 10, 20, 30, 40, 50, 60, 70, 80, 90

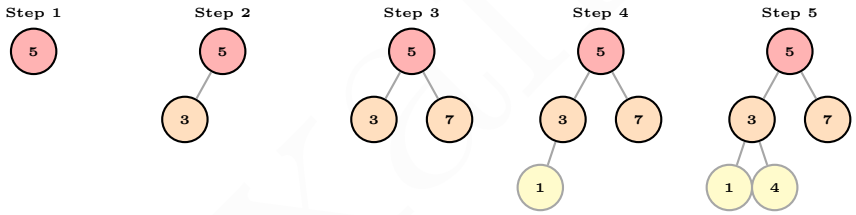
5.5.1 Construct a BST from a Given Insertion Sequence

Concept: BST from Insertion Sequence

To construct a BST from a given sequence of keys, we insert them one by one in the given order. The first key becomes the root. Each subsequent key is inserted using the standard BST insert procedure. The shape of the final BST depends entirely on the order of insertion, even if the set of keys is the same.

Example 243: Step by Step: Insert [5, 3, 7, 1, 4, 6, 8]

We insert each number one by one.



Continue inserting 6 and 8 to complete the BST

Python implementation: bst-from-sequence.py

```
def build_bst(sequence):
    root = None
    for val in sequence:
        root = bst_insert(root, val)
    return root


def print_tree(root, level=0, prefix="Root: "):
    if root:
        print(" " * (level * 4) + prefix + str(root.data))
        if root.left or root.right:
            if root.left:
                print_tree(root.left, level + 1, "L: ")
            else:
                print(" " * ((level+1)*4) + "L: None")
            if root.right:
                print_tree(root.right, level + 1, "R: ")
            else:
                print(" " * ((level+1)*4) + "R: None")


# --- Driver Code ---
seq  = [5, 3, 7, 1, 4, 6, 8]
root = build_bst(seq)
```

```
print("BST built from:", seq)
print_tree(root)
print("\nInorder (sorted):", end=" ")
inorder(root)
# Output:
# Root: 5
#     L: 3
#         L: 1
#         R: 4
#     R: 7
#         L: 6
#         R: 8
# Inorder (sorted): 1 3 4 5 6 7 8
```

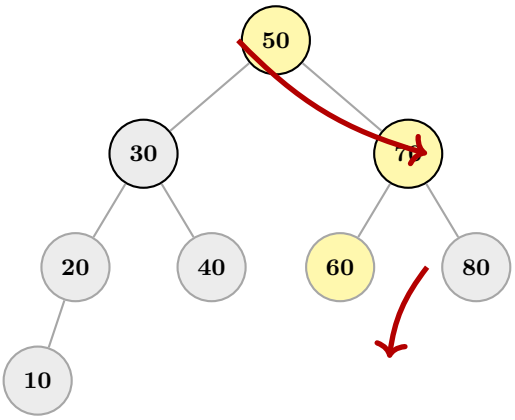
BST Operations Summary

Operation	Best Case (Balanced)	Worst Case (Skewed)	Average
Search	$O(\log n)$	$O(n)$	$O(\log n)$
Insert	$O(\log n)$	$O(n)$	$O(\log n)$
Delete	$O(\log n)$	$O(n)$	$O(\log n)$
Inorder Traversal	$O(n)$	$O(n)$	$O(n)$
Find Min/Max	$O(\log n)$	$O(n)$	$O(\log n)$
Inorder Successor	$O(\log n)$	$O(n)$	$O(\log n)$
Build BST	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

5.5.2 Search in BST

Concept: BST Search in $O(h)$

To search for a value in a BST, we start at the root. If the value equals the root, we found it. If the value is less, we go to the left subtree. If more, we go right. We keep doing this until we find the node or reach NULL. At each step, we eliminate one entire subtree from consideration. The time taken is $O(h)$, where h is the height of the tree.



Search 60:

$60 < 50$? No. Go right.

$60 < 70$? Yes. Go left.

$60 = 60$. Found!

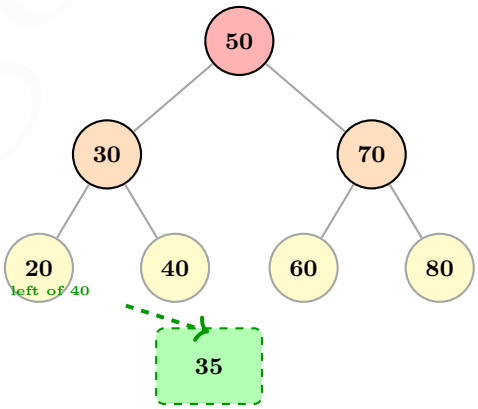
Algorithm: BST-Search(node, key)

```
1: if node = NULL or node.data = key then
2:     return node
3: end if
4: if key < node.data then
5:     return BST-Search(node.left, key)
6: else
7:     return BST-Search(node.right, key)
8: end if
```

5.5.3 Insert in BST

Concept: BST Insertion in $O(h)$

Inserting a new key follows the same path as search. We search for the key. When we reach NULL (a position where the key would be if it existed), we insert the new node there. We never change existing nodes. We only attach a new leaf. This also takes $O(h)$ time.



Insert 35

$35 < 50 \rightarrow$ go left

$35 > 30 \rightarrow$ go right

$35 < 40 \rightarrow$ go left

NULL found $\rightarrow$ insert here

Algorithm: BST-Insert(node, key)

```
if node = NULL then
    return Node(key)
else if key < node.data then
    node.left ← BST-Insert(node.left, key)
else if key > node.data then
    node.right ← BST-Insert(node.right, key)
end if
return node
```

5.5.4 Inorder Successor and Predecessor

Concept: Inorder Successor

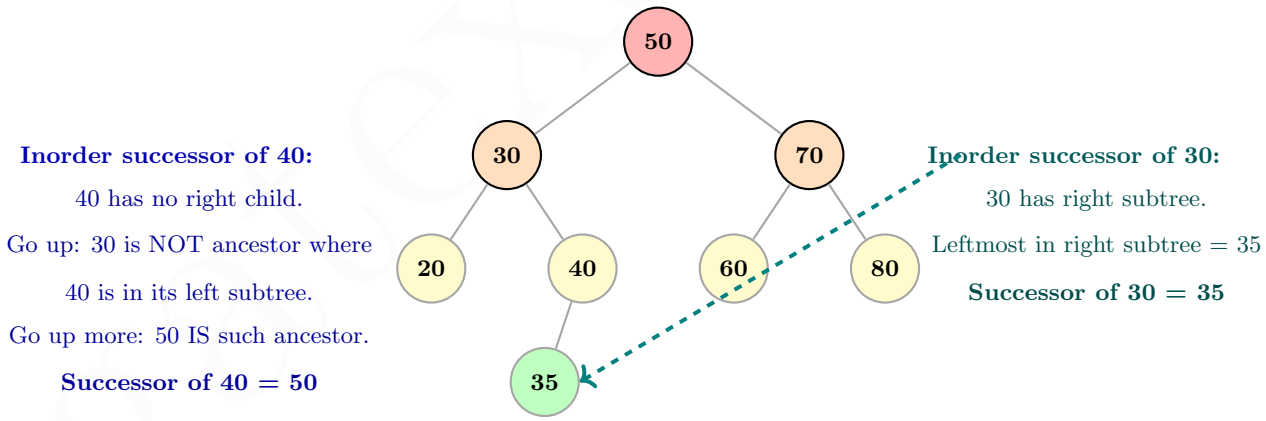
The inorder successor of a node is the node that comes immediately after it in the inorder traversal. In BST terms, it is the smallest value that is still greater than the node's value.

- **Case 1: Node has a right subtree.** The inorder successor is the **leftmost node** in the right subtree. We go right once, then keep going left until we cannot go further.
- **Case 2: Node has no right subtree.** The inorder successor is the **lowest ancestor** for which this node lies in the ancestor's left subtree. We go up the tree until we take a left turn.

Concept: Inorder Predecessor

The inorder predecessor of a node is the node that comes immediately before it in inorder traversal. It is the largest value still smaller than the node's value.

- **Case 1: Node has a left subtree.** The inorder predecessor is the **rightmost node** in the left subtree. We go left once, then keep going right until we cannot go further.
- **Case 2: Node has no left subtree.** The inorder predecessor is the **lowest ancestor** for which this node lies in the ancestor's right subtree. We go up the tree until we take a right turn.



Example 244: More Examples on the Same BST

Using the BST with inorder sequence: 20, 30, 35, 40, 50, 60, 70, 80.
Inorder Successors:

- Successor of 20: 20 has no right child. Go up. 30 is an ancestor and 20 is in 30's left subtree. So successor of 20 = **30**.
- Successor of 50: 50 is root and has right subtree. Leftmost in right subtree (rooted at 70): go right to 70, then left to 60. No more left. Successor of 50 = **60**.
- Successor of 80: 80 has no right child. Go up. 80 is in 70's right subtree. Go up more. 70 is in 50's right

subtree. Go up more. 50 is root, no more ancestors. Successor of 80 = **NULL** (80 is the largest).

Inorder Predecessors:

- Predecessor of 50: 50 has left subtree. Rightmost in left subtree (rooted at 30): go left to 30, then right to 40, then right would be... 40 has no right child. Predecessor of 50 = **40**.
- Predecessor of 20: 20 has no left child. Go up. 20 is in 30's left subtree, so 30 is NOT the answer (we need a right-turn ancestor). Go up more. 30 is in 50's left subtree. Still no. 50 has no parent. Predecessor of 20 = **NULL** (20 is the smallest).
- Predecessor of 60: 60 has no left child. Go up. 60 is in 70's left subtree, so 70 is NOT the answer. Go up. 70 is in 50's right subtree. So predecessor of 60 = **50**.

Algorithm: BST-Inorder-Successor(root, target_val)

```
curr ← root, successor ← NULL
while curr ≠ NULL do
    if curr.data > target_val then
        successor ← curr                          ▷ curr could be successor
        curr ← curr.left                          ▷ try to find smaller valid successor
    else if curr.data < target_val then
        curr ← curr.right
    else                                          ▷ found the target node
        if curr.right ≠ NULL then
            temp ← curr.right
            while temp.left ≠ NULL do             ▷ go to leftmost of right subtree
                temp ← temp.left
            end while
            return temp
        end if
        break
    end if
end while
return successor
```

Algorithm: BST-Inorder-Predecessor(root, target_val)

```
curr ← root, predecessor ← NULL
while curr ≠ NULL do
    if curr.data < target_val then
        predecessor ← curr                        ▷ curr could be predecessor
        curr ← curr.right                         ▷ try to find larger valid predecessor
    else if curr.data > target_val then
        curr ← curr.left
    else                                          ▷ found the target node
        if curr.left ≠ NULL then
            temp ← curr.left
            while temp.right ≠ NULL do            ▷ go to rightmost of left subtree
                temp ← temp.right
            end while
            return temp
        end if
        break
    end if
end while
return predecessor
```

Python implementation: bst-successor-predecessor.py

```python
class Node:
    def __init__(self, data):
```

```
        self.data  = data
        self.left  = None
        self.right = None


def inorder_successor(root, target_val):
    """
    Returns the node with the smallest value
    greater than target_val in the BST.
    Returns None if no successor exists.
    """
    curr      = root
    successor = None

    while curr:
        if curr.data > target_val:
            successor = curr        # curr is a valid candidate
            curr      = curr.left   # try to find smaller valid one
        elif curr.data < target_val:
            curr = curr.right
        else:                       # found the target node
            if curr.right:
                # Case 1: right subtree exists
                # successor = leftmost node in right subtree
                temp = curr.right
                while temp.left:
                    temp = temp.left
                return temp
            break                   # Case 2: use the stored successor

    return successor


def inorder_predecessor(root, target_val):
    """
    Returns the node with the largest value
    less than target_val in the BST.
    Returns None if no predecessor exists.
    """
    curr        = root
    predecessor = None

    while curr:
        if curr.data < target_val:
            predecessor = curr      # curr is a valid candidate
            curr        = curr.right  # try to find larger valid one
        elif curr.data > target_val:
            curr = curr.left
        else:                       # found the target node
            if curr.left:
                # Case 1: left subtree exists
                # predecessor = rightmost node in left subtree
                temp = curr.left
                while temp.right:
                    temp = temp.right
                return temp
            break                   # Case 2: use the stored predecessor

    return predecessor


# ============================================================
# Helper: build BST from insertion sequence
# ============================================================
def insert(root, val):
    if root is None:
        return Node(val)
    if val < root.data:
```

```
        root.left  = insert(root.left, val)
    elif val > root.data:
        root.right = insert(root.right, val)
    return root


# --- Driver Code ---
# Build BST: 50, 30, 70, 20, 40, 60, 80, 35
root = None
for val in [50, 30, 70, 20, 40, 60, 80, 35]:
    root = insert(root, val)

# Inorder: 20, 30, 35, 40, 50, 60, 70, 80

# Successors
for val in [20, 30, 40, 50, 80]:
    s = inorder_successor(root, val)
    print(f"Successor of {val}: {s.data if s else 'NULL'}")

print()

# Predecessors
for val in [20, 30, 50, 60, 80]:
    p = inorder_predecessor(root, val)
    print(f"Predecessor of {val}: {p.data if p else 'NULL'}")

# Output:
# Successor of 20: 30
# Successor of 30: 35
# Successor of 40: 50
# Successor of 50: 60
# Successor of 80: NULL
#
# Predecessor of 20: NULL
# Predecessor of 30: 20
# Predecessor of 50: 40
# Predecessor of 60: 50
# Predecessor of 80: 70
```

Concept: Time Complexity Analysis

Both inorder successor and inorder predecessor algorithms follow a single root-to-node path in the BST, then at most one more root-to-leaf path inside a subtree.
Time Complexity: $O(h)$ where h is the height of the BST.

- **Case 1 (subtree exists):** We first search for the target node going down the BST: $O(h)$ steps. Then we find the leftmost (or rightmost) node in the subtree: at most $O(h)$ additional steps. Total: $O(h)$.
- **Case 2 (no subtree):** We traverse from root to the target node while recording ancestor candidates. This is a single top-down pass: $O(h)$ steps.

Space Complexity: $O(1)$ for the iterative version (only a few pointer variables used). $O(h)$ for a recursive version due to the call stack.
In terms of n (number of nodes):

- Best case (balanced BST): $h = O(\log n)$, so time $= O(\log n)$.
- Worst case (skewed BST): $h = O(n)$, so time $= O(n)$.

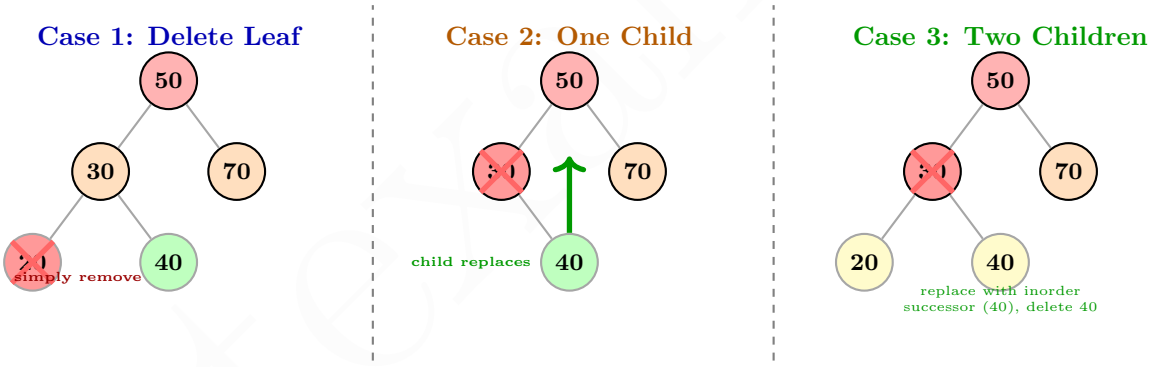
Inorder Successor and Predecessor: Summary

Property	Inorder Successor	Inorder Predecessor
Definition	Next larger value in BST	Next smaller value in BST
If subtree exists	Leftmost of right subtree	Rightmost of left subtree
If no subtree	Lowest ancestor: node is in left subtree	Lowest ancestor: node is in right subtree
Time Complexity	$O(h)$	$O(h)$
Space Complexity (iterative)	$O(1)$	$O(1)$
Balanced BST case	$O(\log n)$	$O(\log n)$
Skewed BST case	$O(n)$	$O(n)$
Boundary case	Largest node has no successor (NULL)	Smallest node has no predecessor (NULL)

5.5.5 Delete in BST: Three Cases

Concept: BST Deletion — Three Cases

Deleting a node from a BST is the trickiest operation. There are three cases depending on how many children the node to be deleted has.
Case 1: Leaf node (0 children). Simply remove it. Set the parent's pointer to NULL.
Case 2: One child. Replace the node with its only child. Connect the parent directly to the child.
Case 3: Two children. Find the inorder successor (smallest node in the right subtree). Copy the inorder successor's value into the current node. Then delete the inorder successor (which has at most one child, so it falls into Case 1 or Case 2).



Algorithm: BST-Delete(node, key)

```
1: if node = NULL then return NULL
2: end if
3: if key < node.data then
4:     node.left ← BST-Delete(node.left, key)
5: else if key > node.data then
6:     node.right ← BST-Delete(node.right, key)
7: else                                          ▷ Found the node to delete
8:     if node.left = NULL then return node.right
9:     end if
10:    if node.right = NULL then return node.left
11:    end if
12:    succ ← Find-Min(node.right)                ▷ inorder successor
13:    node.data ← succ.data
14:    node.right ← BST-Delete(node.right, succ.data)
15: end if
16: return node
```

Python implementation: bst-operations.py

```python
class Node:
    def __init__(self, data):
        self.data  = data
        self.left  = None
        self.right = None


def bst_insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left  = bst_insert(root.left, key)
    elif key > root.data:
        root.right = bst_insert(root.right, key)
    return root


def bst_search(root, key):
    if root is None or root.data == key:
        return root
    if key < root.data:
        return bst_search(root.left, key)
    return bst_search(root.right, key)


def find_min(node):
    while node.left:
        node = node.left
    return node


def bst_delete(root, key):
    if root is None:
        return None
    if key < root.data:
        root.left  = bst_delete(root.left, key)
    elif key > root.data:
        root.right = bst_delete(root.right, key)
    else:
        if root.left is None:        # Case 1 and 2
            return root.right
        if root.right is None:       # Case 2
            return root.left
        succ       = find_min(root.right)   # Case 3
        root.data  = succ.data
        root.right = bst_delete(root.right, succ.data)
    return root


def inorder(root):
    if root:
        inorder(root.left)
        print(root.data, end=" ")
        inorder(root.right)


# --- Driver Code ---
root = None
for val in [50, 30, 70, 20, 40, 60, 80]:
    root = bst_insert(root, val)

print("Inorder (sorted):", end=" ")
inorder(root)    # 20 30 40 50 60 70 80
print()

root = bst_delete(root, 30)
print("After deleting 30:", end=" ")
```

```
inorder(root)    # 20 40 50 60 70 80
print()

node = bst_search(root, 60)
print("Search 60:", node.data if node else "Not found")    # 60
```

5.5.6 Inorder Traversal of BST Gives Sorted Order

Concept: Why Inorder of BST is Sorted

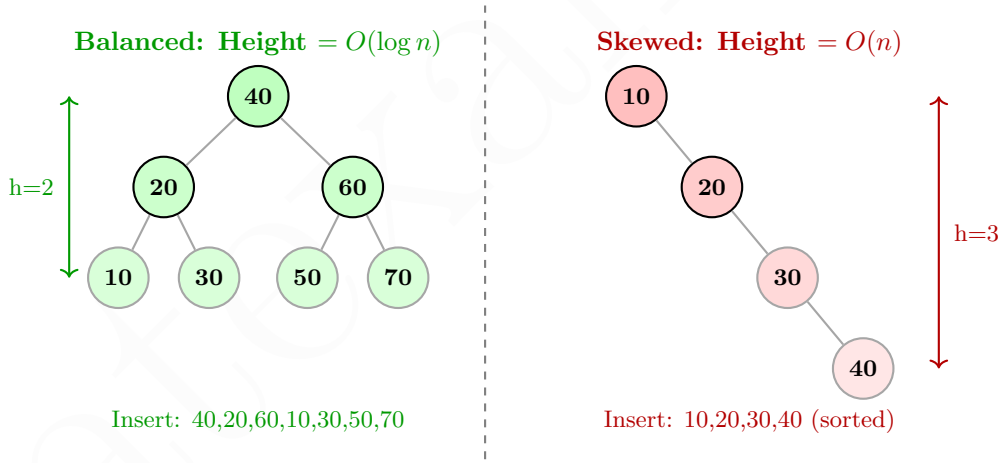
In a BST, for any node, all left descendants are smaller and all right descendants are larger. Inorder traversal visits left subtree first, then root, then right subtree. Since left is always smaller than root, and root is always smaller than right, the inorder sequence comes out in increasing order automatically. This property is used to sort a list using a BST.

5.5.7 Height of BST: Best and Worst Case

Concept: Best and Worst Case Height

The height of a BST determines the time for search, insert, and delete, all of which are $O(h)$.

- **Best case** (balanced BST): height $= O(\log n)$. This happens when keys are inserted in a way that the tree stays balanced. Operations are $O(\log n)$.
- **Worst case** (skewed BST): height $= O(n)$. This happens when keys are inserted in sorted order (ascending or descending), making the tree degenerate into a linked list. Operations become $O(n)$.



5.5.8 Number of Distinct BSTs: Catalan Numbers

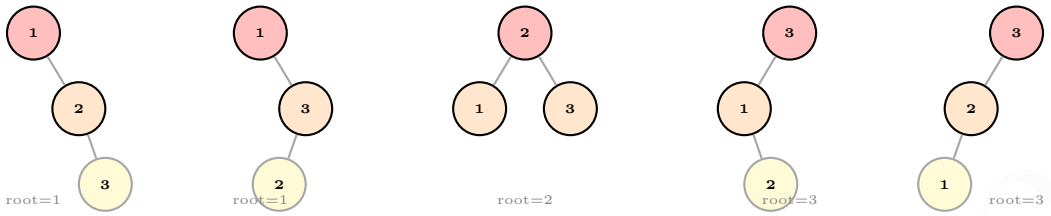
Concept: Catalan Number C_n

The number of structurally distinct BSTs that can be formed with n distinct keys is the nth Catalan number, given by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

The first few Catalan numbers are: $C_0 = 1, C_1 = 1, C_2 = 2, C_3 = 5, C_4 = 14, C_5 = 42$.
For $n = 3$ keys $\{1, 2, 3\}$, there are $C_3 = 5$ distinct BSTs.

5 Distinct BSTs with keys {1, 2, 3}



$C_3 = 5$ **distinct BSTs**

gatexamin.in

5.6 Practice Problems

Problem 145 ***[MCQ]*** *A full binary tree has 15 nodes. How many of these nodes are leaf nodes?*

(A) 6

(B) 7

(C) 8

(D) 9

Problem 146 ***[NAT]*** *A full binary tree has exactly 21 leaf nodes. The total number of nodes in this tree is _____.*

Problem 147 ***[MSQ]*** *A binary tree has n nodes. The minimum possible height of this tree (using the convention that a single-node tree has height 0) is:*

(A) $\lfloor \log_2 n \rfloor$

(B) $\lceil \log_2 n \rceil$

(C) $\lceil \log_2(n+1) \rceil - 1$

(D) $\lfloor \log_2(n+1) \rfloor$

Problem 148 ***[MSQ]*** *Which of the following are TRUE about a complete binary tree with n nodes (using 1-indexed array representation)?*

(A) The parent of the node at index i is at index $\lfloor i/2 \rfloor$, for $i > 1$.

(B) All leaf nodes have indices in the range $\lfloor n/2 \rfloor + 1$ to n.

(C) A node at index i has a right child if and only if $2i + 1 \leq n$.

(D) For any $n \geq 1$, the root is always a leaf node.

Problem 149 ***[NAT]*** *In a binary tree stored in a 1-indexed array of size 20, if the node at index 7 has a right child, what is the index of the right child? _____*

Problem 150 ***[MCQ]*** *A perfect binary tree has height $h = 4$ (root at height 0). The number of internal nodes is:*

(A) 15

(B) 16

(C) 31

(D) 30

Problem 151 ***[MSQ]*** *Which of the following statements correctly distinguish a complete binary tree from a perfect binary tree?*

(A) Every perfect binary tree is also a complete binary tree.

(B) Every complete binary tree is also a perfect binary tree.

(C) A complete binary tree with n nodes has height $\lfloor \log_2 n \rfloor$.

(D) In a complete binary tree, all leaf nodes must be at the same level.

Problem 152 ***[MCQ]*** *Consider a binary tree stored in a 1-indexed array. Node at index 5 has value 40. Its grandparent (parent of parent) is at index:*

(A) 1

(B) 2

(C) 3

(D) 4

Problem 153 ***[MSQ]*** *Which of the following are TRUE about a complete binary tree with n nodes (using 0-indexed array representation)?*

(A) The parent of the node at index i is at index $\lfloor (i-1)/2 \rfloor$, for $i > 0$.

(B) All leaf nodes have indices in the range $\lfloor n/2 \rfloor$ to $n-1$.

(C) A node at index i has a right child if and only if $2i + 2 \leq n - 1$.

(D) For any $n \geq 1$, the root is always a leaf node.

Problem 154 ***[MCQ]*** *The inorder traversal of a binary tree gives: $4, 2, 5, 1, 6, 3, 7$. The preorder traversal gives: $1, 2, 4, 5, 3, 6, 7$. Which of the following is the postorder traversal?*

(A) $4, 5, 2, 6, 7, 3, 1$

(B) $4, 2, 5, 6, 3, 7, 1$

(C) $7, 6, 3, 5, 4, 2, 1$

(D) $4, 5, 2, 7, 6, 3, 1$

Problem 155 ***[NAT]*** *A binary tree's preorder traversal gives: A, B, D, E, C, F. Its inorder traversal gives: D, B, E, A, C, F. In the reconstructed tree, the right child of the root is a node whose value in the postorder traversal appears at position (1-indexed in postorder) _____.*

Problem 156 ***[MCQ]*** *Consider the following Python function on a binary tree:*

```
def mystery(root):
    if root is None:
        return 0
    return 1 + max(mystery(root.left), mystery(root.right))
```

What does this function compute?

(A) Number of nodes in the tree

(B) Height of the tree (where a single node has height 1)

(C) Number of leaf nodes

(D) Depth of the deepest leaf (0-indexed)

Problem 157 ***[MSQ]*** *A binary tree has inorder traversal I and preorder traversal P. Which of the following are TRUE?*

(A) The first element of P is always the root.

(B) The last element of I is always a leaf node.

(C) Given I and P, the binary tree can always be uniquely reconstructed.

(D) The root splits I into left and right subtree elements.

Problem 158 ***[MCQ]*** *Consider the following algorithm on a binary tree:*

Algorithm: Traverse(root)

```
1: if root = NULL then return
2: end if
3: print root.data
4: Traverse(root.right)
5: Traverse(root.left)
```

For the tree with root 1, left child 2, right child 3, where 2 has left child 4 and right child 5, this algorithm prints:

(A) 1, 2, 4, 5, 3

(B) 1, 3, 2, 5, 4

(C) 1, 3, 5, 2, 4

(D) 4, 2, 5, 1, 3

Problem 159 ***[MCQ]*** *A binary tree has the same inorder and preorder traversal. Which of the following must be TRUE about this tree?*

(A) It is a perfect binary tree.

(B) Every node has only a right child (right-skewed) or is a leaf.

(C) Every node has only a left child (left-skewed) or is a leaf.

(D) Not possible.

Problem 160 ***[NAT]*** *A binary tree T has 12 nodes in total. The preorder and postorder traversal sequences of T are given, and it is known that exactly 4 of the internal nodes of T have exactly one child (the remaining internal nodes have either zero or two children). The number of distinct binary trees T that can be constructed is _____.*

Problem 161 ***[MCQ]*** *The preorder and postorder traversals of a binary tree are:*

- *Preorder: A, B, C*
- *Postorder: C, B, A*

How many structurally distinct binary trees produce this pair of traversals?

(A) 1

(B) 2

(C) 4

(D) Cannot be determined

Problem 162 ***[NAT]*** *A binary tree with 8 nodes has inorder traversal: 8, 4, 9, 2, 10, 5, 1, 6 and preorder traversal: 1, 2, 4, 8, 9, 5, 10, 6. In the reconstructed tree, the depth (root at depth 0) of the node with value 10 is _____.*

Problem 163 ***[MCQ]*** *The following keys are inserted one by one into an initially empty BST: $10, 5, 15, 3, 7, 12, 18$. What is the height of the resulting BST (height of single-node tree = 0)?*

(A) 2

(B) 3

(C) 4

(D) 1

Problem 164 ***[MCQ]*** *The keys $5, 3, 7, 1, 4, 6, 8$ are inserted one by one into an initially empty BST. The node that is the inorder successor of node 5 in the resulting BST is:*

(A) 6

(B) 7

(C) 8

(D) 4

Problem 165 ***[NAT]*** *The following sequence is used to build a BST: $50, 30, 70, 20, 40, 60, 80, 10, 25$. After building the BST, the node with value 30 is deleted. The inorder successor used to replace 30 (if applicable) has value _____.*

Problem 166 ***[MSQ]*** *Consider a BST where the inorder traversal gives the sequence $2, 4, 6, 8, 10, 12, 14$. Which of the following statements are TRUE?*

(A) The root could be any of the 7 values.

(B) If the root is 8, the left subtree contains $\{2, 4, 6\}$ and the right subtree contains $\{10, 12, 14\}$.

(C) There exists exactly one BST with this inorder traversal and root value 8.

(D) The number of distinct BSTs possible with these 7 keys is $C_7 = 429$.

Problem 167 ***[MCQ]*** *Consider the following Python function on a BST:*

```
mystery(root, lo, hi):
    if root is NULL:
        return 0

    result <- mystery(root.left, lo, hi) + mystery(root.right, lo, hi)

    if lo <= root.data <= hi:
        result <- result + root.data

    return result
```

For a BST with inorder sequence $1, 3, 5, 7, 9, 11, 13$ *and root = 7, calling* `mystery(root, 3, 9)` *returns:*

(A) 24

(B) 18

(C) 15

(D) 25

Problem 168 ***[MCQ]*** *In a BST, node X has two children. When X is deleted using the inorder successor method, the inorder successor is guaranteed to:*

(A) Have no children (be a leaf)

(B) Have at most one child (specifically, at most a right child)

(C) Have exactly two children

(D) Have at most one child (specifically, at most a left child)

Problem 169 ***[MCQ]*** *Consider a BST where searching for a key k takes exactly 5 comparisons in the worst case. What is the height of the BST (where a single-node tree has height 0)?*

(A) 3

(B) 4

(C) 5

(D) 6

Problem 170 ***[MCQ]*** *In a BST, the inorder predecessor of a node X that has a left subtree is:*

(A) The parent of X

(B) The leftmost node in X's right subtree

(C) The rightmost node in X's left subtree

(D) The parent of X's left child

Problem 171 ***[MSQ]*** *Consider the following Python BST deletion function:*

```
def delete(root, key):
    if root is NULL:
        return NULL
    if key < root.data:
        root.left  = delete(root.left, key)
    elif key > root.data:
        root.right = delete(root.right, key)
    else:
        if root.left is NULL:
            return root.right
        if root.right is NULL:
            return root.left
        succ      = root.right
        while succ.left:
            succ  = succ.left
        root.data  = succ.data
        root.right = delete(root.right, succ.data)
    return root
```

Which of the following are TRUE about this implementation?

(A) For a node with two children, the inorder successor is found in $O(h)$ time.

(B) After deletion, the BST property is always maintained.

(C) The function deletes the node physically and frees memory.

(D) For a leaf node, the function returns NULL to its parent.

Problem 172 ***[MCQ]*** *A BST is constructed by inserting keys in the order $k_1, k_2, \ldots, k_n$. The same set of keys is inserted in sorted ascending order into a second BST. Comparing the two resulting trees:*

(A) They always have the same height.

(B) The sorted-order BST is right-skewed with height $n - 1$.

(C) The sorted-order BST is left-skewed with height $n - 1$.

(D) The two trees always have the same inorder traversal.

Problem 173 ***[MCQ]*** *Consider the following pseudocode function on a BST:*

```
FindKth(node, k, count):
    if node == NULL:
      return NULL
    result = FindKth(node.left, k, count)
    if result != NULL:
      return result
    count = count + 1
    if count == k:
      return node
    return FindKth(node.right, k, count)
```

This function returns:

(A) The node with the kth smallest key in the BST

(B) The node at depth k in the BST

(C) The kth node in preorder traversal

(D) The node with the kth largest key in the BST

Problem 174 ***[MSQ]*** *Consider a BST with n nodes where all keys are distinct. Which of the following are TRUE?*

(A) The minimum key is always the leftmost node (no left child) reachable from root by going only left.

(B) The maximum key is always the root's rightmost descendant.

(C) If the BST is built by inserting n keys in random order, the expected height is $O(\log n)$.

(D) Searching for a key takes $O(\log n)$ time in ALL cases.

Problem 175 ***[NAT]*** *A BST is given with the following structure (shown as insertion sequence): $20, 10, 30, 5, 15, 25, 35$. The node with value 10 is now deleted. After deletion using the inorder successor strategy, the value at the root's left child is _____.*

Problem 176 ***[MCQ]*** *Consider the following algorithm defined on a binary tree, where each node has fields `left`, `right`, and `data`. The algorithm uses global variables `minVal` and `maxVal` to track the minimum and maximum data values seen in the subtree processed so far:*

```
check(root):
    if root is NULL:
        minVal <- +infinity
        maxVal <- -infinity
        return True

    lValid <- check(root.left)
    savedMin <- minVal
```

```
    savedMax <- maxVal

    rValid <- check(root.right)

    if lValid = True and rValid = True and savedMax < root.data and root.data < minVal:
        minVal <- smaller of savedMin and root.data
        maxVal <- larger of maxVal and root.data
        return True

    return False
```

Let v denote the value returned by `check(root)`*. What does v verify about the tree?*

(A) Whether the tree is a complete binary tree

(B) Whether the tree is a valid BST

(C) Whether the tree is height-balanced

(D) Whether the tree is a full binary tree

Problem 177 ***[NAT]*** *Consider the following recursive function:*

```
int countPaths(Node root, int target, int currSum) {
    if (root == NULL)
        return 0;

    currSum = currSum + root.data;

    int paths = 0;
    if (currSum == target)
        paths = 1;

    paths = paths + countPaths(root.left, target, currSum);
    paths = paths + countPaths(root.right, target, currSum);

    return paths;
}
```

For a BST built by inserting $[5, 3, 7, 1, 4]$ in this order, the value returned by `count_paths(root, 9, 0)` *is* _____.

Problem 178 ***[NAT]*** *Consider the following algorithm on a binary tree that computes a property of each node and stores it:*

Algorithm: Compute(node)

```
1: if node = NULL then return 0
2: end if
3: L ← Compute(node.left)
4: R ← Compute(node.right)
5: node.val ← L + R + 1
6: return node.val
```

After running `Compute` *on a perfect binary tree of height 3 (8 leaf nodes, 15 total nodes), the value stored at the root is* _____.

Problem 179 ***[MCQ]*** *The following function is run on a BST:*

```
transform(root):
    if root == NULL:
        return

    transform(root.right)
    transform(root.left)

    temp <- root.left
    root.left <- root.right
    root.right <- temp
```

After calling `transform(root)`*, the inorder traversal of the resulting tree gives elements in what order (relative to the original BST's inorder)?*

(A) Same sorted order as original

(B) Reverse sorted order

(C) Random order

(D) Preorder of the original tree

Problem 180 ***[MSQ]*** *A binary tree T has the following property: for every node, the number of nodes in its left subtree equals the number of nodes in its right subtree (a* perfectly balanced *tree). Which of the following are TRUE?*

(A) The total number of nodes n in T must be of the form $2^k - 1$ for some positive integer k.

(B) The height of T is $\lfloor \log_2 n \rfloor$.

(C) Every such tree is also a perfect binary tree.

(D) If $n = 7$ and the values are 1 through 7, there is exactly one such BST.

Problem 181 ***[NAT]*** *Consider a BST where we repeatedly delete the root (replacing with inorder successor) until the tree is empty. For the BST built from the insertion sequence $4, 2, 6, 1, 3, 5, 7$, what is the value deleted in the 3rd deletion step?* _____

Problem 182 ***[NAT]*** *Consider a BST where the keys $1, 2, 3, \ldots, n$ are inserted in the following order: first insert the median, then recursively insert medians of the left and right halves. For $n = 7$ (keys 1 through 7), the key inserted first (root) is 4, and the resulting BST has height* _____.

Problem 183 ***[MCQ]*** *A BST has n keys. We want to check whether a given value v lies between the minimum and maximum values stored in the BST without finding the actual minimum and maximum explicitly. The most efficient approach using BST properties requires at most how many comparisons?*

(A) $O(1)$

(B) $O(\log n)$

(C) $O(h)$ where h is the height

(D) $O(n)$

Problem 184 ***[MCQ]*** *Consider two traversals of a binary tree. If the preorder traversal visits nodes in order $P_1, P_2, \ldots, P_n$ and the inorder traversal visits nodes in order $I_1, I_2, \ldots, I_n$, then the root of the right subtree of the root appears at what position in the preorder traversal?*

(A) Position 2

(B) Position $k + 2$, where k is the number of nodes in the left subtree of the root

(C) Position $k + 1$, where k is the number of nodes in the left subtree of the root

(D) Position $n/2 + 1$

Problem 185 ***[NAT]*** *A BST supports a* `rank(k)` *operation that returns the kth smallest element. For the BST built by inserting $[8, 3, 10, 1, 6, 14, 4, 7, 13]$ in this order, the value returned by* `rank(6)` *is* _____.

5.7 GATE PYQs

GATEPYQ 85 (GATE CSE 2026) *The set T represents various traversals over a binary tree. The set S represents the order of visiting nodes during a traversal.*

	T		***S***
I.	*Inorder*	*L.*	*left subtree, node, right subtree*
II.	*Preorder*	*M.*	*node, left subtree, right subtree*
III.	*Postorder*	*N.*	*left subtree, right subtree, node*

Which one of the following is the correct match from T to S?

A. I-L, II-M, III-N

B. I-M, II-L, III-N

C. I-N, II-M, III-L

D. I-L, II-N, III-M

GATEPYQ 86 (GATE CSE 2026) *The height of a binary tree is the number of edges in the longest path from the root to a leaf in the tree. The maximum possible height of a full binary tree with 23 nodes is _____. (Answer in integer)*

GATEPYQ 87 (GATE CSE 2025) *Consider a binary tree T in which every node has either zero or two children. Let $n > 0$ be the number of nodes in T. Which ONE of the following is the number of nodes in T that have exactly two children?*

A. $\frac{n-2}{2}$

B. $\frac{n-1}{2}$

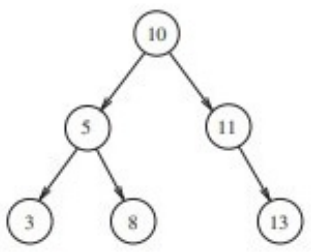
C. $\frac{n}{2}$

D. $\frac{n+1}{2}$

GATEPYQ 88 (GATE CSE 2023) *Consider the C function `foo` and the binary tree structure defined below:*

```
typedef struct node {
   int val;
   struct node *left, *right;
} node;

int foo(node *p) {
   int retval;
   if (p == NULL)
       return 0;
   else {
       retval = p->val + foo(p->left) + foo(p->right);
       printf("%d ", retval);
       return retval;
   }
}
```



When foo is called with a pointer to the root node of the given binary tree, what will it print?

A. 3 8 5 13 11 10

B. 3 5 8 10 11 13

C. 3 8 16 13 24 50

D. 3 16 8 50 24 13

GATEPYQ 89 (GATE CSE 2019) *Let T be a full binary tree with 8 leaves. (A full binary tree has every level full.) Suppose two leaves a and b of T are chosen uniformly and independently at random. The expected value of the distance between a and b in T (i.e., the number of edges in the unique path between a and b) is (rounded off to 2 decimal places) _____.*

GATEPYQ 90 (GATE CSE 2019) *The postorder traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from the root to any leaf. The height of the binary tree above is _____.*

GATEPYQ 91 (GATE CSE 2016) *Consider the following New-order strategy for traversing a binary tree:*

Visit the root;

Visit the right subtree using New-order;

Visit the left subtree using New-order;

The New-order traversal of the expression tree corresponding to the reverse polish expression $3 \quad 4 \quad * \quad 5 \quad - \quad 2 \quad \wedge \quad 6 \quad 7 \quad * \quad 1 \quad + \quad - \;$ *is given by:*

A. $+ - 167 * 2 \wedge 5 - 34*$

B. $- + 1 * 76 \wedge 2 - 5 * 43$

C. $- + 1 * 67 \wedge 2 - 5 * 34$

D. $176 * +2543 * - \wedge -$

GATEPYQ 92 (GATE CSE 2015 SET-3) *Consider a binary tree T that has 200 leaf nodes. Then, the number of nodes in T that have exactly two children are _____.*

GATEPYQ 93 (GATE CSE 2015) *A binary tree T has 20 leaves. The number of nodes in T having two children is _____.*

GATEPYQ 94 (GATE CSE 2015) *The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are:*

A. 63 and 6, respectively

B. 64 and 5, respectively

C. 32 and 6, respectively

D. 31 and 5, respectively

GATEPYQ 95 (GATE CSE 2014 SET-3) *Consider the pseudocode given below. The function `DoSomething()` takes as argument a pointer to the root of an arbitrary tree represented by the `leftMostChild-rightSibling` representation. Each node of the tree is of type `treeNode`.*

```
typedef struct treeNode* treeptr;
struct treeNode
{
    treeptr leftMostChild, rightSibling;
};

int DoSomething (treeptr tree)
{
     int value = 0;
     if (tree != NULL) {
          if (tree->leftMostChild == NULL)
               value = 1;
          else
```

```
            value = DoSomething(tree->leftMostChild);
        value = value + DoSomething(tree->rightSibling);
    }
    return(value);
}
```

When the pointer to the root of a tree is passed as the argument to `DoSomething`*, the value returned by the function corresponds to the:*

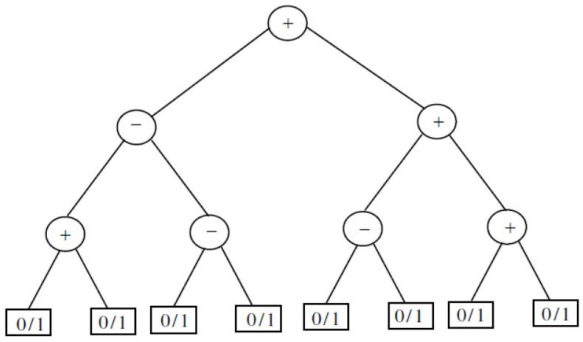
A. number of internal nodes in the tree

B. height of the tree

C. number of nodes without a right sibling in the tree.

D. number of leaf nodes in the tree.

GATEPYQ 96 (GATE CSE 2014) *Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is* _____.



GATEPYQ 97 (GATE CSE 2014) *Consider rooted n node binary tree represented using pointers. The best upper bound on the time required to determine the number of subtrees having exactly 4 nodes is* $O(n^a \log^b n)$*. Then the value of* $a + 10b$ *is* _____.

GATEPYQ 98 (GATE CSE 2012) *The height of a tree is defined as the number of edges on the longest path in the tree. The function shown in the pseudocode below is invoked as* `height(root)` *to compute the height of a binary tree rooted at the tree pointer* `root`*.*

```
int height (treeptr n)
{
   if (n == NULL) return -1;
   if (n -> left == NULL)
        if (n -> right == NULL) return 0;
        else return B1;
   else {
        h1 = height (n -> left);
        if (n -> right == NULL) return (1 + h1);
        else {
            h2 = height (n -> right);
            return B2;
        }
   }
}
```

The appropriate expressions for the two boxes `B1` *and* `B2` *are:*

A. `B1: (1 + height(n->right))`
`B2: (1 + max(h1, h2))`

B. `B1: (height(n->right))`
`B2: (1 + max(h1, h2))`

C. `B1: height(n->right)`
`B2: max(h1, h2)`

D. `B1: (1 + height(n->right))`
`B2: max(h1, h2)`

GATEPYQ 99 (GATE CSE 2010) *In a binary tree with n nodes, every node has an odd number of descendants. Every node is considered to be its own descendant. What is the number of nodes in the tree that have exactly one child?*

A. *0*

B. *1*

C. $(n-1)/2$

D. $n-1$

GATEPYQ 100 (GATE CSE 2007) *Consider the following C program segment where* `CellNode` *represents a node in a binary tree:*

```
struct CellNode
{
  struct CellNode *leftChild;
  int element;
  struct CellNode *rightChild;
};

int GetValue(struct CellNode *ptr)
{
  int value = 0;
  if (ptr != NULL)
  {
     if ((ptr->leftChild == NULL) && (ptr->rightChild == NULL))
        value = 1;
     else
        value = value + GetValue(ptr->leftChild) + GetValue(ptr->rightChild);
  }
  return(value);
}
```

The value returned by `GetValue` *when a pointer to the root of a binary tree is passed as its argument is:*

A. *the number of nodes in the tree*

B. *the number of internal nodes in the tree*

C. *the number of leaf nodes in the tree*

D. *the height of the tree*

GATEPYQ 101 (GATE CSE 2007) *The inorder and preorder traversal of a binary tree are* `d b e a f c g` *and* `a b d e c f g`*, respectively. The postorder traversal of the binary tree is:*

A. *d e b f g c a*

B. *e d b g f c a*

C. *e d b f g c a*

D. *d e f g b c a*

GATEPYQ 102 (GATE CSE 2007) *The maximum number of binary trees that can be formed with three unlabeled nodes is:*

A. *1*

B. *5*

C. *4*

D. *3*

GATEPYQ 103 (GATE CSE 2007) *The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is:*

A. $2^h - 1$

B. $2^{h-1} - 1$

C. $2^{h+1} - 1$

D. 2^{h+1}

GATEPYQ 104 (GATE IT 2006) *An array X of n distinct integers is interpreted as a complete binary tree. The index of the first element of the array is 0. If the root node is at level 0, the level of element $X[i], i \neq 0$, is:*

A. $\lfloor \log_2 i \rfloor$

B. $\lceil \log_2 (i+1) \rceil$

C. $\lfloor \log_2 (i+1) \rfloor$

D. $\lceil \log_2 i \rceil$

GATEPYQ 105 (GATE IT 2006) *An array X of n distinct integers is interpreted as a complete binary tree. The index of the first element of the array is 0. The index of the parent of element $X[i], i \neq 0$, is?*

A. $\lfloor \frac{i}{2} \rfloor$

B. $\lceil \frac{i-1}{2} \rceil$

C. $\lceil \frac{i}{2} \rceil$

D. $\lceil \frac{i}{2} \rceil - 1$

GATEPYQ 106 (GATE IT 2006) *In a binary tree, the number of internal nodes of degree 1 is 5, and the number of internal nodes of degree 2 is 10. The number of leaf nodes in the binary tree is:*

A. 10

B. 11

C. 12

D. 15

GATEPYQ 107 (GATE CSE 2006) *A scheme for storing binary trees in an array X is as follows. Indexing of X starts at 1 instead of 0. The root is stored at $X[1]$. For a node stored at $X[i]$, the left child, if any, is stored in $X[2i]$ and the right child, if any, in $X[2i+1]$. To be able to store any binary tree on n vertices, the minimum size of X should be:*

A. $\log n$

B. n

C. $2n + 1$

D. $2^n - 1$

GATEPYQ 108 (GATE CSE 2004) *Consider the following C program segment:*

```
struct CellNode
{
  struct CellNode *leftChild;
  int element;
  struct CellNode *rightChild;
};

int DoSomething(struct CellNode *ptr)
{
    int value = 0;
    if (ptr != NULL)
    {
      if (ptr->leftChild != NULL)
        value = 1 + DoSomething(ptr->leftChild);
      if (ptr->rightChild != NULL)
        value = max(value, 1 + DoSomething(ptr->rightChild));
    }
    return (value);
}
```

The value returned by the function `DoSomething` when a pointer to the root of a non-empty tree is passed as argument is:

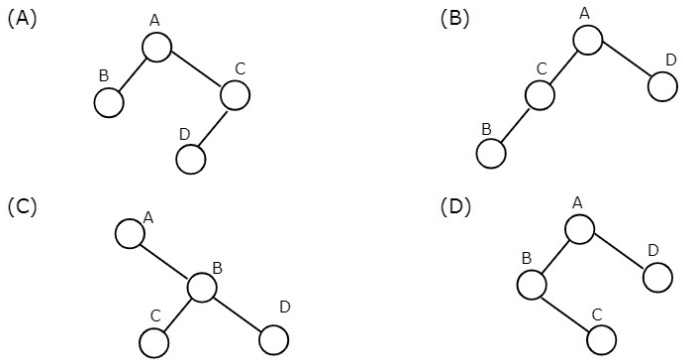
A. *The number of leaf nodes in the tree*

B. *The number of nodes in the tree*

C. *The number of internal nodes in the tree*

D. *The height of the tree*

GATEPYQ 109 (GATE IT 2004) *Which one of the following binary trees has its inorder and preorder traversals as BCAD and ABCD, respectively?*



A. *A*

B. *B*

C. *C*

D. *D*

GATEPYQ 110 (GATE CSE 2004) *Consider the label sequences obtained by the following pairs of traversals on a labeled binary tree. Which of these pairs identify a tree uniquely?*

(i) preorder and postorder

(ii) inorder and postorder

(iii) preorder and inorder

(iv) level order and postorder

A. *(i) only*

B. *(iii) only*

C. *(iv) only*

D. *(ii), (iii)*

GATEPYQ 111 (GATE CSE 2000) *Let `LASTPOST`, `LASTIN` and `LASTPRE` denote the last vertex visited in a postorder, inorder and preorder traversal respectively, of a complete binary tree. Which of the following is always true?*

A. `LASTIN` = `LASTPOST`

B. `LASTIN` = `LASTPRE`

C. `LASTPRE` = `LASTPOST`

D. *None of the above*

GATEPYQ 112 (GATE CSE 2000) *Consider the following nested representation of binary trees: `(X Y Z)` indicates `Y` and `Z` are the left and right subtrees, respectively, of node `X`. Note that `Y` and `Z` may be `NULL`, or further nested. Which of the following represents a valid binary tree?*

A. `(1 2 (4 5 6 7))`

B. `(1 (2 3 4) 5 6) 7)`

C. `(1 (2 3 4) (5 6 7))`

D. `(1 (2 3 NULL) (4 5))`

GATEPYQ 113 (GATE CSE 1998) *Which of the following statements is false?*

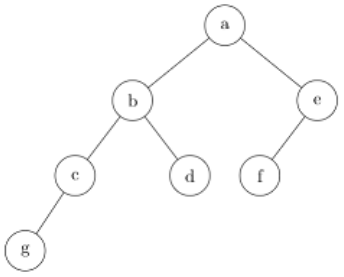
A. *A tree with n nodes has $(n-1)$ edges*

B. *A labeled rooted binary tree can be uniquely constructed given its postorder and preorder traversal results.*

C. *A complete binary tree with n internal nodes has $(n+1)$ leaves.*

D. *The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$*

GATEPYQ 114 (GATE CSE 1996) *Which of the following sequences denotes the post order traversal sequence of the below tree?*



A. *fegcdba*

B. *gcbdafe*

C. *gcdbfea*

D. *fedgcba*

GATEPYQ 115 (GATE CSE 1995) *A binary tree T has n leaf nodes. The number of nodes of degree 2 in T is:*

A. $\log_2 n$

B. $n-1$

C. n

D. $2n$

GATEPYQ 116 (GATE CSE 2026) *Consider a binary search tree (BST) with n leaf nodes $(n > 0)$. Given any node V, the key present in the node is denoted as $Val(V)$. All the keys present in the given BST are distinct. The keys belong to the set of real numbers.*

For a node V, let $Suc(V)$ denote the node that is its inorder successor. If a node V does not have an inorder successor, then $Suc(V)$ is `NULL`. As there are no duplicates, if $Suc(V)$ is not `NULL`, then $Val(V) < Val(Suc(V))$.

Corresponding to every leaf node L_i that has a non-`NULL` $Suc(L_i)$, a new key k_i with the following property is to be inserted into the BST:

$$Val(L_i) < k_i < Val(Suc(L_i))$$

Let K represent the list of all such new keys to be inserted into the BST.

Which of the following statements is/are true?

A. *K cannot have any duplicates*

B. *K will have at least one element*

C. *After inserting all keys from K, the height of the BST can increase at most by one*

D. *Number of nodes in the BST will double after inserting all keys from K*

GATEPYQ 117 (GATE CSE 2026) *The following sequence corresponds to the preorder traversal of a binary search tree T:*

$$50, 25, 13, 40, 30, 47, 75, 60, 70, 80, 77$$

The position of the element 60 in the postorder traversal of T is _____. (Answer in integer)

Note: The position begins with 1.

GATEPYQ 118 (GATE CSE 2026) *Let P be the set of all integers from 1 to 15. Consider any order of insertion of the elements of P into a binary search tree that creates a complete binary tree.*
Which one of the following elements can NEVER be the third element that is inserted?

A. *4*

B. *2*

C. *10*

D. *5*

GATEPYQ 119 (GATE CSE 2025) *Suppose the values 10, -4, 15, 30, 20, 5, 60, 19 are inserted in that order into an initially empty binary search tree. Let T be the resulting binary search tree. The number of edges in the path from the node containing 19 to the root node of T is _____. (Answer in integer)*

GATEPYQ 120 (GATE CSE 2025) *Which of the following statement(s) is/are TRUE for any binary search tree (BST) having n distinct integers?*

A. *The maximum length of a path from the root node to any other node is $(n-1)$.*

B. *An inorder traversal will always produce a sorted sequence of elements.*

C. *Finding an element takes $O(\log_2 n)$ time in the worst case.*

D. *Every BST is also a Min-Heap.*

GATEPYQ 121 (GATE CSE 2024) *You are given a set V of distinct integers. A binary search tree T is created by inserting all elements of V one by one, starting with an empty tree. The tree T follows the convention that, at each node, all values stored in the left subtree of the node are smaller than the value stored at the node. You are not aware of the sequence in which these values were inserted into T, and you do not have access to T.*
Which one of the following statements is TRUE?

A. *Inorder traversal of T can be determined from V*

B. *Root node of T can be determined from V*

C. *Preorder traversal of T can be determined from V*

D. *Postorder traversal of T can be determined from V*

GATEPYQ 122 (GATE CSE 2022) *Suppose a binary search tree with 1000 distinct elements is also a complete binary tree. The tree is stored using the array representation of binary heap trees. Assuming that the array indices start with 0, the 3rd largest element of the tree is stored at index _____.*

GATEPYQ 123 (GATE CSE 2021) *A binary search tree T contains n distinct elements. What is the time complexity of picking an element in T that is smaller than the maximum element in T?*

A. $\Theta(n \log n)$

B. $\Theta(n)$

C. $\Theta(\log n)$

D. $\Theta(1)$

GATEPYQ 124 (GATE CSE 2020) *The preorder traversal of a binary search tree is 15, 10, 12, 11, 20, 18, 16, 19. Which one of the following is the postorder traversal of the tree?*

A. *10, 11, 12, 15, 16, 18, 19, 20*

B. *11, 12, 10, 16, 19, 18, 20, 15*

C. *20, 19, 18, 16, 15, 12, 11, 10*

D. *19, 16, 18, 20, 11, 12, 10, 15*

GATEPYQ 125 (GATE CSE 2017) *The pre-order traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20. Then the post-order traversal of this tree is:*

A. *2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20*

B. 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12

C. 7, 2, 6, 8, 9, 10, 20, 17, 19, 15, 16, 12

D. 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 12

GATEPYQ 126 (GATE CSE 2017) *Let T be a binary search tree with 15 nodes. The minimum and maximum possible heights of T are:*
Note: The height of a tree with a single node is 0.

A. 4 and 15 respectively

B. 3 and 14 respectively

C. 4 and 14 respectively

D. 3 and 15 respectively

GATEPYQ 127 (GATE CSE 2016) *The number of ways in which the numbers 1, 2, 3, 4, 5, 6, 7 can be inserted in an empty binary search tree, such that the resulting tree has height 6, is _____.*
Note: The height of a tree with a single node is 0.

GATEPYQ 128 (GATE CSE 2015 SET-3) *While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is:*

A. 65

B. 67

C. 69

D. 83

GATEPYQ 129 (GATE CSE 2015) *What are the worst-case complexities of insertion and deletion of a key in a binary search tree?*

A. $\Theta(\log n)$ for both insertion and deletion

B. $\Theta(n)$ for both insertion and deletion

C. $\Theta(n)$ for insertion and $\Theta(\log n)$ for deletion

D. $\Theta(\log n)$ for insertion and $\Theta(n)$ for deletion

GATEPYQ 130 (GATE CSE 2015) *Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)?*

I. 3, 5, 7, 8, 15, 19, 25

II. 5, 8, 9, 12, 10, 15, 25

III. 2, 7, 10, 8, 14, 16, 20

IV. 4, 6, 7, 9, 18, 20, 25

A. I and IV only

B. II and III only

C. II and IV only

D. II only

GATEPYQ 131 (GATE CSE 2013) *The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?*

A. 10, 20, 15, 23, 25, 35, 42, 39, 30

B. 15, 10, 25, 23, 20, 42, 35, 39, 30

C. 15, 20, 10, 23, 25, 42, 35, 39, 30

D. 15, 10, 23, 25, 20, 35, 42, 39, 30

GATEPYQ 132 (GATE CSE 2013) *Which one of the following is the tightest upper bound that represents the time complexity of inserting an object into a binary search tree of n nodes?*

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n \log n)$

GATEPYQ 133 (GATE CSE 2011) *We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?*

A. *0*

B. *1*

C. $n!$

D. $\frac{1}{n+1} \cdot \binom{2n}{n}$

GATEPYQ 134 (GATE IT 2008) *How many distinct BSTs can be constructed with 3 distinct keys?*

A. *4*

B. *5*

C. *6*

D. *9*

GATEPYQ 135 (GATE IT 2008) *A Binary Search Tree (BST) stores values in the range 37 to 573. Consider the following sequence of keys.*

I. *81, 537, 102, 439, 285, 376, 305*

II. *52, 97, 121, 195, 242, 381, 472*

III. *142, 248, 520, 386, 345, 270, 307*

IV. *550, 149, 507, 395, 463, 402, 270*

Which of the following statements is TRUE?

A. *I, II and IV are inorder sequences of three different BSTs*

B. *I is a preorder sequence of some BST with 439 as the root*

C. *II is an inorder sequence of some BST where 121 is the root and 52 is a leaf*

D. *IV is a postorder sequence of some BST with 149 as the root*

GATEPYQ 136 (GATE IT 2008) *A Binary Search Tree (BST) stores values in the range 37 to 573. Consider the following sequence of keys.*

I. *81, 537, 102, 439, 285, 376, 305*

II. *52, 97, 121, 195, 242, 381, 472*

III. *142, 248, 520, 386, 345, 270, 307*

IV. *550, 149, 507, 395, 463, 402, 270*

Suppose the BST has been unsuccessfully searched for key 273. Which all of the above sequences list nodes in the order in which we could have encountered them in the search?

A. *II and III only*

B. *I and III only*

C. *III and IV only*

D. *III only*

GATEPYQ 137 (GATE CSE 2008) *You are given the postorder traversal, P, of a binary search tree on the n elements $1, 2, \ldots, n$. You have to determine the unique binary search tree that has P as its postorder traversal. What is the time complexity of the most efficient algorithm for doing this?*

A. $\Theta(\log n)$

B. $\Theta(n)$

C. $\Theta(n \log n)$

D. None of the above, as the tree cannot be uniquely determined

GATEPYQ 138 (GATE IT 2006) *Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?*

A. {*10, 75, 64, 43, 60, 57, 55*}

B. {*90, 12, 68, 34, 62, 45, 55*}

C. {*9, 85, 47, 68, 43, 57, 55*}

D. {*79, 14, 72, 56, 16, 53, 55*}

GATEPYQ 139 (GATE IT 2007) *When searching for the key value 60 in a binary search tree, nodes containing the key values 10, 20, 40, 50, 70, 80, 90 are traversed, not necessarily in the order given. How many different orders are possible in which these key values can occur on the search path from the root to the node containing the value 60?*

A. 35

B. 64

C. 128

D. 5040

GATEPYQ 140 (GATE IT 2005) *A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in pre-order and the values in each node printed out, the sequence of values obtained is 5, 3, 1, 2, 4, 6, 8, 7. If the tree is traversed in post-order, the sequence obtained would be:*

A. 8, 7, 6, 5, 4, 3, 2, 1

B. 1, 2, 3, 4, 8, 7, 6, 5

C. 2, 1, 4, 3, 6, 7, 8, 5

D. 2, 1, 4, 3, 7, 8, 6, 5

GATEPYQ 141 (GATE IT 2005) *The numbers $1, 2, \ldots, n$ are inserted in a binary search tree in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be inserted in the tree must be:*

A. p

B. $p + 1$

C. $n - p$

D. $n - p + 1$

GATEPYQ 142 (GATE CSE 2005) *How many distinct binary search trees can be created out of 4 distinct keys?*

A. 5

B. 14

C. 24

D. 42

GATEPYQ 143 (GATE CSE 2005) *Postorder traversal of a given binary search tree, T, produces the following sequence of keys:*

$$10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29$$

Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?

A. *9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95*

B. *9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29*

C. *29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95*

D. *95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29*

GATEPYQ 144 (GATE CSE 2004) *The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?*

A. *2*

B. *3*

C. *4*

D. *6*

GATEPYQ 145 (GATE CSE 2003) *Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the inorder traversal sequence of the resultant tree?*

A. *7 5 1 0 3 2 4 6 8 9*

B. *0 2 4 3 1 6 5 9 8 7*

C. *0 1 2 3 4 5 6 7 8 9*

D. *9 8 6 4 2 3 0 1 5 7*

GATEPYQ 146 (GATE CSE 2003) *Let $T(n)$ be the number of different binary search trees on n distinct elements. Then:*

$$T(n) = \sum_{k=1}^{n} T(k-1)T(x)$$

where x is:

A. $n-k+1$

B. $n-k$

C. $n-k-1$

D. $n-k-2$

GATEPYQ 147 (GATE CSE 1997) *A binary search tree contains the values 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in pre-order and the values are printed out. Which of the following sequences is a valid output?*

A. *5 3 1 2 4 7 8 6*

B. *5 3 1 2 6 4 8 7*

C. *5 3 2 4 1 6 7 8*

D. *5 3 1 2 4 7 6 8*

GATEPYQ 148 (GATE CSE 1996) *A binary search tree is generated by inserting in order the following integers:*

$$50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24$$

The number of nodes in the left subtree and right subtree of the root respectively is:

A. *(4, 7)*

B. *(7, 4)*

C. *(8, 3)*

D. *(3, 8)*

5.8 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
40	Introduction to Trees	https://www.youtube.com/watch?v=60spyQWyv9M	
41	Binary Trees	https://www.youtube.com/watch?v=mN26bLUfvn8	
42	Binary Tree Traversal: Inorder, Preorder & Postorder	https://www.youtube.com/watch?v=RGXBp7eU8XA	
43	Tree Reconstruction: Inorder, Preorder & Postorder	https://www.youtube.com/watch?v=rmgpTdhqvqY	
44	Theorems & Properties of Binary Trees: Full, Perfect & Complete	https://www.youtube.com/watch?v=5JWPEfPxLcs	
45	Inorder Successor & Predecessor in BST	https://www.youtube.com/watch?v=Mv8aW4_Jroc	
46	Binary Search Trees | Properties, Insertion and Search	https://www.youtube.com/watch?v=mR4VmTUtGvM	
47	BST Deletion: All 3 Cases | Leaf, One Child & Two Children	https://www.youtube.com/watch?v=fLtdKlI34u4	

Lec. No	Title	URL	QR Code
48	GATE PYQs on Binary Trees	https://www.youtube.com/watch?v=2b9b6m4958E	
49	GATE PYQs on Binary Search Trees	https://www.youtube.com/watch?v=DQNgmVK1-lM	
50	Practice Problems on Binary Trees & Binary Search Trees	https://www.youtube.com/watch?v=_jLpDuYBMnA	

gatefairy

Chapter 6

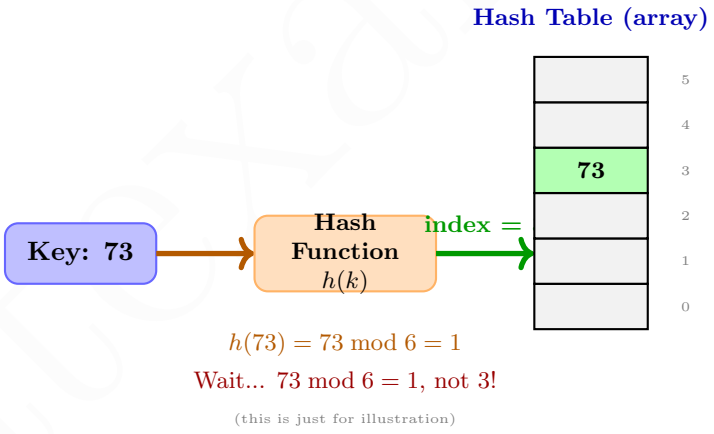
Hash Tables

Concept: What is a Hash Table

A hash table is a data structure that stores key-value pairs and allows us to find, insert, or delete a value very fast, ideally in $O(1)$ time. The secret behind this speed is a **hash function**, which converts a key (like a student roll number or a name) into an index of an array. Instead of searching the entire array, we directly jump to the right position using this index. Think of a hash table as a very intelligent filing cabinet where the drawer number is computed from the file name itself.

Example 245: Analogy: Library with Direct-Access Shelves

Imagine a library where instead of searching all shelves, you compute the shelf number from the book title directly. Book titled "Algorithms" might always go to shelf 7, "Data Structures" to shelf 3, and so on. When you want "Algorithms", you go directly to shelf 7 without checking other shelves. This is exactly what a hash table does.



6.1 Hash Functions and Load Factor

Concept: What is a Hash Function

A hash function $h(k)$ takes a key k and maps it to an integer in the range $[0, m-1]$, where m is the size of the hash table array. A good hash function should:

- Be fast to compute.
- Distribute keys uniformly across the array, to avoid many keys landing at the same position.
- Be deterministic, meaning the same key always gives the same index.

6.1.1 Division Method

Concept: Division Method

The simplest hash function. We divide the key by the table size m and take the remainder.

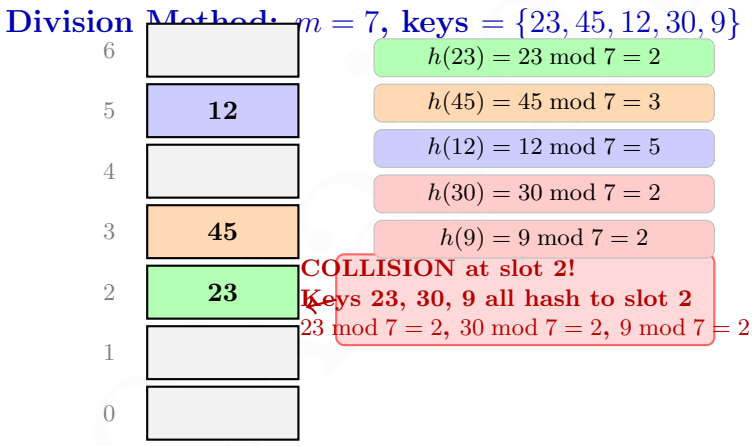
$$h(k) = k \bmod m$$

For example, if $m = 7$ and key $k = 23$, then $h(23) = 23 \bmod 7 = 2$. Key 23 goes to slot 2.
Choosing m: A prime number that is not close to a power of 2 works best for m. For example, if we have 1000 entries, choosing $m = 997$ (a prime) distributes better than $m = 1024 = 2^{10}$.

Example 246: Division Method: Step by Step

Table size $m = 7$. Insert keys: $\{23, 45, 12, 30, 9\}$.

- $h(23) = 23 \bmod 7 = 2 \rightarrow$ slot 2
- $h(45) = 45 \bmod 7 = 3 \rightarrow$ slot 3
- $h(12) = 12 \bmod 7 = 5 \rightarrow$ slot 5
- $h(30) = 30 \bmod 7 = 2 \rightarrow$ slot 2 **(COLLISION with 23!)**
- $h(9) = 9 \bmod 7 = 2 \rightarrow$ slot 2 **(COLLISION again!)**



6.1.2 Multiplication Method

Concept: Multiplication Method

In the multiplication method, we multiply the key by a constant A (where $0 < A < 1$), take the fractional part of the result, and then multiply by m.

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

Here, $k \cdot A \bmod 1$ means we take only the fractional part of $k \cdot A$. A commonly used value is $A = \dfrac{\sqrt{5}-1}{2} \approx 0.6180$, known as the golden ratio conjugate. This method works for any value of m, and powers of 2 are fine for m here.

Example 247: Multiplication Method: Step by Step

Let $m = 8$, $A = 0.6180$, key $k = 23$.

1. $k \cdot A = 23 \times 0.6180 = 14.214$
2. Fractional part $= 0.214$
3. $h(23) = \lfloor 8 \times 0.214 \rfloor = \lfloor 1.712 \rfloor = 1$

So key 23 goes to slot 1.

6.1.3 Load Factor

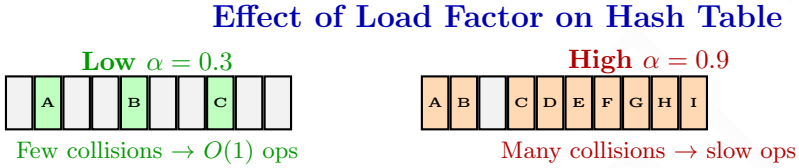
Concept: Load Factor α

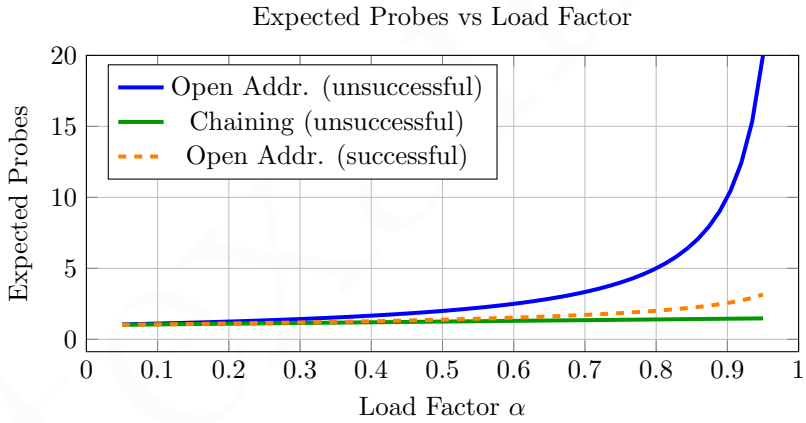
The load factor α tells us how full the hash table is.

$$\alpha = \frac{n}{m}$$

where n is the number of keys currently stored in the table, and m is the total number of slots. The load factor directly affects performance.

- When α is small (table mostly empty), collisions are rare, and operations are fast.
- When α is large (table mostly full), collisions are frequent, and operations become slow.
- For chaining, α can be > 1 (more keys than slots).
- For open addressing, α must be < 1 (we need at least one empty slot).





Example 248: Why α Matters

Suppose table size $m = 10$ and we insert $n = 7$ keys. Then $\alpha = 7/10 = 0.7$. This means 70% of slots are occupied. The remaining 30% are empty. Each new insertion is likely to find a collision, taking more time to find an empty slot. If we had $m = 20$ for the same 7 keys, $\alpha = 0.35$, and performance would be much better.

Concept: Expected $O(1)$ Time

Under the assumption of **simple uniform hashing** (each key is equally likely to hash to any slot, independently), the expected time for insert, search, and delete is $O(1)$ when the load factor α is kept constant. This is why hash tables are called $O(1)$ data structures in practice. However, in the worst case (all n keys hash to the same slot), operations take $O(n)$.

6.2 Collision Resolution

Concept: What is a Collision

A collision happens when two different keys k_1 and k_2 produce the same hash value, meaning $h(k_1) = h(k_2)$. Since there are infinitely many possible keys but only m slots, collisions are unavoidable (Pigeonhole Principle). We must handle them. There are two main strategies: **Chaining** and **Open Addressing**.

6.2.1 Chaining (Separate Chaining)

Concept: Chaining

In chaining, each slot of the hash table holds a **linked list** (or chain) of all keys that hash to that slot. When a collision occurs, we simply add the new key to the linked list at that slot. There is no limit on how many keys can go into the same slot.
Insert: Hash the key, go to that slot, add the key to the front of the linked list. $O(1)$.
Search: Hash the key, go to that slot, search the linked list at that slot. Average $O(1 + \alpha)$.
Delete: Hash the key, go to that slot, find and remove from the linked list. Average $O(1 + \alpha)$.

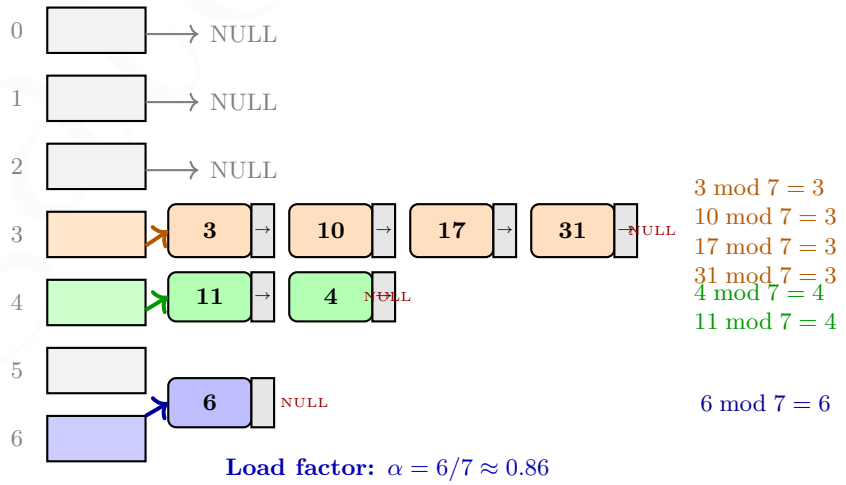
Example 249: Chaining: Step by Step

Table size $m = 5$. Insert keys: $\{12, 17, 22, 3, 7, 27\}$ using $h(k) = k \bmod 5$.

- $h(12) = 2 \rightarrow$ slot 2: [12]
- $h(17) = 2 \rightarrow$ slot 2: [17, 12] (collision, added to chain)
- $h(22) = 2 \rightarrow$ slot 2: [22, 17, 12] (another collision)
- $h(3) = 3 \rightarrow$ slot 3: [3]
- $h(7) = 2 \rightarrow$ slot 2: [7, 22, 17, 12]
- $h(27) = 2 \rightarrow$ slot 2: [27, 7, 22, 17, 12]

Here, $\alpha = 6/5 = 1.2 > 1$ (allowed in chaining!). Slot 2 has a very long chain, which is bad.

Separate Chaining: $m = 7$, keys $= \{10, 17, 24, 3, 31, 11\}$



Load factor: $\alpha = 6/7 \approx 0.86$

Python implementation: chaining-hash-table.py

```
class ChainingHashTable:
    def __init__(self, m):
        self.m     = m
        self.table = [[] for _ in range(m)]   # each slot is a list
```

```
    def hash(self, key):
        return key % self.m              # division method

    def insert(self, key):
        slot = self.hash(key)
        if key not in self.table[slot]:  # avoid duplicates
            self.table[slot].insert(0, key)  # insert at front O(1)

    def search(self, key):
        slot = self.hash(key)
        return key in self.table[slot]   # search the chain

    def delete(self, key):
        slot = self.hash(key)
        if key in self.table[slot]:
            self.table[slot].remove(key)
        else:
            print(f"{key} not found")

    def display(self):
        for i, chain in enumerate(self.table):
            print(f"Slot {i}: {chain}")


# --- Driver Code ---
ht = ChainingHashTable(7)
for key in [10, 20, 15, 7, 3, 24]:
    ht.insert(key)
ht.display()
# Slot 0: [7]
# Slot 1: [15]
# Slot 2: [2]
# Slot 3: [10, 3]
# Slot 4: []
# Slot 5: []
# Slot 6: [20]
print("Search 10:", ht.search(10))    # True
ht.delete(10)
print("Search 10:", ht.search(10))    # False
```

6.2.2 Open Addressing

Concept: Open Addressing

In open addressing, all keys are stored directly in the hash table array itself. There are no linked lists. If a collision occurs at slot $h(k)$, we probe other slots in a specific sequence until we find an empty slot. The probing sequence is determined by a **probe function** $h(k, i)$, where i is the probe number (0, 1, 2, ...). We always have $\alpha < 1$ because every key needs its own slot.
The key rule: When searching for a key, we follow the same probe sequence. When we find an empty slot, we stop (key not present). When we find the key, we stop (found).

Linear Probing

Concept: Linear Probing

In linear probing, when a collision occurs at slot $h(k)$, we simply move to the next slot, and then the next, and so on, one step at a time.

$$h(k, i) = (h(k) + i) \bmod m, \quad i = 0, 1, 2, \ldots$$

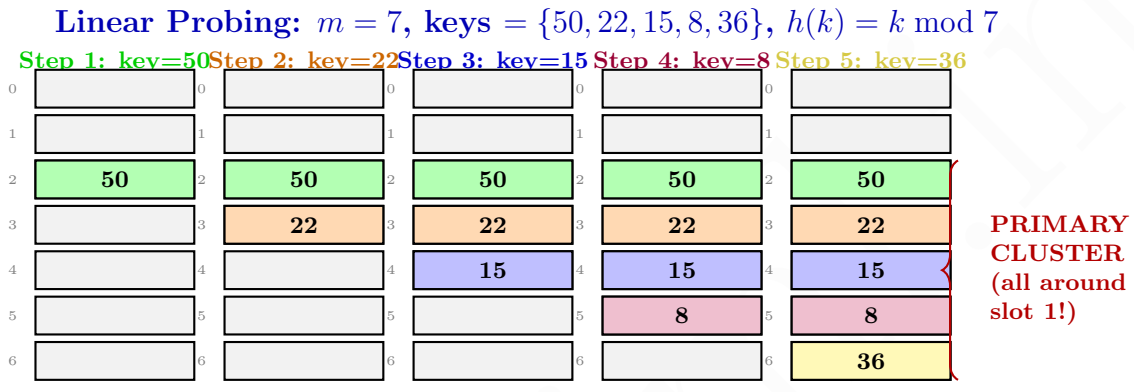
So if $h(k) = 3$ and slot 3 is occupied, we try slot 4, then 5, then 6, and so on, wrapping around if needed.
Analogy: Think of looking for a parking spot in a lot. You drove to your preferred spot (your hash slot). It is taken. So you look at the next spot, then the next, going one by one until you find a free one.

Example 250: Linear Probing: Step by Step

Table size $m = 7$. Hash function $h(k) = k \bmod 7$. Insert: $\{50, 22, 15, 8, 36\}$.

- $h(50) = 50 \bmod 7 = 1 \rightarrow$ slot 1 empty. Insert at slot 1.
- $h(22) = 22 \bmod 7 = 1 \rightarrow$ slot 1 occupied! Try slot 2. Empty. Insert at slot 2.
- $h(15) = 15 \bmod 7 = 1 \rightarrow$ slot 1 occupied! Try slot 2. Occupied! Try slot 3. Empty. Insert at slot 3.
- $h(8) = 8 \bmod 7 = 1 \rightarrow$ slots 1,2,3 occupied! Try slot 4. Empty. Insert at slot 4.
- $h(36) = 36 \bmod 7 = 1 \rightarrow$ slots 1,2,3,4 occupied! Try slot 5. Empty. Insert at slot 5.

Notice how all keys pile up around slot 1. This is called **primary clustering**.



Final table: slot 0 empty, slots 1-5 filled, slot 6 empty

All 5 keys hash to slot 1, but linear probing spreads them to slots 1, 2, 3, 4, 5

Concept: Primary Clustering

Primary clustering is the main problem with linear probing. When many keys hash to the same slot and create a cluster (a long continuous run of filled slots), future keys that hash anywhere in that cluster will have to probe past the entire cluster to find an empty slot. The cluster keeps growing, making future insertions and searches slower. This is a snowball effect: the larger the cluster, the more likely new keys land in it, making it even larger.

Quadratic Probing

Concept: Quadratic Probing

In quadratic probing, instead of moving one step at a time, we move by quadratic distances.

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m, \quad i = 0, 1, 2, \ldots$$

A common simple form is $h(k, i) = (h(k) + i^2) \bmod m$.
So if $h(k) = 3$: try slot 3, then $3 + 1 = 4$, then $3 + 4 = 7$, then $3 + 9 = 12$, and so on (all modulo m).
This spreads out keys much better than linear probing, reducing primary clustering. But quadratic probing can still cause **secondary clustering**: two keys with the same initial hash always follow the same probe sequence.

Example 251: Quadratic Probing: Step by Step

Table size $m = 11$. $h(k) = k \bmod 11$. Insert keys $\{18, 26, 35, 9\}$.

- $h(18) = 7 \rightarrow$ slot 7 empty. Insert at 7.
- $h(26) = 4 \rightarrow$ slot 4 empty. Insert at 4.
- $h(35) = 2 \rightarrow$ slot 2 empty. Insert at 2.
- $h(9) = 9 \rightarrow$ slot 9 empty. Insert at 9.

Now insert 44: $h(44) = 0 \to$ slot 0 empty. Insert at 0.

Now insert 55: $h(55) = 0 \to$ slot 0 occupied! Try $i = 1$: $(0 + 1) \bmod 11 = 1 \to$ empty. Insert at 1.

Now insert 66: $h(66) = 0 \to$ slot 0 occupied! Try $i = 1$: slot 1 occupied. Try $i = 2$: $(0 + 4) \bmod 11 = 4 \to$ occupied! Try $i = 3$: $(0 + 9) \bmod 11 = 9 \to$ occupied! Try $i = 4$: $(0 + 16) \bmod 11 = 5 \to$ empty. Insert at 5.

Concept: Secondary Clustering

Secondary clustering occurs in quadratic probing. Two different keys that hash to the same initial slot $h(k)$ will follow the exact same probe sequence. So they always compete for the same set of slots. However, two keys with different initial hashes never follow the same sequence. This is less severe than primary clustering but still an issue.

Double Hashing

Concept: Double Hashing

Double hashing uses a second hash function to determine the step size.

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m, \quad i = 0, 1, 2, \ldots$$

where h_1 and h_2 are two different hash functions. The step size $h_2(k)$ is different for different keys, so two keys that have the same $h_1(k)$ will likely have different $h_2(k)$ values, meaning they follow completely different probe sequences. This eliminates both primary and secondary clustering, and gives the best performance among open addressing methods.
Common choice: $h_1(k) = k \bmod m$ and $h_2(k) = 1 + (k \bmod (m - 1))$. We ensure $h_2(k) \neq 0$ so we actually move forward.

Example 252: Double Hashing: Step by Step

Table size $m = 7$. $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 6)$. Insert $\{18, 25, 35\}$.

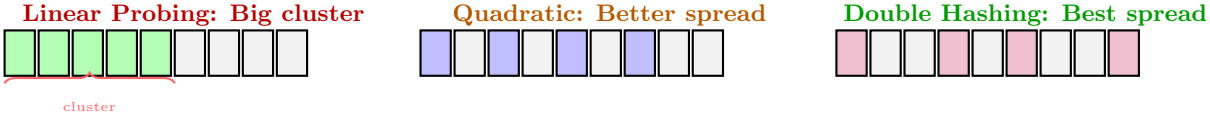
- Insert 18: $h_1(18) = 4 \to$ slot 4 empty. Insert at 4.
- Insert 25: $h_1(25) = 4 \to$ slot 4 occupied! $h_2(25) = 1 + 25 \bmod 6 = 1 + 1 = 2$. Try $(4 + 1 \times 2) \bmod 7 = 6 \to$ slot 6 empty. Insert at 6.
- Insert 35: $h_1(35) = 0 \to$ slot 0 empty. Insert at 0.

Now insert 11: $h_1(11) = 4 \to$ occupied. $h_2(11) = 1 + 11 \bmod 6 = 1 + 5 = 6$. Try $(4 + 6) \bmod 7 = 3 \to$ empty. Insert at 3. A different probe sequence from key 25 (even though both first went to slot 4).

Open Addressing: Comparison of Methods

Feature	Linear Probing	Quadratic Probing	Double Hashing
Probe formula	$(h(k) + i) \bmod m$	$(h(k) + i^2) \bmod m$	$(h_1(k) + i \cdot h_2(k)) \bmod m$
Primary clustering	**YES** (severe)	No	No
Secondary clustering	No	**YES** (mild)	No
Distribution	Poor	Better	Best
Cache performance	Best (sequential)	Medium	Worst (random)
Implementation	Simplest	Simple	Complex (2 funcs)

Cluster Pattern Comparison for Same Input



Python implementation: open-addressing-hash-table.py

```
class OpenAddressingHT:
    def __init__(self, m, method='linear'):
        self.m       = m
        self.table   = [None] * m
        self.deleted = [False] * m   # lazy deletion marker
        self.method  = method

    def h1(self, key):
        return key % self.m

    def h2(self, key):
        return 1 + (key % (self.m - 1))  # for double hashing

    def probe(self, key, i):
        if self.method == 'linear':
            return (self.h1(key) + i) % self.m
        elif self.method == 'quadratic':
            return (self.h1(key) + i*i) % self.m
        else:  # double hashing
            return (self.h1(key) + i * self.h2(key)) % self.m

    def insert(self, key):
        for i in range(self.m):
            slot = self.probe(key, i)
            if self.table[slot] is None or self.deleted[slot]:
                self.table[slot]   = key
                self.deleted[slot] = False
                return True
        print("Hash table is FULL")
        return False

    def search(self, key):
        for i in range(self.m):
            slot = self.probe(key, i)
            if self.table[slot] is None and not self.deleted[slot]:
                return False        # empty slot means key not present
            if self.table[slot] == key and not self.deleted[slot]:
                return True
        return False

    def delete(self, key):
        for i in range(self.m):
            slot = self.probe(key, i)
            if self.table[slot] is None and not self.deleted[slot]:
                return False
            if self.table[slot] == key and not self.deleted[slot]:
                self.deleted[slot] = True   # lazy deletion
                return True
        return False

    def display(self):
        for i, val in enumerate(self.table):
            status = "DEL" if self.deleted[i] else str(val)
            print(f"Slot {i}: {status}")


# --- Driver Code ---
print("=== Linear Probing ===")
ht = OpenAddressingHT(7, 'linear')
for key in [50, 22, 15, 8, 36]:
    ht.insert(key)
ht.display()

print("\n=== Double Hashing ===")
ht2 = OpenAddressingHT(7, 'double')
for key in [50, 22, 15, 8, 36]:
    ht2.insert(key)
```

```
ht2.display()
```

6.3 Performance Analysis

Concept: Expected Number of Probes

The performance of open addressing depends on the load factor α. Under the assumption of uniform hashing, the expected number of probes for:
Unsuccessful Search (looking for a key that is NOT in the table):

$$\text{Expected probes} \leq \frac{1}{1-\alpha}$$

Successful Search (looking for a key that IS in the table):

$$\text{Expected probes} \leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

These formulas assume open addressing with uniform probing. For chaining, the analysis is different.

Concept: Performance Analysis for Chaining

For chaining with load factor $\alpha = n/m$:
Unsuccessful Search: Expected number of elements examined = α (length of a randomly chosen chain), plus 1 for computing the hash. So total work is $\Theta(1+\alpha)$.
Successful Search: Expected work is $\Theta(1+\alpha)$.
Insert: $O(1)$ (insert at front of chain, plus computing hash).
Delete: $O(1+\alpha)$ (find the element first, then delete).
When $\alpha = O(1)$ (load factor kept constant by resizing), all operations are $O(1)$.

Expected Probes vs Load Factor α

α	Unsuccessful (OA)	Successful (OA)	Chaining (both)
0.10	1.11	1.05	≈ 1.10
0.25	1.33	1.15	≈ 1.25
0.50	2.00	1.39	≈ 1.50
0.75	4.00	1.85	≈ 1.75
0.90	10.00	2.56	≈ 1.90
0.95	20.00	3.15	≈ 1.95

OA = Open Addressing. At $\alpha = 0.9$: Unsuccessful search takes 10 probes on average!

Example 253: Numerical Example: Expected Probes

Consider a hash table with $m = 10$ slots and $n = 7$ keys inserted. So $\alpha = 7/10 = 0.7$.
Open Addressing:

- Expected probes for unsuccessful search $= \frac{1}{1-0.7} = \frac{1}{0.3} \approx 3.33$
- Expected probes for successful search $\approx \frac{1}{0.7} \ln \frac{1}{0.3} \approx 1.43 \times 1.20 \approx 1.72$

Chaining:

- Expected work $= \Theta(1+0.7) = \Theta(1.7)$, roughly 1.7 operations per search.

So at $\alpha = 0.7$, chaining is significantly more efficient than open addressing for unsuccessful search.

Concept: Effect of Load Factor on Performance

The load factor is the single most important tuning parameter of a hash table. Key observations:

- For $\alpha \leq 0.5$: open addressing is fast, with expected probes close to 2 or less.
- For $\alpha > 0.7$: open addressing degrades significantly.
- Chaining degrades gracefully because chains simply get longer, but the overhead is just the chain length.
- Real-world hash tables are resized (rehashed) when α crosses a threshold, typically 0.7 for open addressing and around 1.0 for chaining.
- Resizing means creating a new larger array and reinserting all keys. This costs $O(n)$ but happens infrequently, so amortized cost remains $O(1)$ per operation.

Hash Table Operations: Final Summary

Operation	Chaining	Open Addressing	Note
Insert	$O(1)$	$O(1)$ amortized	OA: finds empty slot
Search (succ)	$O(1+\alpha)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$	Both: avg case
Search (unsucc)	$O(1+\alpha)$	$O\left(\frac{1}{1-\alpha}\right)$	OA much worse at high α
Delete	$O(1+\alpha)$	$O(1)$ with lazy deletion	OA: mark deleted
Space	$O(n+m)$	$O(m)$	Chaining uses extra ptr
Max α	> 1 OK	< 1 required	OA needs free slot
Worst case	$O(n)$	$O(n)$	All keys same slot

Use Chaining when:
α is unknown or can exceed 1, frequent deletions

Use Open Addressing when:
Memory is tight, cache performance matters, $\alpha < 0.7$

Python implementation: hash-table-performance.py

```python
import math

def expected_probes_open_addressing(alpha):
    """Expected probes for open addressing (uniform hashing)"""
    if alpha >= 1:
        return float('inf')
    unsuccessful = 1 / (1 - alpha)
    successful   = (1 / alpha) * math.log(1 / (1 - alpha))
    return unsuccessful, successful


def expected_probes_chaining(alpha):
    """Expected probes for chaining"""
    unsuccessful = 1 + alpha
    successful   = 1 + alpha / 2
    return unsuccessful, successful


print(f"{'alpha':>6} | {'OA-Unsucc':>10} | {'OA-Succ':>9} | {'Chain-Unsucc':>12} | {'
    Chain-Succ':>10}")
print("-" * 65)
for alpha in [0.1, 0.25, 0.50, 0.75, 0.90, 0.95]:
    oa_u, oa_s   = expected_probes_open_addressing(alpha)
    ch_u, ch_s   = expected_probes_chaining(alpha)
    print(f"{alpha:>6.2f} | {oa_u:>10.2f} | {oa_s:>9.2f} | {ch_u:>12.2f} | {ch_s:>10.2f
        }")

# Output:
# alpha | OA-Unsucc |    OA-Succ | Chain-Unsucc | Chain-Succ
# -----------------------------------------------------------------
#   0.10 |       1.11 |       1.05 |          1.10 |       1.05
```

```
#  0.25 |      1.33 |      1.15 |         1.25 |       1.12
#  0.50 |      2.00 |      1.39 |         1.50 |       1.25
#  0.75 |      4.00 |      1.85 |         1.75 |       1.38
#  0.90 |     10.00 |      2.56 |         1.90 |       1.45
#  0.95 |     20.00 |      3.15 |         1.95 |       1.48
```

Example 254: Quick Summary: Which Method to Choose

- **Division method** for hash function: simple, works well if m is a prime not close to a power of 2.
- **Multiplication method**: works well for any m, including powers of 2.
- **Chaining**: easy to implement, handles $\alpha > 1$, graceful degradation, but uses extra memory for pointers.
- **Linear probing**: best cache performance (sequential access), but primary clustering makes it slow at high α.
- **Quadratic probing**: reduces primary clustering, but secondary clustering remains, and not all slots guaranteed to be probed.
- **Double hashing**: best performance among open addressing methods, avoids both types of clustering, but needs two hash functions.

Scaler Learning

6.4 Practice Problems

Problem 186 *[MCQ] A hash table of size $m = 13$ uses the division method $h(k) = k \bmod m$. Which of the following keys does not map to slot 5?*

(A) 57

(B) 44

(C) 31

(D) 19

Problem 187 *[NAT] A hash table has $m = 11$ slots and currently stores $n = 8$ keys. The load factor α, rounded to two decimal places, is _____.*

Problem 188 *[MCQ] A hash table uses the multiplication method with $A = 0.6180$ and table size $m = 16$. The hash value $h(100)$ is:*

(A) 12

(B) 14

(C) 9

(D) 7

Problem 189 *[MSQ] Which of the following are TRUE about the division method $h(k) = k \bmod m$?*

(A) A prime number not close to a power of 2 is a good choice for m.

(B) $m = 16$ is a good choice because $2^4 = 16$ makes computation fast.

(C) If m is a power of 2, the hash depends only on the lower-order bits of k.

(D) The method guarantees no collisions for any table size m.

Problem 190 *[MCQ] A hash table has 100 slots. The number of keys stored is n. For the expected number of probes in an unsuccessful search (using open addressing with uniform hashing) to be at most 4, the maximum value of n is:*

(A) 70

(B) 75

(C) 80

(D) 90

Problem 191 *[NAT] A hash table with open addressing uses uniform hashing. The load factor is $\alpha = 0.5$. The expected number of probes for an unsuccessful search is _____.*

Problem 192 *[MSQ] Consider a hash table with m slots, n stored keys, and load factor α. Which of the following are TRUE?*

(A) In chaining, α can exceed 1.

(B) In open addressing, α must be strictly less than 1.

(C) If $\alpha = 0$, the table is empty and every lookup takes $O(1)$.

(D) Doubling the table size halves the load factor exactly, regardless of the collision resolution method.

Problem 193 *[NAT] A hash table of size $m = 7$ uses the division method. Keys $\{14, 21, 28, 35\}$ are inserted. The number of distinct slots that are occupied after all insertions (ignoring collision resolution) is _____.*

Problem 194 *[MCQ] Consider a hash function $h(k) = k \bmod m$ where $m = 10$. If we want $h(k_1) = h(k_2)$ for $k_1 = 37$ and some $k_2 > 100$, the minimum integer value for k_2 is _____.*

Problem 195 *[NAT] A hash table with open addressing has $m = 20$ slots. The maximum number of keys that can be stored such that the load factor does not exceed 0.75 is _____.*

Problem 196 ***[MCQ]*** *A hash table of size $m = 5$ uses chaining and the hash function $h(k) = k \bmod 5$. Keys $\{3, 8, 13, 18, 23\}$ are inserted in this order. The length of the longest chain after all insertions is:*

(A) 3

(B) 4

(C) 5

(D) 2

Problem 197 ***[NAT]*** *A hash table of size $m = 7$ uses chaining and $h(k) = k \bmod 7$. The keys $\{15, 1, 22, 8, 29, 36, 43\}$ are inserted. The number of slots that have exactly one element after all insertions is _____.*

Problem 198 ***[MSQ]*** *A hash table uses separate chaining with the hash function $h(k) = k \bmod 6$. After inserting keys $\{6, 12, 18, 1, 7, 13\}$, which of the following are TRUE?*

(A) Slot 0 has a chain of length 3.

(B) Slot 1 has a chain of length 3.

(C) The load factor is exactly 1.

(D) Searching for key 13 requires examining at most 3 elements (including the one found).

Problem 199 ***[NAT]*** *A hash table of size $m = 6$ uses chaining and $h(k) = k \bmod 6$. Keys are inserted in the order $\{2, 8, 14, 20, 26, 32\}$. Assuming new keys are inserted at the **front** of the chain, searching for key 14 examines how many keys (including the found key) before returning? _____*

Problem 200 ***[MCQ]*** *In a hash table using chaining with n keys and m slots, the expected time for a successful search (assuming simple uniform hashing and insertion at the front of chains) is:*

(A) $\Theta(1 + \alpha)$

(B) $\Theta(\alpha)$

(C) $\Theta(\alpha^2)$

(D) $\Theta(n)$

Problem 201 ***[MSQ]*** *A professor claims: "For a hash table with chaining, if all n keys hash to the same slot, search takes $O(n)$; but if we use a balanced BST instead of a linked list for each chain, worst-case search becomes $O(\log n)$." Which of the following are TRUE?*

(A) The professor's claim about linked list worst case is correct.

(B) Using a BST per chain reduces the worst-case search time to $O(\log n)$.

(C) The average-case search time for chaining with linked lists is $O(1)$ when $\alpha = O(1)$.

(D) Replacing linked lists with BSTs eliminates the need for a good hash function.

Problem 202 ***[MCQ]*** *A hash table of size $m = 7$ uses linear probing with $h(k) = k \bmod 7$. Keys are inserted in order: $\{50, 700, 76, 85, 92, 73\}$. After all insertions, the key 76 is stored at slot:*

(A) 6

(B) 0

(C) 1

(D) 5

Problem 203 ***[NAT]*** *A hash table of size $m = 10$ uses linear probing with $h(k, i) = (k \bmod 10 + i) \bmod 10$. Keys are inserted in order: $\{72, 27, 36, 24, 63, 81\}$. How many collisions (probes that find an occupied slot) occur in total during all insertions? _____*

Problem 204 ***[MSQ]*** *Consider a hash table of size $m = 11$ using linear probing with $h(k) = k \bmod 11$. After inserting keys $\{10, 22, 31, 4, 15, 28, 17, 88, 59\}$, which of the following are TRUE?*

(A) Primary clustering has occurred.

(B) The key 88 is stored at slot 0.

(C) Searching for key 59 requires more than 1 probe.

(D) All keys with the same $h(k)$ are guaranteed to be in adjacent slots.

Problem 205 ***[MCQ]*** *In linear probing, when a key is deleted, we cannot simply mark the slot as empty. The correct approach is:*

(A) Delete the key and mark the slot as a special "DELETED" marker that counts as occupied for search but empty for insertion.

(B) Delete the key and leave the slot empty, as future searches will stop at the next occupied slot.

(C) Delete the key and shift all subsequent keys in the cluster backward by one slot.

(D) Delete the key and rehash the entire table immediately.

Problem 206 ***[MSQ]*** *Consider the algorithm for linear probing insertion:*

Algorithm: LP-Insert(T, m, key)

```
1: i ← 0
2: repeat
3:     j ← (h(key) + i) mod m
4:     if T[j] = NULL or T[j] = DELETED then
5:         T[j] ← key
6:         return
7:     end if
8:     i ← i + 1
9: until i = m
10: error "Hash table overflow"
```

Which of the following are TRUE about this algorithm?

(A) A DELETED marker is needed to correctly handle keys inserted after a deletion.

(B) The algorithm probes at most m slots before reporting overflow.

(C) If the table has one empty slot, insertion always succeeds.

(D) Insertions into DELETED slots help reduce average search time over time.

Problem 207 ***[MCQ]*** *A hash table of size $m = 11$ uses quadratic probing with $h(k, i) = (h(k) + i^2) \bmod m$ where $h(k) = k \bmod 11$. Key 35 has $h(35) = 2$. Slot 2 and slot 3 are occupied. The next probe is at slot:*

(A) 4

(B) 6

(C) 5

(D) 7

Problem 208 ***[NAT]*** *A hash table of size $m = 11$ uses quadratic probing $h(k, i) = (k \bmod 11 + i^2) \bmod 11$. Key 20 has $h_1(20) = 9$. Slots 9 and 10 are occupied. The slot where key 20 is finally inserted is _____.*

Problem 209 ***[MCQ]*** *Secondary clustering in quadratic probing means:*

(A) Keys that hash to the same slot always follow the same probe sequence.

(B) Keys that hash to different slots can collide in the probe sequence.

(C) Every insertion causes at least one collision.

(D) The probe sequence visits all m slots exactly once.

Problem 210 ***[MSQ]*** *A hash table of size $m = 13$ (a prime) uses quadratic probing with $h(k, i) = (h_1(k) + i^2) \bmod 13$. Which of the following are TRUE?*

(A) The probe sequence for two different keys with the same initial hash $h_1(k)$ is identical.

(B) If m is prime and the table is less than half full, quadratic probing always finds an empty slot.

(C) Quadratic probing completely eliminates primary clustering.

(D) Quadratic probing guarantees that all m slots are probed before revisiting any slot.

Problem 211 ***[MCQ]*** *A hash table of size $m = 11$ uses double hashing with $h_1(k) = k \bmod 11$ and $h_2(k) = 1 + (k \bmod 10)$. The probe sequence for key 21 starts at slot $h_1(21) = 10$. If slot 10 is occupied, the next slot probed is:*

(A) 2

(B) 0

(C) 1

(D) 3

Problem 212 ***[NAT]*** *A hash table of size $m = 7$ uses double hashing: $h_1(k) = k \bmod 7$ and $h_2(k) = 1 + (k \bmod 6)$. Keys are inserted in order: $\{18, 41, 22, 44, 59\}$. The slot where key 44 is stored after insertion is _____.*

Problem 213 ***[MSQ]*** *Which of the following are TRUE about double hashing compared to linear and quadratic probing?*

(A) Double hashing avoids both primary and secondary clustering.

(B) Two keys with the same $h_1(k)$ but different $h_2(k)$ will follow different probe sequences.

(C) Double hashing guarantees $O(1)$ worst-case insertion.

(D) If $h_2(k) = 0$ for some key k, the probe never advances and causes an infinite loop.

Problem 214 ***[MCQ]*** *In double hashing, the second hash function $h_2(k)$ must satisfy which condition to ensure all slots are eventually probed?*

(A) $h_2(k)$ must be equal to $h_1(k)$ for all k

(B) $h_2(k)$ must be relatively prime to m for all k

(C) $h_2(k)$ must always return an even number

(D) $h_2(k)$ must always return a value less than $m/2$

Problem 215 ***[NAT]*** *A hash table of size $m = 11$ uses double hashing: $h_1(k) = k \bmod 11$, $h_2(k) = 7 - (k \bmod 7)$. Key 58 has $h_1(58) = 3$ and $h_2(58) = 7 - (58 \bmod 7) = 7 - 2 = 5$. If slots 3 and 8 are occupied, the third probe slot (for $i = 2$) is _____.*

Problem 216 ***[NAT]*** *A hash table with $m = 10$ and $n = 5$ uses chaining. Under the simple uniform hashing assumption, the expected number of elements examined in a successful search is _____.*

Problem 217 ***[MSQ]*** *Under simple uniform hashing, which of the following are TRUE about the expected number of probes for open addressing?*

(A) For unsuccessful search, expected probes $= \dfrac{1}{1-\alpha}$.

(B) For successful search, expected probes $\approx \dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$.

(C) At $\alpha = 0.5$, unsuccessful search takes about 2 probes on average.

(D) At $\alpha = 0.9$, unsuccessful search takes about 5 probes on average.

Problem 218 ***[MCQ]*** *A hash table uses linear probing and has been filled up to $\alpha = 0.9$. We now rehash into a new table twice the size. The new load factor immediately after rehashing is:*

(A) 0.45

(B) 0.9

(C) 0.18

(D) 0.5

Problem 219 ***[MCQ]*** *A hash table of size m uses chaining. The hash function distributes n keys such that exactly $n/2$ keys map to slot 0 and the remaining $n/2$ keys are distributed uniformly among the remaining $m-1$ slots. The expected number of comparisons to find a key chosen uniformly at random is approximately:*

(A) $O(1)$

(B) $O(n/m)$

(C) $O(n)$

(D) $O(n/4)$

Problem 220 ***[MCQ]*** *Two hash tables H_1 (chaining) and H_2 (linear probing) both have $m = 100$ slots with $n = 70$ keys. A new key is inserted. The expected number of probes for this insertion is P_1 for H_1 and P_2 for H_2. Which of the following is TRUE?*

(A) $P_1 > P_2$

(B) $P_1 = P_2 = 1$

(C) $P_1 < P_2$

(D) $P_1 = P_2 \approx 3.3$

Problem 221 ***[MCQ]*** *Consider a hash table with open addressing. After a sequence of insertions and deletions, the table has the following state (LP = linear probing, $m = 7$, $h(k) = k \bmod 7$):*

Slot 0: DEL, Slot 1: 15, Slot 2: DEL, Slot 3: DEL, Slot 4: 4, Slot 5: empty, Slot 6: empty

We now insert key 22 ($h(22) = 1$). The key 22 is stored at slot:

(A) 0 (first DELETED slot encountered)

(B) 2 (second DELETED slot)

(C) 1 (first probe, occupied, skip to next)

(D) 5 (first truly empty slot)

Problem 222 ***[MCQ]*** *Consider a hash table of size $m = 9$ using quadratic probing $h(k, i) = (k \bmod 9 + i^2) \bmod 9$. Starting from initial hash position 0, the probe sequence visits slots:*

(A) 0, 1, 4, 9, 16, ... (not all taken modulo 9)

(B) 0, 1, 4, 0, 7, 7, ... (modulo 9 — note revisit at 0)

(C) 0, 1, 4, 0, 5, 0, 8, 4, 1 (all 9 probes)

(D) 0, 1, 2, 3, 4, 5, 6, 7, 8 (linear)

Problem 223 ***[MCQ]*** *A hash table of size m uses double hashing with $h_1(k) = k \bmod m$ and $h_2(k) = 1+(k \bmod (m-1))$. For this to guarantee probing all m slots when m is prime, which condition is required?*

(A) m must be even

(B) m must be prime (so that any $h_2(k)$ in $[1, m-1]$ is coprime with m)

(C) $h_2(k)$ must equal $h_1(k)$ for all k

(D) m must be a power of 2

Problem 224 ***[MSQ]*** *A hash table with linear probing has the property that if $\alpha > 0$, the expected number of probes for a successful search is less than the expected number of probes for an unsuccessful search. Which of the following correctly explain why?*

(A) A successful search stops as soon as the key is found, which could be early in the probe sequence.

(B) An unsuccessful search must probe until an empty slot (or full table scan), guaranteed to be longer.

(C) Both successful and unsuccessful search always take the same number of probes.

(D) Keys inserted earlier tend to have shorter probe distances than keys inserted later (due to clustering), so successful searches find early-inserted keys faster.

Problem 225 ***[NAT]*** *A hash table of size* $m = 13$ *uses linear probing with* $h(k) = k \bmod 13$*. The table is initially empty, and the following five keys are inserted in some order :*

Key	$h(k)$
17	*4*
30	*4*
44	*5*
19	*6*
32	*6*

The number of insertion orders for which key 44 *ends up stored in slot* 8 *is* _____.

Problem 226 ***[NAT]*** *A hash table of size* $m = 23$ *uses linear probing with* $h(k) = k \bmod 23$*. The table is initially empty, and the following five keys are inserted in some order :*

Key	$h(k)$
33	*10*
56	*10*
34	*11*
35	*12*
58	*12*

The number of insertion orders for which key 34 *does* not *end up stored in its own home slot* 11 *is* _____.

Problem 227 ***[MCQ]*** *A hash table of size* $m = 11$ *uses linear probing with* $h(k) = k \bmod 11$*. Starting from an empty table, the keys* 25*,* 47*, and* 26 *were inserted in some order, resulting in the following table:*

Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Key	–	–	–	*47*	*26*	*25*	–	–	–	–	–

Which of the following could be the order in which the three keys were inserted to produce the table above?

(A) $25, 47, 26$

(B) $47, 26, 25$

(C) $25, 26, 47$

(D) $26, 25, 47$

Problem 228 ***[NAT]*** *A hash table of size* $m = 11$ *uses linear probing with* $h(k) = k \bmod 11$*. Starting from an empty table, the keys* 25*,* 47*, and* 26 *were inserted in some order, resulting in the following table:*

Slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Key	–	–	–	*47*	*26*	*25*	–	–	–	–	–

Considering all possible orders in which the three keys 25*,* 47*,* 26 *could have been inserted into the initially empty table, the number of insertion orders that result in the exact table shown above is* _____.

6.5 GATE PYQs

GATEPYQ 149 (GATE CSE 2026) *The keys* $5, 28, 19, 15, 26, 33, 12, 17, 10$ *are inserted into a hash table using the hash function* $h(k) = k \bmod 9$. *The collisions are resolved by chaining. After all the keys are inserted, the length of the longest chain is* _____. *(Answer in integer)*

GATEPYQ 150 (GATE CSE 2026) *Consider a hash table* $P[0, 1, \ldots, 10]$ *that is initially empty. The hash table is maintained using open addressing with linear probing. The hash function used is* $h(x) = (x + 7) \bmod 11$.
Consider the following sequence of insertions performed on P*:*

$1, 13, 22, 15, 11, 24$

Which of the following positions in the hash table is/are empty after these insertions are performed?

A. *0*

B. *10*

C. *2*

D. *1*

GATEPYQ 151 (GATE CSE 2025) *In a double hashing scheme,* $h_1(k) = k \bmod 11$ *and* $h_2(k) = 1 + (k \bmod 7)$ *are the auxiliary hash functions. The size* m *of the hash table is 11. The hash function for the* i*-th probe in the open address table is* $[h_1(k) + i \cdot h_2(k)] \bmod m$. *The following keys are inserted in the given order: 63, 50, 25, 79, 67, 24.*
The slot at which key 24 gets stored is _____. *(Answer in integer)*

GATEPYQ 152 (GATE DA 2025) *Consider a hash table of size 10 with indices* $\{0, 1, \ldots, 9\}$*, with the hash function* $h(x) = 3x \pmod{10}$*, where linear probing is used to handle collisions. The hash table is initially empty and then the following sequence of keys is inserted into the hash table:*

$1, 4, 5, 6, 14, 15$

The indices where the keys 14 *and* 15 *are stored are, respectively:*

A. *2 and 5*

B. *2 and 6*

C. *4 and 5*

D. *4 and 6*

GATEPYQ 153 (GATE DA 2024) *Match the items in Column 1 with the items in Column 2 in the following table:*

Column 1		*Column 2*	
(p)	*First In First Out*	*(i)*	*Stacks*
(q)	*Lookup Operation*	*(ii)*	*Queues*
(r)	*Last In First Out*	*(iii)*	*Hash Tables*

A. *(p) - (ii), (q) - (iii), (r) - (i)*

B. *(p) - (ii), (q) - (i), (r) - (iii)*

C. *(p) - (i), (q) - (ii), (r) - (iii)*

D. *(p) - (i), (q) - (iii), (r) - (ii)*

GATEPYQ 154 (GATE DA 2024) *Consider performing uniform hashing on an open address hash table with load factor* $\alpha = \dfrac{n}{m} < 1$*, where* n *elements are stored in the table with* m *slots. The expected number of probes in an unsuccessful search is at most* $\dfrac{1}{1-\alpha}$.
Inserting an element in this hash table requires at most _____ *probes, on average.*

A. $\ln\left(\dfrac{1}{1-\alpha}\right)$

B. $\dfrac{1}{1-\alpha}$

C. $1 + \frac{\alpha}{2}$

D. $\frac{1}{1+\alpha}$

GATEPYQ 155 (GATE CSE 2023) *An algorithm has to store several keys generated by an adversary in a hash table. The adversary is malicious who tries to maximize the number of collisions. Let k be the number of keys, m be the number of slots in the hash table, and $k > m$. Which one of the following is the best hashing strategy to counteract the adversary?*

A. *Division method, i.e., use the hash function $h(k) = k \bmod m$.*

B. *Multiplication method, i.e., use the hash function $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, where A is a carefully chosen constant.*

C. *Universal hashing method.*

D. *If k is a prime number, use Division method. Otherwise, use Multiplication method.*

GATEPYQ 156 (GATE CSE 2022) *Suppose we are given n keys, m hash table slots, and two simple uniform hash functions h_1 and h_2. Further suppose our hashing scheme uses h_1 for the odd keys and h_2 for the even keys. What is the expected number of keys in a slot?*

A. $\frac{n}{m}$

B. $\frac{m}{n}$

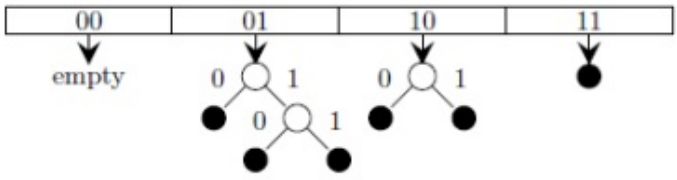
C. $\frac{2n}{m}$

D. $\frac{n}{2m}$

GATEPYQ 157 (GATE CSE 2021) *Consider a dynamic hashing approach for 4-bit integer keys:*

1. *There is a main hash table of size 4.*
2. *The 2 least significant bits of a key is used to index into the main hash table.*
3. *Initially, the main hash table entries are empty.*
4. *Thereafter, when more keys are hashed into it, to resolve collisions, the set of all keys corresponding to a main hash table entry is organized as a binary tree that grows on demand.*
5. *First, the 3rd least significant bit is used to divide the keys into left and right subtrees.*
6. *To resolve more collisions, each node of the binary tree is further sub-divided into left and right subtrees based on the 4th least significant bit.*
7. *A split is done only if it is needed, i.e., only when there is a collision.*

Consider the following state of the hash table.



Which of the following sequences of key insertions can cause the above state of the hash table (assume the keys are in decimal notation)?

A. *5, 9, 4, 13, 10, 7*

B. *9, 5, 10, 6, 7, 1*

C. *10, 9, 6, 7, 5, 13*

D. *9, 5, 13, 6, 10, 14*

GATEPYQ 158 (GATE CSE 2020) *Consider a double hashing scheme in which the primary hash function is $h_1(k) = k \bmod 23$, and the secondary hash function is $h_2(k) = 1 + (k \bmod 19)$. Assume that the table size is 23. Then the address returned by probe 1 in the probe sequence (assume that the probe sequence begins at probe 0) for key value $k = 90$ is* _____.

GATEPYQ 159 (GATE CSE 2015 SET-3) *Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is _____.*

GATEPYQ 160 (GATE CSE 2015) *Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?*

A. $h(i) = i^2 \bmod 10$

B. $h(i) = i^3 \bmod 10$

C. $h(i) = (11 \cdot i^2) \bmod 10$

D. $h(i) = (12 \cdot i^2) \bmod 10$

GATEPYQ 161 (GATE CSE 2014 SET-3) *Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?*

A. $(97 \times 97 \times 97)/100^3$

B. $(99 \times 98 \times 97)/100^3$

C. $(97 \times 96 \times 95)/100^3$

D. $(97 \times 96 \times 95)/(3! \times 100^3)$

GATEPYQ 162 (GATE CSE 2014) *Consider a hash table with 9 slots. The hash function is $h(k) = k \bmod 9$. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are:*

A. 3, 0, and 1

B. 3, 3, and 3

C. 4, 0, and 1

D. 3, 0, and 2

GATEPYQ 163 (GATE CSE 2010) *A hash table of length 10 uses open addressing with hash function $h(k) = k \bmod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below:*

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

A. 46, 42, 34, 52, 23, 33

B. 34, 42, 23, 52, 33, 46

C. 46, 34, 42, 23, 52, 33

D. 42, 46, 33, 23, 34, 52

GATEPYQ 164 (GATE CSE 2010) *A hash table of length 10 uses open addressing with hash function $h(k) = k \bmod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below:*

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above?

A. 10

B. 20

C. 30

D. 40

GATEPYQ 165 (GATE CSE 2009) *The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function* $h(k) = k \bmod 10$ *and linear probing. What is the resultant hash table?*

(A)

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(C)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(D)

0	
1	
2	12,2
3	13,3,23
4	
5	5,15
6	
7	
8	18
9	

A. A

B. B

C. C

D. D

GATEPYQ 166 (GATE IT 2008) *Consider a hash table of size 11 that uses open addressing with linear probing. Let* $h(k) = k \bmod 11$ *be the hash function used. A sequence of records with keys* $43, 36, 92, 87, 11, 4, 71, 13, 14$ *is inserted into an initially empty hash table, the bins of which are indexed from zero to ten. What is the index of the bin into which the last record is inserted?*

A. 3

B. 4

C. 6

D. 7

GATEPYQ 167 (GATE IT 2007) *Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5?*

A. 5

B. 6

C. 7

D. 10

GATEPYQ 168 (GATE CSE 2007) *Consider a hash table of size seven, with starting index zero, and a hash function* $(3x + 4) \bmod 7$. *Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that '-' denotes an empty location in the table.*

A. 8, -, -, -, -, -, 10

B. 1, 8, 10, -, -, -, 3

C. 1, -, -, -, -, -, 3

D. 1, 10, 8, -, -, -, 3

GATEPYQ 169 (GATE IT 2005) *A hash table contains 10 buckets and uses linear probing to resolve collisions. The key values are integers and the hash function used is key%10. If the values 43, 165, 62, 123, 142 are inserted in the table, in what location would the key value 142 be inserted?*

A. 2

B. 3

C. 4

D. 6

GATEPYQ 170 (GATE CSE 2004) *Given the following input* $(4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199)$ *and the hash function* $x \bmod 10$*, which of the following statements are true?*

i) 9679, 1989, 4199 hash to the same value

ii) 1471, 6171 hash to the same value

iii) All elements hash to the same value

iv) Each element hashes to a different value

A. i only

B. ii only

C. i and ii only

D. iii or iv

GATEPYQ 171 (GATE CSE 1996) *An advantage of chained hash table (external hashing) over the open addressing scheme is:*

A. Worst case complexity of search operations is less

B. Space used is less

C. Deletion is easier

D. None of the above

GATEPYQ 172 (GATE CSE 1989) *A hash table with ten buckets with one slot per bucket is shown in the following figure. The symbols* S_1 *to* S_7 *initially entered using a hashing function with linear probing. The maximum number of comparisons needed in searching an item that is not present is:*

0	S_7
1	S_1
2	
3	S_4
4	S_2
5	
6	S_5
7	
8	S_6
9	S_3

A. 4

B. 5

C. 6

D. 3

6.6 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
51	Introduction to Hashing & Load Factor	https://www.youtube.com/watch?v=-1YLhVu9hwA	
52	Collision Resolution: Separate Chaining | Hashing	https://www.youtube.com/watch?v=PdpeoIfrfH0	
53	Collision Resolution: Open Addressing & Linear Probing | Hashing	https://www.youtube.com/watch?v=X2s250e1-RM	
54	Collision Resolution: Open Addressing & Quadratic Probing	https://www.youtube.com/watch?v=TQ0Nbi7d4Ys	
55	Collision Resolution: Open Addressing & Double Hashing | Hashing	https://www.youtube.com/watch?v=71VsFATFYuA	
56	Performance Analysis of Open Addressing & Separate Chaining	https://www.youtube.com/watch?v=H49osNQUhlU	
57	GATE PYQs on Hashing	https://www.youtube.com/watch?v=5NSSSQGiAxg	
58	Practice Problems on Hashing	https://www.youtube.com/watch?v=Yt64C-ZypZI	

Chapter 7

Graph Theory — Introduction

Concept: What is a Graph

A graph is a mathematical structure used to model relationships between objects. The objects are called **vertices** (or nodes), and the relationships between them are called **edges**. A graph G is written as $G = (V, E)$, where V is the set of vertices and E is the set of edges.
Think of a graph as a map of cities connected by roads. Each city is a vertex. Each road connecting two cities is an edge. Using this map, we can answer questions like, "Is there a path from city A to city B?" or "What is the shortest route?"

Example 255: Real World Examples of Graphs

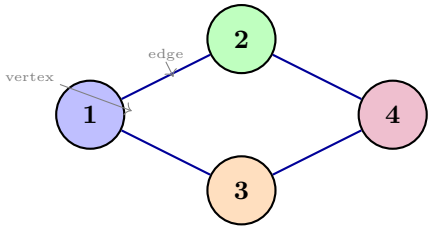
- **Social Network:** People are vertices. A friendship is an edge between two people.
- **Road Network:** Intersections are vertices. Roads are edges.
- **Internet:** Computers are vertices. Network cables or wireless links are edges.
- **Course Prerequisites:** Courses are vertices. An edge from course A to B means A must be done before B.

7.1 Basic Definitions

7.1.1 Vertex, Edge, and Graph $G = (V, E)$

Concept: Vertex and Edge

A **vertex** (plural: vertices) is a single point or node in a graph. A **vertex set** V is the collection of all vertices. We write $|V|$ to mean the total number of vertices.
An **edge** connects two vertices. An **edge set** E is the collection of all edges. Each edge is written as a pair (u, v), meaning there is a connection between vertex u and vertex v.
For example, in the graph $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (2, 4), (3, 4)\}$, there are 4 vertices and 4 edges.



$V = \{1, 2, 3, 4\}$ $E = \{(1, 2), (1, 3), (2, 4), (3, 4)\}$

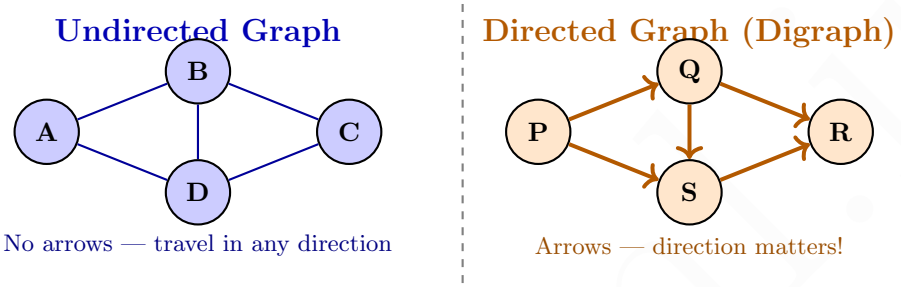
7.1.2 Directed vs. Undirected Graph

Concept: Undirected Graph

In an **undirected graph**, edges have no direction. If there is an edge between u and v, you can go from u to v AND from v to u. The edge (u, v) and (v, u) are the same edge. Think of a two-way road between two cities, you can drive in either direction.

Concept: Directed Graph (Digraph)

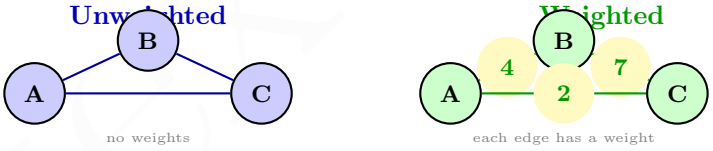
In a **directed graph** (also called a **digraph**), every edge has a direction, shown by an arrow. Edge (u, v) means you can go from u to v, but NOT necessarily from v to u. Think of a one-way road: you can only drive in one direction.



7.1.3 Weighted vs. Unweighted Graph

Concept: Weighted vs. Unweighted

In an **unweighted graph**, all edges are equal. We only care if an edge exists or not. In a **weighted graph**, each edge has a number called a **weight** (or cost). This weight can represent distance, time, cost, or capacity. For example, in a road map, the weight on each edge might be the distance in kilometres between two cities.



7.1.4 Degree, In-Degree, and Out-Degree

Concept: Degree in Undirected Graph

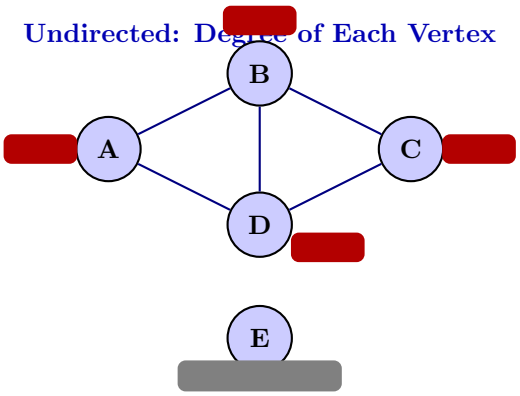
The **degree** of a vertex v in an undirected graph is the number of edges connected to v. It is written as $\deg(v)$. A vertex with degree 0 is called an **isolated vertex**. A vertex with degree 1 is called a **leaf** or **pendant vertex**.

Concept: In-Degree and Out-Degree in Directed Graph

In a directed graph, each vertex v has two degree values:

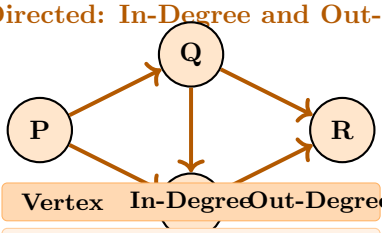
- **In-degree** ($\deg^-(v)$): number of edges coming *into* v (arrows pointing to v).
- **Out-degree** ($\deg^+(v)$): number of edges going *out of* v (arrows pointing away from v).

Undirected: Degree of Each Vertex



Sum of degrees $= 2+3+2+3+0 = 10 = 2 \times |E| = 2 \times 5$

Directed: In-Degree and Out-Degree



Vertex	In-Degree	Out-Degree
P	$\deg^- = 0$	$\deg^+ = 2$
Q	$\deg^- = 1$	$\deg^+ = 2$
R	$\deg^- = 2$	$\deg^+ = 0$
S	$\deg^- = 2$	$\deg^+ = 1$

7.1.5 Handshaking Lemma

Concept: Handshaking Lemma

The **Handshaking Lemma** states that for any undirected graph $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2|E|$$

The sum of degrees of all vertices equals twice the number of edges. This is because every edge contributes exactly 2 to the total degree sum: one for each of its two endpoints. Counting degrees is like counting handshakes at a party: each handshake is counted once by each participant, so total counts = twice the handshakes.
Useful consequence: The number of vertices with odd degree is always even in any graph.

Example 256: Verifying the Handshaking Lemma

In the undirected graph above with vertices A(deg=2), B(deg=3), C(deg=2), D(deg=3), E(deg=0):

$$\sum \deg(v) = 2+3+2+3+0 = 10 = 2 \times 5 = 2|E|$$

The graph has 5 edges. Twice 5 is 10. Confirmed.
Also, vertices with odd degree are B and D. That is 2 vertices, an even count. This is always true.

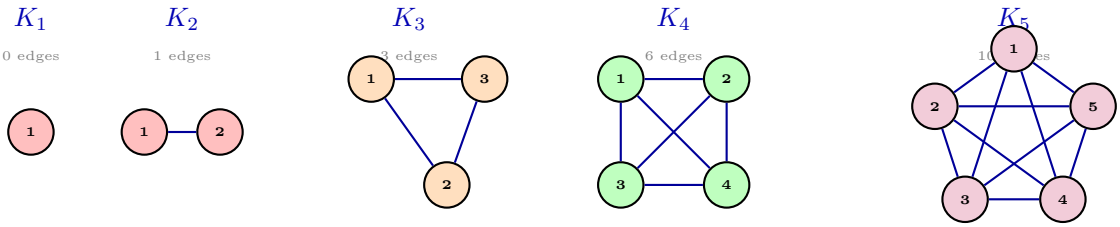
7.1.6 Complete Graph K_n

Concept: Complete Graph K_n

A **complete graph** K_n is an undirected graph with n vertices where every pair of distinct vertices is connected by exactly one edge. There is an edge between every possible pair.
The number of edges in K_n is

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}$$

because we choose 2 vertices from n vertices for each edge. Also, every vertex in K_n has degree $n-1$, since each vertex connects to all other $n-1$ vertices.

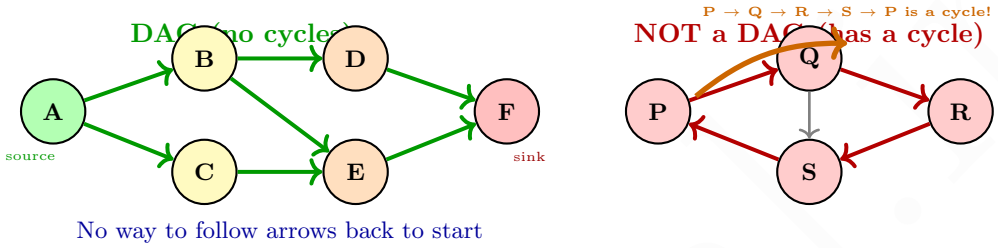


$|E(K_n)| = \frac{n(n-1)}{2}$: K_3: 3 edges, K_4: 6 edges, K_5: 10 edges

7.1.7 Directed Acyclic Graph (DAG)

Concept: DAG — Directed Acyclic Graph

A **DAG** is a directed graph that has **no directed cycles**. That means, starting from any vertex, if you follow directed edges, you can never come back to the starting vertex. DAGs are used to represent dependencies, like course prerequisites, build systems, or scheduling tasks where some tasks must be done before others.
Key property: A DAG always has at least one vertex with in-degree 0 (no one points to it), called a **source**, and at least one vertex with out-degree 0 (it points to no one), called a **sink**.



7.1.8 Tree as a Special Graph

Concept: Tree as a Graph

A **tree** is a special graph that satisfies all three of these properties simultaneously:

- **Undirected:** edges have no direction.
- **Connected:** there is a path between every pair of vertices.
- **Acyclic:** there are no cycles.

From these three properties, a very important relationship follows automatically:

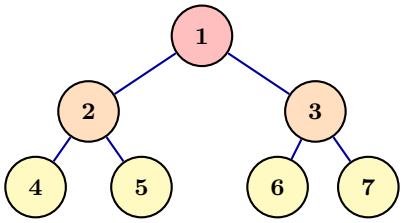
$$|E| = |V| - 1$$

A tree with n vertices always has exactly $n - 1$ edges. Remove any one edge and the tree becomes disconnected. Add any one edge and you create exactly one cycle.

Example 257: Why $|E| = |V| - 1$

Start with a single vertex (0 edges). Each time we add a new vertex to a tree, we must add exactly 1 edge to connect it (adding more than one edge would either create a cycle or disconnect something). So, after adding $n-1$ vertices to the first vertex, we have added $n - 1$ edges total. Hence $|E| = |V| - 1 = n - 1$.

Tree: Connected + Acyclic, $|E| = |V| - 1$



$|V| = 7$, $|E| = 6 = 7 - 1$ ✓

Tree Properties

Connected:	Yes — path between every pair
Acyclic:	Yes — no cycles
$\lvert E\rvert = \lvert V\rvert - 1$**:**	Always true for trees
Remove 1 edge:	Becomes disconnected
Add 1 edge:	Creates exactly 1 cycle

7.1.9 Connected Graph and Connected Components

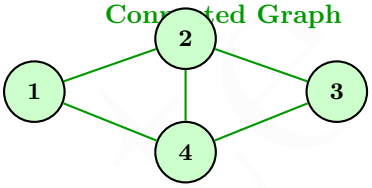
Concept: Connected Graph

An **undirected graph** is called **connected** if there is a path between every pair of vertices. That means you can reach any vertex from any other vertex by following edges.
A graph that is NOT connected is called **disconnected**. In a disconnected graph, there are groups of vertices that are isolated from each other. Each such group is called a **connected component**.
Formally, a connected component is a **maximal** set of vertices such that every pair of vertices in the set is connected by some path, and no vertex outside the set is connected to any vertex inside it.

Example 258: Connected vs Disconnected

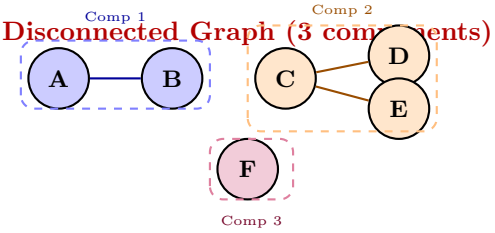
Think of a telephone network. If every phone can call every other phone (directly or through relays), the network is connected. If some phones are cut off from the rest, those isolated phones form a separate component.

Connected Graph



1 component, all reachable

Disconnected Graph (3 components)



A, B cannot reach C, D, E, F

Theorem: Minimum Edges for a Connected Graph

A connected undirected graph with n vertices needs **at least** $n - 1$ edges. This minimum is achieved exactly when the graph is a tree.

connected graph: $|E| \geq n - 1$

Why: To connect n vertices into a single group, we need enough edges to link every vertex, directly or indirectly, to every other. The most economical way (using the fewest edges) is a tree structure with exactly $n - 1$ edges. Fewer than $n - 1$ edges can never connect n vertices, because each edge can merge at most 2 previously separate components into 1, so we need at least $n - 1$ merging steps.

Theorem: Edges and Components (General Forest Formula)

For an undirected graph with n vertices, m edges, and c connected components, IF the graph is acyclic (a forest):

$$m = n - c$$

This generalizes the tree formula ($|E| = |V| - 1$, i.e., $c = 1$) to the case of multiple components. Each component, being a tree, contributes (its own vertices -1) edges, and summing over all components gives $n - c$ total edges.
If the graph is **not** necessarily acyclic, then instead we only know:

$$m \geq n - c$$

with equality holding if and only if every component is a tree (no cycles anywhere).

Example 259: Applying the Component Formula

A forest has $n = 20$ vertices and $c = 5$ connected components (5 separate trees). The total number of edges is:

$$m = n - c = 20 - 5 = 15$$

Now suppose instead we are told a general (not necessarily acyclic) graph has $n = 20$ vertices, $c = 5$ components, and $m = 18$ edges. Since $m = 18 > n - c = 15$, this graph has $18 - 15 = 3$ "extra" edges, meaning at least one component contains a cycle.

Connected Components: Formula Summary

Condition	Formula
Connected graph (tree, $c = 1$)	$m = n - 1$
Forest with c components (acyclic)	$m = n - c$
General graph with c components	$m \geq n - c$
Connected graph, has a cycle	$m \geq n$ (**strictly** $> n - 1$)

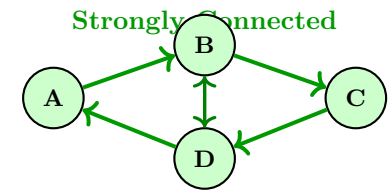
7.1.10 Strongly Connected and Weakly Connected (Directed Graphs)

Concept: Strongly Connected

A **directed graph** is called **strongly connected** if for every pair of vertices u and v, there is a directed path from u to v *and* a directed path from v to u. In simple words, you can reach every vertex from every other vertex by following the directed edges.
A **Strongly Connected Component (SCC)** is a maximal subset of vertices that are all mutually reachable from each other.

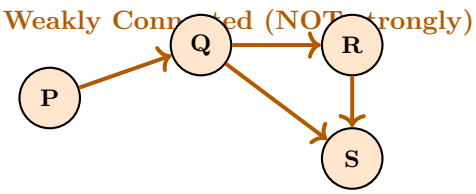
Concept: Weakly Connected

A directed graph is called **weakly connected** if the underlying undirected graph (obtained by ignoring all arrow directions) is connected. So weakly connected means connected if we pretend all edges are two-way. Weakly connected does NOT require paths in both directions.



Can reach any vertex from any vertex

e.g., $A \to B \to C \to D$ and $D \to A$



Can't go from Q to P (no reverse edge!)

Ignoring arrows: all connected → weakly connected

Theorem: Relationship Between Strong and Weak Connectivity

For any directed graph:

$$\text{Strongly Connected} \implies \text{Weakly Connected}$$

But the reverse is NOT true, weakly connected does not imply strongly connected (as shown in the example above).
Special case: If a directed graph has **no edges at all** between any pair of vertices (i.e., it's just isolated vertices), it is neither strongly nor weakly connected (unless it has just 1 vertex, in which case it is trivially both).

Example 260: Determining Strong vs Weak Connectivity

Consider a directed graph with vertices $\{P, Q, R, S\}$ and edges: $P \to Q$, $Q \to R$, $R \to S$, $Q \to S$.
Check strong connectivity: Can we go from S back to P? There is no outgoing edge from S at all (out-degree of S is 0). So S cannot reach any other vertex. **NOT strongly connected.**
Check weak connectivity: Ignoring directions, the underlying undirected graph has edges $P-Q$, $Q-R$, $R-S$, $Q-S$. This connects all 4 vertices into one component. **Weakly connected: YES.**

7.1.11 Walk, Trail, Circuit, Path, and Cycle

Concept: Walk

A **walk** is a finite alternating sequence of vertices and edges, where consecutive vertices are joined by the edge between them. Both vertices and edges are allowed to repeat freely in a walk. The **length** of a walk is the number of edges covered in it.
A walk is called **open** if its length is greater than 0 and the starting and ending vertices are different. A walk is called **closed** if its length is greater than 0 and the starting and ending vertices are the same. A walk of length 0 (a single vertex, no edges) is called a **trivial walk**.

Example 261: Analogy: Walk as Wandering

Think of a person wandering through a city with no particular destination in mind. They may cross the same street multiple times, or pass through the same square again and again. This unrestricted wandering, where nothing is off-limits to repeat, is exactly a walk.

Concept: Trail — Edges Cannot Repeat

A **trail** is an open walk in which vertices **may repeat**, but edges are **not allowed to repeat**. Think of a trail as a walk with one extra restriction: once you have used a particular road, you cannot use that exact road again, but you are still allowed to revisit the same junction.

Concept: Circuit — A Closed Trail

A **circuit** is a closed walk in which vertices may repeat, but edges are not allowed to repeat. In other words, a circuit is simply a trail that returns to its starting vertex. A closed trail is called a circuit.

Concept: Path — Neither Vertices Nor Edges Repeat

A **path** is an open walk in which neither vertices (except possibly the starting and ending vertex) nor edges are allowed to repeat. This is the strictest form of an open walk: every vertex is visited at most once, so naturally every edge is also used at most once (since repeating an edge would force repeating a vertex).

Concept: Cycle — A Closed Path

A **cycle** is a closed walk in which neither vertices nor edges are allowed to repeat, except that the starting and ending vertex is the same. In other words, a cycle is a closed path. A closed path is called a cycle.

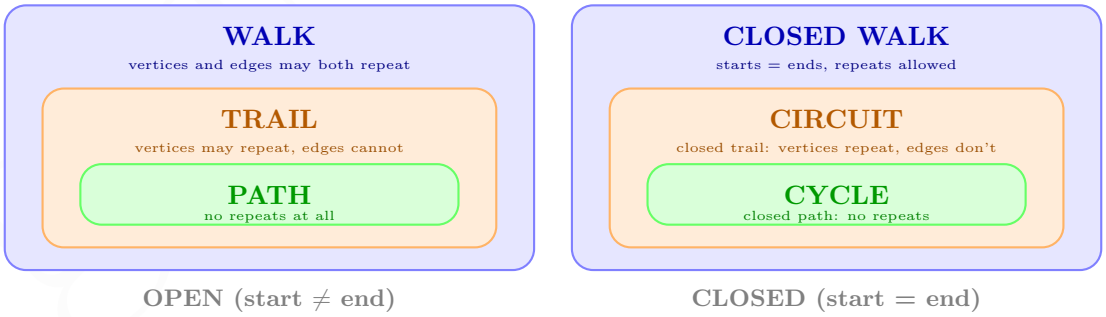
Example 262: Key Consequence: Path ⇒ Trail, Cycle ⇒ Circuit

Since a path never repeats vertices, it automatically never repeats edges either. So:

- **Every path is a trail**, but every trail need not be a path (a trail is allowed to repeat vertices, a path is not).
- **Every cycle is a circuit**, but every circuit need not be a cycle (a circuit is allowed to repeat vertices, a cycle is not).

For directed graphs, we simply place the word "directed" in front of each of these terms: directed walk, directed trail, directed circuit, directed path, directed cycle. The same repetition rules apply, just following the direction of the arrows.

Walk → Trail → Path (and Closed Versions)



Comparison Table: Walk, Trail, Circuit, Path, Cycle

Term	Open/Closed	Vertices Repeat?	Edges Repeat?
Walk	Either	Yes	Yes
Trail	Open	Yes	No
Circuit	Closed	Yes	No
Path	Open	No	No
Cycle	Closed	No	No

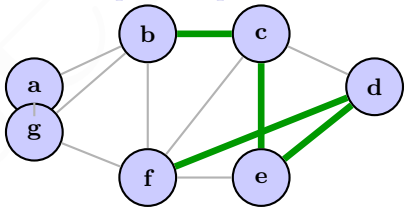
Example 263: Worked Example: Classifying Sequences on a Graph

Consider a graph with vertices $\{a, b, c, d, e, f, g\}$. Classify each of the following vertex sequences.

- a, b, g, f, c, b: Open (starts at a, ends at b). Vertex b repeats, but no edge repeats. This is a **trail** (not a path, since vertex b is repeated).
- b, g, f, c, b, g, a: Open. Both vertices (b, g) and possibly edges repeat here (edge b–g used twice). This is just a general **walk** (not even a trail, since an edge repeats).
- c, e, f, c: Closed (starts and ends at c). No vertex or edge repeats other than the start/end vertex c. This is a **cycle**.
- c, e, f, c, e: Open (starts at c, ends at e). If no edge repeats but vertices c and e appear more than once internally, this is a **trail**.
- a, b, f, a: If there is no edge directly between f and a in the graph, this sequence is **not a valid walk** at all, since consecutive vertices must be joined by an edge.
- f, d, e, c, b: Open, no vertex repeats, no edge repeats. This is a **path**.

Example 264: Worked Example: Directed Walks

Consider a directed graph with vertices $\{u, v, w, x, y\}$ and directed edges $x \to v$, $v \to y$, $y \to w$, $w \to v$, $x \to u$, $u \to x$.
Sequence x, v, y, w, v: Check each consecutive pair has a directed edge: $x \to v$ ✓, $v \to y$ ✓, $y \to w$ ✓, $w \to v$ ✓. This is a valid directed walk of length 4. Vertex v repeats, so it is not a directed path; since start (x) $\neq$ end (v), it is not a directed cycle either. If no edge repeats, it is a **directed trail**.
Sequence x, u, v, y, x: Check $u \to v$: no such directed edge exists (only $u \to x$ exists from u). This is **not a valid directed walk** at all.



Highlighted: path f, d, e, c, b — no repeats

7.1.12 Havel–Hakimi Theorem

Concept: The Graphical Sequence Problem

Given a list of numbers, a natural question is: can we build a simple undirected graph where these numbers are exactly the degrees of the vertices? Such a sequence is called a **graphical sequence** (or **graphic sequence**). Not every sequence of numbers is graphical; some sequences simply cannot correspond to any real graph, however hard we try to draw edges. The **Havel–Hakimi algorithm** gives us a step-by-step method to test this, without

needing to try every possible graph.

Concept: Havel–Hakimi Theorem

A degree sequence $d_1 \geq d_2 \geq \cdots \geq d_n \geq 0$ (sorted in non-increasing order) is **graphical** if and only if the sequence

$$(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$$

obtained by removing the largest value d_1 and subtracting 1 from the next d_1 largest entries, is **also graphical**.
In plain words: take the vertex with the highest degree, say d_1. Connect it to the d_1 next-highest-degree vertices (this is always possible, and always the safest choice). Once that vertex is fully connected, remove it from the list, and reduce the degree of each vertex it was connected to by 1 (since each of those vertices has now used up one of its edges). Repeat this process on the smaller sequence.

Example 265: Analogy: Havel–Hakimi as a Greedy Matchmaking Process

Think of n people, each wanting a specific number of handshakes $(d_1, d_2, \ldots, d_n)$. The greedy idea is: take the person who wants the **most** handshakes, and simply pair them up with the next most eager people, since that is always a safe, non-wasteful choice. Once that person's handshakes are all arranged, we remove them from the room and reduce the remaining handshake counts of the people they shook hands with by 1 each (since those handshakes have already happened). We keep repeating this with the reduced list.
If at any point during this process we get a **negative number** in the list, it means the sequence is **not graphical**, impossible to satisfy. If we can reduce the whole list down to all zeros, the sequence **is graphical**.

Concept: Stopping Conditions of the Algorithm

At each step of the algorithm, sort the sequence in non-increasing order first. Then check:

- If all remaining values are 0, the sequence is **graphical**. Stop.
- If the largest value d_1 is greater than the number of remaining entries (excluding itself), the sequence is **NOT graphical**. Stop (there simply aren't enough other vertices to connect to).
- If subtracting 1 produces any **negative** value, the sequence is **NOT graphical**. Stop.
- Otherwise, repeat the reduction step on the new sequence.

Algorithm: Havel-Hakimi(sequence)

```
1:  while true do
2:      Sort sequence in non-increasing order
3:      Remove all trailing zeros
4:      if sequence is empty then
5:          return true                                  ▷ graphical
6:      end if
7:      d_1 ← sequence[0]                                ▷ largest value
8:      Remove d_1 from sequence
9:      if d_1 > length(sequence) then
10:         return false                                 ▷ not enough vertices left
11:     end if
12:     for i = 0 to d_1 − 1 do
13:         sequence[i] ← sequence[i] − 1
14:         if sequence[i] < 0 then
15:             return false                             ▷ not graphical
16:         end if
17:     end for
18: end while
```

Example 266: Example I: Testing $7, 6, 5, 4, 4, 3, 2, 0$

Step 1: Sequence: $7, 6, 5, 4, 4, 3, 2, 0$. Largest value is 7. Remove it, and subtract 1 from the next 7 entries.

$$6-1, 5-1, 4-1, 4-1, 3-1, 2-1, 0-1$$

The last entry becomes $0 - 1 = -1$, which is **negative**.
Conclusion: Since a negative value appears, the sequence $7, 6, 5, 4, 4, 3, 2, 0$ is **NOT graphical**.
Intuition: A vertex wanting degree 0 cannot be connected to at all, but the vertex with degree 7 needs to connect to all 7 remaining vertices, forcing a connection with the degree-0 vertex. This is a direct contradiction.

Example 267: Example II: Testing $6, 6, 6, 6, 3, 3, 2, 2$

Step 1: Sequence: $6, 6, 6, 6, 3, 3, 2, 2$. Largest value is 6. Remove it, subtract 1 from the next 6 entries.

$$6-1, 6-1, 6-1, 3-1, 3-1, 2-1,\ 2$$

New sequence (before sorting): $5, 5, 5, 2, 2, 1, 2$. Sorted: $5, 5, 5, 2, 2, 2, 1$.
Step 2: Largest value is 5. Remove it, subtract 1 from the next 5 entries.

$$5-1, 5-1, 2-1, 2-1, 2-1,\ 1$$

New sequence (before sorting): $4, 4, 1, 1, 1, 1$. Sorted: $4, 4, 1, 1, 1, 1$.
Step 3: Largest value is 4. Remove it, subtract 1 from the next 4 entries.

$$4-1, 1-1, 1-1, 1-1,\ 1$$

New sequence (before sorting): $3, 0, 0, 0, 1$. Sorted: $3, 1, 0, 0, 0$.
Step 4: Largest value is 3. Remove it, subtract 1 from the next 3 entries.

$$1-1, 0-1, 0-1$$

The entry $0 - 1 = -1$ is **negative**.
Conclusion: The sequence $6, 6, 6, 6, 3, 3, 2, 2$ is **NOT graphical**.

Example 268: Example III: Testing $7, 6, 6, 4, 4, 3, 2, 2$

Step 1: Sequence: $7, 6, 6, 4, 4, 3, 2, 2$. Largest value is 7, and there are exactly 7 remaining entries, so this is valid (not too large). Subtract 1 from all 7 remaining entries.

$$6-1, 6-1, 4-1, 4-1, 3-1, 2-1, 2-1$$

New sequence: $5, 5, 3, 3, 2, 1, 1$. Already sorted.
Step 2: Largest value is 5. Remove it, subtract 1 from the next 5 entries.

$$5-1, 3-1, 3-1, 2-1, 1-1,\ 1$$

New sequence (before sorting): $4, 2, 2, 1, 0, 1$. Sorted: $4, 2, 2, 1, 1, 0$.
Step 3: Largest value is 4. Remove it, subtract 1 from the next 4 entries.

$$2-1, 2-1, 1-1, 1-1,\ 0$$

New sequence (before sorting): $1, 1, 0, 0, 0$. Sorted: $1, 1, 0, 0, 0$.
Step 4: Drop trailing zeros: $1, 1$. Largest value is 1. Remove it, subtract 1 from the next 1 entry.

$$1-1=0$$

New sequence: 0.
Step 5: Drop trailing zero. Sequence is now empty.
Conclusion: We reached an empty sequence (all zeros) without ever hitting a negative number. So $7, 6, 6, 4, 4, 3, 2, 2$ **IS graphical**.

Example 269: Example IV: Testing $8, 7, 7, 6, 4, 2, 1, 1$

Step 1: Sequence: $8, 7, 7, 6, 4, 2, 1, 1$. The largest value is 8, but there are only 7 remaining entries in the sequence.

$$d_1 = 8 > (\text{number of remaining entries}) = 7$$

Conclusion: Since the largest degree (8) exceeds the number of other vertices available to connect to (only 7 remain), it is impossible for this vertex to achieve degree 8 in a simple graph (no self-loops or multi-edges allowed). So $8, 7, 7, 6, 4, 2, 1, 1$ is **NOT graphical**.
Intuition: With only 8 vertices total, the maximum possible degree for any single vertex is $8 - 1 = 7$ (connecting to every other vertex). A degree of 8 is simply impossible here.

Havel–Hakimi: Summary of Four Examples

Example	Sequence	Result
I	`7,6,5,4,4,3,2,0`	NOT graphical
II	`6,6,6,6,3,3,2,2`	NOT graphical
III	`7,6,6,4,4,3,2,2`	Graphical
IV	`8,7,7,6,4,2,1,1`	NOT graphical

7.2 Graph Representations

Concept: Why Representation Matters

The same graph can be stored in a computer in different ways. The choice of representation affects:

- How much memory we use.
- How fast we can check if an edge exists.
- How fast we can traverse all neighbours of a vertex.
- How fast graph algorithms run in total.

The two most common representations are the **adjacency matrix** and the **adjacency list**.

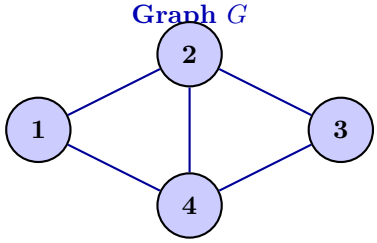
7.2.1 Adjacency Matrix

Concept: Adjacency Matrix

An **adjacency matrix** is a 2D array (matrix) of size $|V| \times |V|$. For a graph with $|V| = n$ vertices numbered 1 to n:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{otherwise} \end{cases}$$

For a **weighted graph**, we store the weight instead of 1. For an **undirected graph**, the matrix is always symmetric, meaning $A[i][j] = A[j][i]$.
Space: $O(V^2)$, regardless of the number of edges.
Edge check: $O(1)$, since $A[u][v]$ gives the answer directly.
Finding all neighbours of v: $O(V)$, since we must scan the entire row v.



Adjacency Matrix

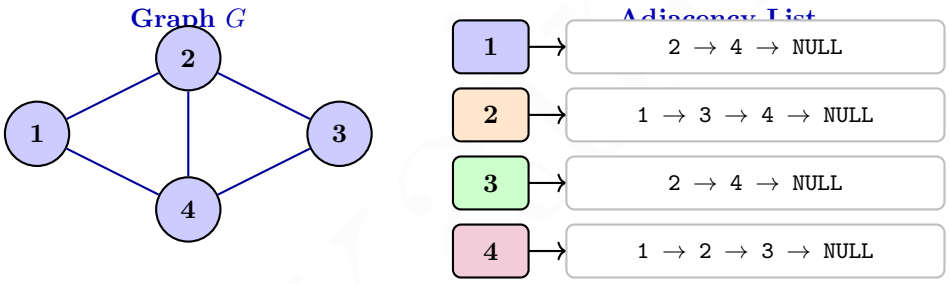
	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	1
4	1	1	1	0

Space: $O(V^2)$, Edge check: $O(1)$
Symmetric for undirected graphs

7.2.2 Adjacency List

Concept: Adjacency List

An **adjacency list** stores, for each vertex v, a list of all vertices adjacent to v (its neighbours). We can implement this as an array of linked lists, or as an array of Python lists.
Space: $O(V+E)$, since we store $|V|$ list headers and $2|E|$ total entries (each edge appears in two lists for undirected graph, or $|E|$ entries total for directed).
Edge check: $O(\deg(v))$, we must scan the list of v to see if u is in it.
Finding all neighbours of v: $O(\deg(v))$, simply iterate the list.
Total traversal: $O(V+E)$, visiting all vertices and all their edges.



Space: $O(V+E)$, Edge check: $O(\deg(v))$

7.2.3 Dense vs. Sparse Graphs

Concept: Dense and Sparse Graphs

A **dense graph** has many edges, close to the maximum possible. A complete graph K_n is the densest possible, with $n(n-1)/2$ edges. For dense graphs, $|E| \approx |V|^2$, so adjacency matrix wastes little space and makes edge checks very fast.
A **sparse graph** has few edges, much less than the maximum. For sparse graphs, $|E| \ll |V|^2$, often $|E| \approx |V|$ or $|E| \approx |V| \log |V|$. Adjacency list is much better here because the matrix would be mostly empty zeros, wasting memory.

Dense vs Sparse: Which Representation to Use?

Property	Adjacency Matrix	Adjacency List
Space	$O(V^2)$	$O(V+E)$
Edge check (u,v)	$O(1)$	$O(\deg(u))$
All neighbors of v	$O(V)$	$O(\deg(v))$
Full BFS/DFS traversal	$O(V^2)$	$O(V+E)$
Add edge	$O(1)$	$O(1)$
Delete edge	$O(1)$	$O(\deg(v))$
Best for	Dense graphs	Sparse graphs

Example 270: When to Use Which

- Facebook network: billions of users (vertices) but each person has only a few thousand friends (edges). This is very sparse. $|E| \ll |V|^2$. Use adjacency list.
- A 5-city road network where every city is connected to every other: K_5, 10 edges. Dense for 5 nodes. Matrix is fine here.
- Flight network: thousands of airports, but not all airports connect to each other. Sparse. Use adjacency list.
- A graph algorithm on a 1000-vertex dense graph: matrix allows $O(1)$ edge check, which is critical here.

Concept: Traversal Complexity — Matrix vs List

When we run graph traversal algorithms like BFS or DFS:

- **Using adjacency matrix:** For each vertex v, we scan its entire row to find neighbours. This takes $O(V)$ per vertex, and with V vertices total, the whole traversal takes $O(V^2)$.
- **Using adjacency list:** For each vertex v, we only look at the neighbours in its list. The total work across all vertices is proportional to the total number of entries in all lists, which equals $2|E|$ (each edge appears twice for undirected). So total traversal takes $O(V+E)$.

For sparse graphs where $|E| \ll |V|^2$, the adjacency list traversal $O(V+E)$ is much faster than matrix traversal $O(V^2)$.

Example 271: Traversal Cost Example

Consider a sparse graph with $|V| = 1000$ and $|E| = 2000$ (each vertex connects to 2 others on average).

- Adjacency matrix BFS/DFS: $O(V^2) = O(1000^2) = O(1{,}000{,}000)$ operations.
- Adjacency list BFS/DFS: $O(V+E) = O(1000+2000) = O(3000)$ operations.

Adjacency list is about 333 times faster here. This difference becomes even larger for bigger sparse graphs.

7.3 Practice Problems

Problem 229 ***[MCQ]*** *An undirected graph has 8 vertices, each with degree exactly 3. The number of edges in this graph is:*

(A) 8

(B) 12

(C) 24

(D) 16

Problem 230 ***[NAT]*** *An undirected graph has 6 vertices with degrees $2, 3, 3, 4, 2, 2$ respectively. The number of edges in this graph is _____.*

Problem 231 ***[MSQ]*** *Which of the following statements about the Handshaking Lemma are TRUE for any simple undirected graph?*

(A) The sum of all vertex degrees is always even.

(B) The number of vertices with odd degree is always even.

(C) A graph can have exactly one vertex with odd degree.

(D) If all vertices have even degree, the number of edges must also be even.

Problem 232 ***[MCQ]*** *In a directed graph with 5 vertices, the sum of all in-degrees equals 12. What is the sum of all out-degrees?*

(A) 6

(B) 12

(C) 24

(D) Cannot be determined

Problem 233 ***[NAT]*** *A directed graph has vertices with out-degrees $2, 1, 3, 0, 2$. The total number of edges in this graph is _____.*

Problem 234 ***[MCQ]*** *A complete graph K_n has 45 edges. The value of n is:*

(A) 9

(B) 10

(C) 11

(D) 8

Problem 235 ***[MCQ]*** *Every vertex in the complete graph K_n has degree:*

(A) n

(B) $n + 1$

(C) $n - 1$

(D) $n/2$

Problem 236 ***[MSQ]*** *Which of the following statements about paths, walks, and cycles are TRUE?*

(A) A simple path visits each vertex at most once.

(B) A walk can repeat vertices and edges.

(C) Every simple cycle is also a simple path.

(D) In an undirected simple graph, the shortest possible simple cycle has length 3.

Problem 237 ***[MSQ]*** *Consider an undirected graph with vertices $\{A, B, C, D, E\}$ and edges $\{(A, B), (B, C), (C, D), (D, E), (E, A), (B, D)\}$. Which of the following is a simple cycle?*

(A) $A \to B \to C \to D \to E \to A$

(B) $A \to B \to D \to A$

(C) $A \to B \to C \to D \to B$

(D) $B \to C \to D \to B$

Problem 238 ***[NAT]*** *An undirected graph has 20 vertices and forms exactly 4 connected components, each of which is a tree. The total number of edges in the graph is _____.*

Problem 239 ***[MSQ]*** *Which of the following statements about strongly connected and weakly connected directed graphs are TRUE?*

(A) Every strongly connected directed graph is also weakly connected.

(B) A weakly connected directed graph is always also strongly connected.

(C) In a strongly connected graph, there is a directed path from every vertex to every other vertex.

(D) A directed graph with a single source and single sink can never be strongly connected (if source $\neq$ sink).

Problem 240 ***[MCQ]*** *A directed graph has 4 vertices $\{P, Q, R, S\}$ with edges $P \to Q, Q \to R, R \to S, S \to P$. This graph is:*

(A) Weakly connected only

(B) Strongly connected

(C) Neither weakly nor strongly connected

(D) A DAG

Problem 241 ***[NAT]*** *A directed graph has edges $A \to B$, $B \to C$, $D \to C$, $D \to E$. The number of connected components is _____.*

Problem 242 ***[MCQ]*** *A forest (a disjoint union of trees) has 30 vertices and 25 edges. The number of trees (connected components) in this forest is:*

(A) 5

(B) 25

(C) 30

(D) 55

Problem 243 ***[MCQ]*** *A connected undirected graph G with n vertices has exactly n edges. Which of the following is TRUE?*

(A) G must be a tree.

(B) G contains exactly one cycle.

(C) G is disconnected.

(D) G cannot exist for any n.

Problem 244 ***[MCQ]*** *For a graph with $|V| = n$ vertices, the adjacency matrix representation always requires how much space, regardless of the number of edges?*

(A) $O(n)$

(B) $O(n \log n)$

(C) $O(n^2)$

(D) $O(n^3)$

Problem 245 ***[NAT]*** *An undirected graph has $|V| = 500$ vertices and $|E| = 800$ edges. Using an adjacency list representation, the total number of entries across all adjacency lists combined is _____.*

Problem 246 ***[MCQ]*** *Checking whether an edge (u, v) exists takes $O(1)$ time using which representation?*

(A) Adjacency list only

(B) Adjacency matrix only

(C) Both adjacency list and adjacency matrix

(D) Neither representation supports $O(1)$ edge check

Problem 247 ***[MSQ]*** *Which of the following are TRUE when comparing adjacency matrix and adjacency list representations for a graph with $|V| = n$ and $|E| = m$?*

(A) Adjacency matrix uses $O(n^2)$ space regardless of m.

(B) Adjacency list uses $O(n + m)$ space.

(C) For a sparse graph where $m = O(n)$, the adjacency list is more space-efficient.

(D) Finding all neighbours of a vertex is always faster with adjacency matrix than adjacency list.

Problem 248 ***[MCQ]*** *A full traversal (visiting every vertex and edge) of a graph with $|V| = n$ vertices and $|E| = m$ edges takes how much time using an adjacency list, versus an adjacency matrix?*

(A) $O(n + m)$ for list, $O(n^2)$ for matrix

(B) $O(n^2)$ for list, $O(n + m)$ for matrix

(C) $O(n + m)$ for both

(D) $O(n^2)$ for both

Problem 249 ***[NAT]*** *A directed graph has $|V| = 200$ vertices and $|E| = 350$ edges. Using an adjacency list representation for this directed graph, the total number of entries across all adjacency lists combined is _____.*

Problem 250 ***[MCQ]*** *A graph has $|V| = 1000$ vertices. It is known that $|E| = 999$ and the graph is connected. Which representation would generally be preferred for this graph, and why?*

(A) Adjacency matrix, because $|V|$ is large

(B) Adjacency list, because the graph is very sparse ($|E| \approx |V|$)

(C) Adjacency matrix, because edge check must be $O(1)$

(D) Both are equally efficient in this case

Problem 251 ***[MCQ]*** *Consider the following adjacency matrix for an undirected graph with 4 vertices (1-indexed):*

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

The degree of vertex 2 is:

(A) 1

(B) 2

(C) 3

(D) 4

Problem 252 ***[NAT]*** *An adjacency matrix for a directed graph with 5 vertices has row sums (sum of each row) equal to $2, 1, 0, 3, 1$. The total number of edges in this graph is _____.*

Problem 253 ***[MSQ]*** *Consider converting an adjacency list to an adjacency matrix for a graph with $|V| = n$ and $|E| = m$. Which of the following are TRUE about this conversion?*

(A) Initializing the matrix to all zeros takes $O(n^2)$ time.

(B) Setting the appropriate matrix entries for all edges takes $O(m)$ time (or $O(2m)$ for undirected).

(C) The total time complexity of the conversion is $O(n^2 + m)$.

(D) The conversion can be done in $O(m)$ time regardless of n.

Problem 254 ***[NAT]*** *A simple undirected graph on 6 vertices has the maximum possible number of edges without being a complete graph. The number of edges in this graph is _____.*

Problem 255 ***[MSQ]*** *Which of the following are TRUE about degree sequences in directed and undirected graphs?*

(A) In an undirected simple graph with n vertices, the maximum possible degree of any vertex is $n-1$.

(B) In a directed simple graph with n vertices, the maximum possible out-degree of any vertex is $n-1$.

(C) In any directed graph, $\sum \deg^-(v) = \sum \deg^+(v)$.

(D) The sum of in-degrees always equals the sum of out-degrees plus the sum of undirected degrees.

Problem 256 ***[MCQ]*** *Consider the following function that checks if a graph (given as adjacency list) could represent a tree:*

```
function could_be_tree(adj_list, n):
    total_degree_sum = 0
    for i = 0 to n - 1:
        total_degree_sum = total_degree_sum + length(adj_list[i])

    edge_count = total_degree_sum / 2

    if edge_count == n - 1 then:
        return true
    else:
        return false
```

*This function alone is **insufficient** to confirm the graph is a tree because:*

(A) It doesn't check whether the graph is connected.

(B) It doesn't check the degree sequence.

(C) It always returns False for valid trees.

(D) Edge count can never equal $n-1$ for any graph.

Problem 257 ***[NAT]*** *A directed graph is used to represent course prerequisites (an edge $u \to v$ means course u must be completed before course v). The graph has 15 courses (vertices) and is guaranteed to be a DAG. If the graph has exactly 1 source vertex (no prerequisites) and 1 sink vertex (no course depends on completing it further), and every other vertex has both in-degree ≥ 1 and out-degree ≥ 1, what is the minimum possible number of edges required to connect all 15 vertices into a single weakly connected component? _____*

Problem 258 ***[MCQ]*** *Consider a graph algorithm that needs to repeatedly check whether specific edges exist (many edge-existence queries) but rarely needs to enumerate all neighbours of a vertex. Which representation is most suitable, and what is the query time?*

(A) Adjacency list, $O(\deg(v))$ per query

(B) Adjacency matrix, $O(1)$ per query

(C) Adjacency list, $O(1)$ per query

(D) Adjacency matrix, $O(V)$ per query

Problem 259 ***[MSQ]*** *Consider an undirected graph with vertices $\{a, b, c, d, e, f, g\}$ and edges $\{(a,b), (a,g), (b,c), (b,f), (b,g), (c,d), (c,e), (c,f), (d,e), (e,f), (f,g)\}$. Which of the following sequences correctly represent a **trail** (but NOT a path)?*

(A) a, b, g, f, c, b

(B) c, e, f, c, e

(C) f, d, e, c, b

(D) b, g, f, c, b, g, a

Problem 260 ***[MCQ]*** *A closed walk in an undirected graph has length 6. No edge is repeated, but vertex v_3 is visited twice (once in the middle, once implicitly counted via the closure). This closed walk is best classified as a:*

(A) Cycle

(B) Circuit (but not a cycle)

(C) Path

(D) Trail (but not a circuit)

Problem 261 ***[MSQ]*** *Which of the following statements correctly describe the relationships between walk, trail, circuit, path, and cycle?*

(A) Every path is a trail, but not every trail is a path.

(B) Every cycle is a circuit, but not every circuit is a cycle.

(C) A trail can repeat vertices but never repeats edges.

(D) A circuit must be an open walk.

Problem 262 ***[MSQ]*** *Which of the following degree sequences (in decreasing order) is NOT graphic, i.e., cannot correspond to any simple undirected graph?*

(A) $(4, 4, 3, 2, 1)$

(B) $(3, 3, 3, 3)$

(C) $(5, 3, 2, 1, 1)$

(D) $(4, 4, 4, 4, 4)$

Problem 263 ***[NAT]*** *A simple undirected connected graph has 14 edges. Every vertex has degree at least 4. The maximum possible number of vertices n in this graph is _____.*

Problem 264 ***[MCQ]*** *Consider a simple connected undirected graph with exactly 7 vertices, where no two vertices have the same degree, except possibly needing to satisfy a graph-theoretic constraint. Which of the following statements is TRUE for ANY simple connected undirected graph with more than 2 vertices?*

(A) No two vertices can have the same degree.

(B) At least two vertices must have the same degree.

(C) Exactly one pair of vertices has the same degree.

(D) All vertices must have distinct degrees except the leaves.

Problem 265 ***[NAT]*** *A simple undirected graph on 12 vertices is disconnected, split into exactly 2 non-empty groups of vertices with no edges between the groups. To maximize the total number of edges in this disconnected graph, one group should have as few vertices as possible (at least 1) and the other as many as possible. The maximum possible number of edges in such a disconnected graph on 12 vertices is _____.*

Problem 266 ***[MCQ]*** *Let G be a complete undirected graph K_7 on 7 labeled vertices. The number of distinct cycles of length 4 in G is:*

(A) 105

(B) 210

(C) 420

(D) 840

Problem 267 ***[NAT]*** *The number of distinct simple undirected graphs (labeled vertices, no self-loops) that can be constructed on a set of exactly 4 vertices is _____.*

Problem 268 ***[MCQ]*** *How many distinct undirected simple graphs (not necessarily connected) can be constructed on a labeled vertex set of n vertices?*

(A) $n!$

(B) 2^n

(C) $\dfrac{n(n-1)}{2}$

(D) $2^{n(n-1)/2}$

Problem 269 ***[NAT]*** *Consider an undirected graph G where each vertex is a cell (i, j) with $1 \le i \le 8$ and $1 \le j \le 8$ (an 8×8 grid), and there is an edge between (a, b) and (c, d) if $\max(|a-c|, |b-d|) = 1$ (they are adjacent including diagonally, like a king's move in chess, but not the same cell). The total number of edges in this graph is _____.*

Problem 270 ***[MCQ]*** *Let G be a graph with 60 vertices labeled 1 to 60. Two vertices i and j are adjacent if and only if $|i-j| = 5$ or $|i-j| = 15$. The number of connected components in G is:*

(A) *3*

(B) *5*

(C) *12*

(D) *15*

Problem 271 ***[MCQ]*** *Consider the undirected graph G whose vertices are all bit-strings of length n, with an edge between u and v if and only if u and v differ in exactly one bit position. The degree of every vertex in this graph is:*

(A) n

(B) $n-1$

(C) 2^n

(D) 2^{n-1}

Problem 272 ***[MSQ]*** *Which of the following statements are TRUE for a simple undirected graph G with n vertices and m edges?*

(A) *If $m > \binom{n}{2} - 1$, then G must be the complete graph K_n.*

(B) *If G is disconnected, then $m \le \binom{n-1}{2}$.*

(C) *If $m \ge n$, G must contain at least one cycle.*

(D) *If G has exactly one cycle and is connected, then $m = n$.*

gateoverflow.in

7.4 GATE PYQs

GATEPYQ 173 (GATE CSE 2024) *The number of edges present in the forest generated by the DFS traversal of an undirected graph G with 100 vertices is 40. The number of connected components in G is*

GATEPYQ 174 (GATE CSE 2022) *Consider a simple undirected unweighted graph with at least three vertices. If A is the adjacency matrix of the graph, then the number of 3-cycles in the graph is given by the trace of*

A. A^3

B. A^3 *divided by 2*

C. A^3 *divided by 3*

D. A^3 *divided by 6*

GATEPYQ 175 (GATE CSE 2022) *Consider a simple undirected graph of 10 vertices. If the graph is disconnected, then the maximum number of edges it can have is _____.*

GATEPYQ 176 (GATE CSE 2018) *Let G be a graph with* $100!$ *vertices, with each vertex labelled by a distinct permutation of the numbers $1, 2, \ldots, 100$. There is an edge between vertices u and v if and only if the label of u can be obtained by swapping two adjacent numbers in the label of v. Let y denote the degree of a vertex in G, and z denote the number of connected components in G. Then, $y + 10z =$ _____.*

GATEPYQ 177 (GATE CSE 2017) *G is an undirected graph with n vertices and 25 edges such that each vertex of G has degree at least 3. Then the maximum possible value of n is _____.*

GATEPYQ 178 (GATE CSE 2017) *Let T be a tree with 10 vertices. The sum of the degrees of all the vertices in T is _____.*

GATEPYQ 179 (GATE CSE 2014) *If G is a forest with n vertices and k connected components, how many edges does G have?*

A. $\lfloor n/k \rfloor$

B. $\lceil n/k \rceil$

C. $n - k$

D. $n - k + 1$

GATEPYQ 180 (GATE CSE 2014) *An ordered n-tuple $(d_1, d_2, \ldots, d_n)$ with $d_1 \geq d_2 \geq \cdots \geq d_n$ is called graphic if there exists a simple undirected graph with n vertices having degrees $d_1, d_2, \ldots, d_n$ respectively. Which of the following 6-tuples is NOT graphic?*

A. $(1, 1, 1, 1, 1, 1)$

B. $(2, 2, 2, 2, 2, 2)$

C. $(3, 3, 3, 1, 0, 0)$

D. $(3, 2, 1, 1, 1, 0)$

GATEPYQ 181 (GATE CSE 2014) *Consider an undirected graph G where self-loops are not allowed. The vertex set of G is $\{(i, j) : 1 \leq i \leq 12, 1 \leq j \leq 12\}$. There is an edge between (a, b) and (c, d) if $|a - c| \leq 1$ and $|b - d| \leq 1$. The number of edges in the graph is _____.*

GATEPYQ 182 (GATE CSE 2014) *Let $G = (V, E)$ be a directed graph where V is the set of vertices and E the set of edges. Then which one of the following graphs has the same strongly connected components as G?*

A. $G_1 = (V, E_1)$ *where* $E_1 \equiv \{(u, v) \mid (u, v) \notin E\}$

B. $G_2 = (V, E_2)$ *where* $E_2 \equiv \{(u, v) \mid (v, u) \in E\}$

C. $G_3 = (V, E_3)$ *where* $E_3 \equiv \{(u, v) \mid$ *there is a path of length* ≤ 2 *from u to v in* $E\}$

D. $G_4 = (V_4, E)$ *where V_4 is the set of vertices in G which are not isolated*

GATEPYQ 183 (GATE CSE 2013) *Which of the following statements is/are TRUE for undirected graphs?*

P: Number of odd degree vertices is even.

Q: Sum of degrees of all vertices is even.

A. P only

B. Q only

C. Both P and Q

D. Neither P nor Q

GATEPYQ 184 (GATE CSE 2013) *Consider an undirected random graph of eight vertices. The probability that there is an edge between a pair of vertices is* $1/2$*. What is the expected number of unordered cycles of length three?*

A. $1/8$

B. 1

C. 7

D. 8

GATEPYQ 185 (GATE CSE 2012) *Let G be a complete undirected graph on 6 vertices. If vertices of G are labeled, then the number of distinct cycles of length 4 in G is equal to*

A. 15

B. 45

C. 90

D. 360

GATEPYQ 186 (GATE CSE 2010) *The degree sequence of a simple graph is the sequence of the degrees of the nodes in the graph in decreasing order. Which of the following sequences can not be the degree sequence of any graph?*

I. 7, 6, 5, 4, 4, 3, 2, 1

II. 6, 6, 6, 6, 3, 3, 2, 2

III. 7, 6, 6, 4, 4, 3, 2, 2

IV. 8, 7, 7, 6, 4, 2, 1, 1

A. I and II

B. III and IV

C. IV only

D. II and IV

GATEPYQ 187 (GATE CSE 2010) *Let* $G = (V, E)$ *be a graph. Define* $\xi(G) = \sum_d i_d * d$*, where* i_d *is the number of vertices of degree d in G. If S and T are two different trees with* $\xi(S) = \xi(T)$*, then*

A. $|S| = 2|T|$

B. $|S| = |T| - 1$

C. $|S| = |T|$

D. $|S| = |T| + 1$

GATEPYQ 188 (GATE CSE 2009) *Which one of the following is TRUE for any simple connected undirected graph with more than 2 vertices?*

A. No two vertices have the same degree.

B. At least two vertices have the same degree.

C. At least three vertices have the same degree.

D. All vertices have the same degree.

GATEPYQ 189 (GATE IT 2005) *In a depth-first traversal of a graph G with n vertices, k edges are marked as tree edges. The number of connected components in G is*

A. k

B. $k+1$

C. $n-k-1$

D. $n-k$

GATEPYQ 190 (GATE IT 2004) *What is the number of vertices in an undirected connected graph with 27 edges, 6 vertices of degree 2, 3 vertices of degree 4 and remaining of degree 3?*

A. 10

B. 11

C. 18

D. 19

GATEPYQ 191 (GATE IT 2004) *What is the maximum number of edges in an acyclic undirected graph with n vertices?*

A. $n-1$

B. n

C. $n+1$

D. $2n-1$

GATEPYQ 192 (GATE CSE 2003) *A graph* $G=(V,E)$ *satisfies* $|E| \leq 3|V|-6$. *The min-degree of G is defined as* $\min_{v \in V}\{degree(v)\}$. *Therefore, min-degree of G cannot be*

A. 3

B. 4

C. 5

D. 6

GATEPYQ 193 (GATE CSE 2003) *Let G be an arbitrary graph with n nodes and k components. If a vertex is removed from G, the number of components in the resultant graph must necessarily lie between*

A. k *and* n

B. $k-1$ *and* $k+1$

C. $k-1$ *and* $n-1$

D. $k+1$ *and* $n-k$

GATEPYQ 194 (GATE CSE 2002) *Maximum number of edges in a n-node undirected graph without self loops is*

A. n^2

B. $\frac{n(n-1)}{2}$

C. $n-1$

D. $\frac{(n+1)n}{2}$

GATEPYQ 195 (GATE CSE 1995) *The minimum number of edges in a connected cyclic graph on n vertices is*

A. $n-1$

B. n

C. $n+1$

D. None of the above

GATEPYQ 196 (GATE CSE 1994) *The number of distinct simple graphs with up to three nodes is*

A. *15*

B. *10*

C. *7*

D. *9*

GATEPYQ 197 (GATE CSE 1993) *Consider a simple connected graph G with n vertices and n edges ($n > 2$). Then, which of the following statements are true?*

A. *G has no cycles*

B. *The graph obtained by removing any edge from G is not connected*

C. *G has at least one cycle*

D. *The graph obtained by removing any two edges from G is not connected*

E. *None of the above*

GATEPYQ 198 (GATE CSE 2001) *How many undirected graphs (not necessarily connected) can be constructed out of a given set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices?*

A. $\frac{n(n-1)}{2}$

B. 2^n

C. $n!$

D. $2^{\frac{n(n-1)}{2}}$

GATEPYQ 199 (GATE IT 2008) *A binary tree with $n > 1$ nodes has n_1, n_2 and n_3 nodes of degree one, two and three respectively. The degree of a node is defined as the number of its neighbors.*
n_3 can be expressed as:

A. $n_1 + n_2 - 1$

B. $n_1 - 2$

C. $[(n_1 + n_2)/2]$

D. $n_2 - 1$

GATEPYQ 200 (GATE IT 2008) *A binary tree with $n > 1$ nodes has n_1, n_2 and n_3 nodes of degree one, two and three respectively. The degree of a node is defined as the number of its neighbors.*
Starting with the above tree, while there remains a node v of degree two in the tree, add an edge between the two neighbors of v and then remove v from the tree. How many edges will remain at the end of the process?

A. $2 * n_1 - 3$

B. $n_2 + 2 * n_1 - 2$

C. $n_3 - n_2$

D. $n_2 + n_1 - 2$

7.5 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
59	Introduction to Graph Theory	https://www.youtube.com/watch?v=prSaCGH0rY8	
60	Connected Graphs & Connected Components	https://www.youtube.com/watch?v=NUJs07KMj3w	
61	Walk, Trail, Circuit, Path & Cycle	https://www.youtube.com/watch?v=B83cMvfUC-k	
62	Graph Representation	https://www.youtube.com/watch?v=SqBu0T9Bc5Q	
63	Havel-Hakimi Theorem | Graph Theory	https://www.youtube.com/watch?v=0T1IYAd_t2c	
64	GATE PYQs on Basics of Graph Theory	https://www.youtube.com/watch?v=9yihYWBPZ3U	
65	Practice Problems on Basics of Graph Theory	https://www.youtube.com/watch?v=gsO3Ns7u4k4	

Chapter 8

Graph Traversals

Concept: What is Graph Traversal

Graph traversal means visiting every vertex of a graph in a systematic way, making sure no vertex is missed and no vertex is visited twice. Unlike arrays or linked lists where there is only one natural order to visit elements, a graph can be traversed in many different ways depending on the strategy used. The two most fundamental and important traversal strategies are **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

Example 272: Why Traversal Matters

Traversal is the foundation for solving many graph problems: finding shortest paths, detecting cycles, topological sorting, finding connected components, and many more. Mastering BFS and DFS step-by-step tracing is the single highest-yield graph topic in competitive exams.

8.1 Breadth-First Search (BFS)

8.1.1 Introduction to BFS

Concept: What is BFS

Breadth-First Search (BFS) explores a graph **level by level**, outward from a chosen source vertex. It visits **all neighbours** of the current vertex before moving any deeper into the graph. BFS uses a **FIFO queue** to control the order of traversal, which is exactly what guarantees the level-by-level behaviour.

Example 273: Where BFS is Used

- Finding the shortest path (fewest edges) in an **unweighted** graph.
- Finding all connected components of a graph.
- Level-order traversal of a tree (BFS applied to a tree is exactly level-order traversal).

8.1.2 BFS Algorithm

Concept: Colors and Labels Used in BFS

Each vertex is given a color to track its traversal state:

- **WHITE:** not yet discovered.
- **GRAY:** discovered, but still has unexplored neighbours (currently in the queue).
- **BLACK:** fully explored, all its neighbours have been looked at.

Along with color, we track $d[v]$ (distance from source s) and $\pi[v]$ (the parent of v in the BFS tree).

Algorithm: BFS(G, s)

```
1: for each u ∈ V[G] − {s} do
2:     color[u] ← WHITE
3:     d[u] ← ∞
4:     π[u] ← NIL
5: end for
6: color[s] ← GRAY, d[s] ← 0, π[s] ← NIL
7: Q ← empty queue
8: Enqueue(Q, s)
9: while Q is not empty do
10:    u ← Dequeue(Q)
11:    for each v ∈ Adj[u] do
12:        if color[v] = WHITE then
13:            color[v] ← GRAY
14:            d[v] ← d[u] + 1
15:            π[v] ← u
16:            Enqueue(Q, v)
17:        end if
18:    end for
19:    color[u] ← BLACK
20: end while
```

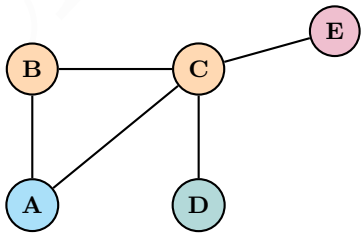
Concept: Time Complexity

With an adjacency list, each vertex is enqueued and dequeued exactly once ($O(V)$ total), and each edge is examined at most twice across the whole run ($O(E)$ total). So BFS runs in $O(V + E)$ time. With an adjacency matrix, finding the neighbours of a vertex takes $O(V)$ each time, giving $O(V^2)$ total.

8.1.3 BFS Trace: Example Graph

Example 274: Worked Trace: BFS on an Undirected Graph

Consider the undirected graph with vertices $\{A, B, C, D, E\}$ and edges $\{(A, B), (A, C), (B, C), (C, D), (C, E)\}$. Adjacency lists (alphabetical order): $A : [B, C]$, $B : [A, C]$, $C : [A, B, D, E]$, $D : [C]$, $E : [C]$. BFS starts from source A.



Step-by-Step Dequeue Trace

Step	Dequeue	Queue After	Action
Init	—	`[A]`	Mark A gray, $d[A] = 0$
1	**A**	`[B, C]`	$d[B]$=1, $d[C]$=1; A → black
2	**B**	`[C]`	C already gray; B → black
3	**C**	`[D, E]`	$d[D]$=2, $d[E]$=2; C → black
4	**D**	`[E]`	No new neighbours; D → black
5	**E**	`[] empty`	No new neighbours; E → black

Example 275: Filled BFS Table (Final State)

	A	B	C	D	E
$color[]$	BLACK	BLACK	BLACK	BLACK	BLACK
$d[]$	0	1	1	2	2
$\pi[]$	NIL	A	A	C	C

BFS order (dequeue order): A, B, C, D, E

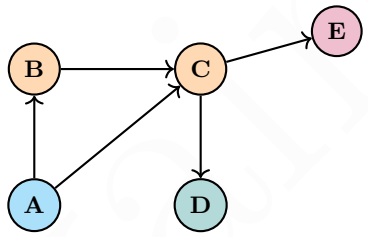
8.1.4 BFS on Directed Graphs

Concept: BFS on Directed Graphs

The core logic and the $O(V+E)$ complexity are **identical** for directed graphs. BFS simply follows the **outbound** edges listed in $Adj[u]$; it never looks at incoming edges.

Example 276: Worked Trace: BFS on the Directed Version of the Same Graph

Take the same 5 vertices, but now the edges are directed: $A \to B$, $A \to C$, $B \to C$, $C \to D$, $C \to E$. BFS from source A.



Example 277: Filled BFS Table for the Directed Graph

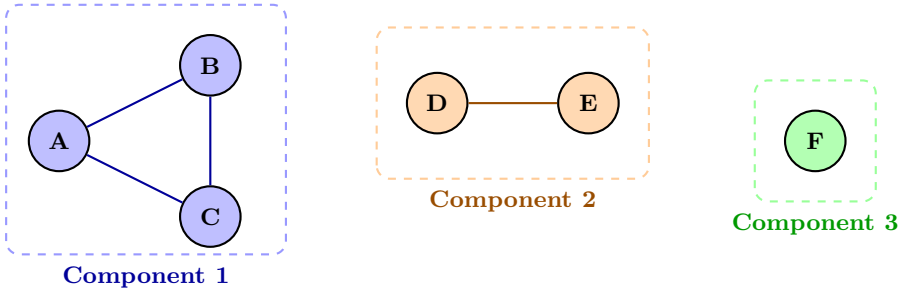
Since B's only out-neighbour C is already discovered (via A) by the time B is processed, the resulting distances and parents come out exactly the same as the undirected case here.

	A	B	C	D	E
$color[]$	BLACK	BLACK	BLACK	BLACK	BLACK
$d[]$	0	1	1	2	2
$\pi[]$	NIL	A	A	C	C

8.1.5 BFS on Disconnected Graph — Connected Components

Concept: Detecting Connected Components with BFS

If the graph is disconnected, a single BFS from one source will not reach all vertices. To find all connected components, we run BFS from every unvisited vertex. Each BFS run that starts fresh (from an unvisited vertex) discovers one complete connected component.
The number of times we start BFS from an unvisited source = the number of connected components.



BFS run 1 from A BFS run 2 from D BFS run 3 from F

3 BFS runs ⇒ 3 Connected Components

8.1.6 Properties of BFS

Property 1: Multiple BFS Traversals

Multiple valid BFS orderings can exist for the **same** graph, depending on:

- which vertex is **chosen as the source**,
- the **order in which neighbours appear** in the adjacency list.

Changing either of these can give a different but equally valid BFS sequence.

Property 2: Distance as Level Number

$d[u]$ equals the exact **level** of u in the BFS tree. In our example graph: $d[A] = 0$ (Level 0), $d[B] = d[C] = 1$ (Level 1), $d[D] = d[E] = 2$ (Level 2).

Property 3: Dequeued Order Defines the BFS Order

The sequence in which vertices are removed from the queue (DEQUEUE calls) **is** the BFS traversal order. For our example:

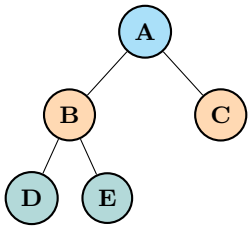
$$A, B, C, D, E$$

Property 4: Shortest Path Length in Unweighted Graphs

In an unweighted graph, $d[u]$ gives the **minimum number of edges** on any path from s to u. In our example, $d[D] = 2$ means the shortest path from A to D uses exactly 2 edges: $A \to C \to D$.

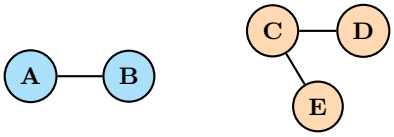
Property 5: Level-Order Traversal on Trees

BFS applied to a **tree** is exactly the same thing as **level-order traversal**. No cycle detection or visited array is even needed, because a tree has no cycles, so a vertex can never be reached a second time.



Property 6: Finding Connected Components

Run BFS from any unvisited (WHITE) vertex. Each full BFS call discovers exactly **one entire connected component**. Repeating this until every vertex has been visited, and counting how many times BFS was started, gives the total number of connected components.

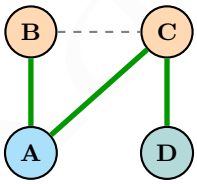


Example 278: Applying Property 6

2 calls to BFS are needed to cover every vertex here, so the graph has **2 connected components**: $\{A, B\}$ and $\{C, D, E\}$.

Property 7: Tree Edges

A **tree edge** is an edge (u, v) that **first discovers** a new WHITE vertex v. The collection of all tree edges forms the **BFS spanning tree**.



Example 279: Applying Property 7

In this graph, the solid green edges (A, B), (A, C), (C, D) are tree edges, since each one first discovers a new vertex. The dashed edge (B, C) connects two vertices that are both already discovered, so it is **not** a tree edge.

Property 8: Cross Edges

A **cross edge** is a non-tree edge connecting two vertices that are **both already discovered**. In an undirected BFS tree, a cross edge always satisfies:

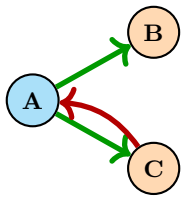
$$|d[u] - d[v]| \leq 1$$

Example 280: Applying Property 8

Edge (B, C) from the previous diagram is a cross edge. Here $d[B] = 1$ and $d[C] = 1$, so $|d[B] - d[C]| = 0 \leq 1$, which confirms it is a valid cross edge.

Property 9: Back Edges

A **back edge** is a directed edge from a **descendant** back to an **ancestor** in the BFS tree. Back edges can only occur in **directed** graphs that contain a cycle; they **never** occur in an undirected BFS tree.

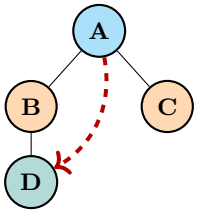


Example 281: Applying Property 9

Edge (C, A) (red, directed back to the root) is a back edge. This situation **cannot** happen in an undirected BFS tree, because if $A \to C$ exists, then (A, C) already becomes a tree edge the moment C is first discovered, which would make $C \to A$ redundant rather than a separate back edge.

Property 10: Forward Edges Never Exist in BFS

A **forward edge** would be a hypothetical edge from an ancestor to a **non-child descendant** further down the tree. Such edges **never appear** in any BFS tree, directed or undirected.



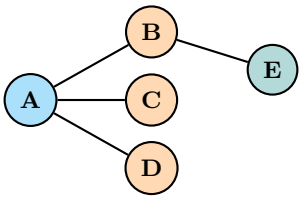
Example 282: Why Forward Edges Never Occur in BFS

BFS always discovers the **shallowest** reachable vertices first. If an edge $A \to D$ really existed, then D would have been discovered directly from A at Level 1, not through B at Level 2. So this hypothetical dashed edge (a "forward edge") is a contradiction, and it simply cannot occur in a BFS tree.

Property 11: Maximum Queue Size

The maximum size the queue ever reaches during BFS equals the **widest level** of the BFS tree:

$$\text{Max Queue Size} = \max_k |\{v \in V : d[v] = k\}|$$



Example 283: Applying Property 11

Level 1 contains 3 vertices (B, C, D), which is the widest level in this graph (Level 0 has only A, Level 2 has only E). So the maximum queue size reached during this BFS run is $\boxed{3}$.

Property 12: Queue Level Monotonicity Invariant

At any point during BFS, if the queue contains $Q = \langle v_1, v_2, \ldots, v_r \rangle$ (front to back), then:

$$d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r], \qquad d[v_r] - d[v_1] \leq 1$$

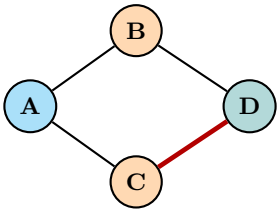
In other words, the queue is always sorted by distance, and it never spans more than **two consecutive levels** at once. You can never have two vertices whose distances differ by 2 or more sitting in the queue at the same time.

Property 13: Odd Cycle Detection via BFS

If, while running BFS, we find an edge (u, v) such that

$$d[u] \equiv d[v] \pmod 2$$

(both distances have the same parity), then the graph contains an **odd-length cycle**.



Example 284: Applying Property 13

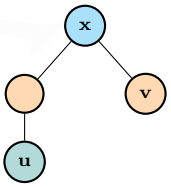
Here $d[C] = 1$ and $d[D] = 1$ (discovered via B). The edge (C, D), shown in red, connects two vertices at the **same** distance (both odd). By the rule, this forces an odd cycle to exist: following $A \to B \to D \to C \to A$ traces out a cycle of length 4 through the tree plus this cross edge; more precisely, the presence of an edge joining two same-parity levels is what certifies an odd cycle exists somewhere in the graph.

Property 14: Tree Diameter via 2-Pass BFS

To find the **diameter** of a tree (the length of the longest path between any two vertices), we do not need to try all pairs. A clever shortcut works:

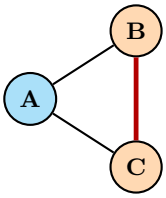
1. Run BFS from any arbitrary vertex x. Find the vertex u that is **farthest** from x.
2. Run BFS again, this time from u. Find the vertex v that is farthest from u.

The distance $d[v]$ found in the second BFS pass is exactly the **diameter** of the tree.



Property 15: Cycle Detection via Parent Check

While BFS explores neighbour v of the current vertex u: if v is already **visited** AND $v \neq \pi[u]$ (i.e., v is not the parent of u), then a **cycle** has been detected.



Example 285: Applying Property 15

BFS from A: visit B, set $\pi[B] = A$. Visit C, set $\pi[C] = A$. When B later checks its neighbour C (the red edge), C is already visited, and $C \neq \pi[B] = A$. This confirms a **cycle** exists: A–B–C–A.

Python implementation: bfs-full-implementation.py

```python
from collections import deque

WHITE, GRAY, BLACK = 0, 1, 2

def bfs(graph, s, vertices):
    color = {v: WHITE for v in vertices}
    d     = {v: float('inf') for v in vertices}
    pi    = {v: None for v in vertices}

    color[s] = GRAY
    d[s]     = 0
    order    = []

    Q = deque([s])
    while Q:
        u = Q.popleft()
        order.append(u)
        for v in graph[u]:
            if color[v] == WHITE:
                color[v] = GRAY
                d[v]     = d[u] + 1
                pi[v]    = u
                Q.append(v)
        color[u] = BLACK

    return order, d, pi


def count_components(graph, vertices):
    visited = {v: False for v in vertices}
    count   = 0
    for v in vertices:
        if not visited[v]:
            count += 1
            Q = deque([v])
            visited[v] = True
            while Q:
                u = Q.popleft()
                for w in graph[u]:
                    if not visited[w]:
                        visited[w] = True
                        Q.append(w)
    return count


def tree_diameter(graph, vertices):
    """Two-pass BFS to find the diameter of a tree."""
    _, d1, _ = bfs(graph, vertices[0], vertices)
    u = max(d1, key=d1.get)          # farthest vertex from arbitrary start
    _, d2, _ = bfs(graph, u, vertices)
    v = max(d2, key=d2.get)          # farthest vertex from u
    return d2[v]                      # diameter


# --- Driver Code ---
vertices = ['A', 'B', 'C', 'D', 'E']
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['C'],
    'E': ['C'],
}

order, d, pi = bfs(graph, 'A', vertices)
print("BFS order:", order)    # ['A', 'B', 'C', 'D', 'E']
print("Distances:", d)        # {'A':0,'B':1,'C':1,'D':2,'E':2}
```

```
print("Parents:", pi)          # {'A':None,'B':'A','C':'A','D':'C','E':'C'}
```

8.2 Depth-First Search (DFS)

Concept: DFS — The Idea

DFS explores a graph by going as **deep as possible** along each path before backtracking. Starting from a source vertex, it visits one neighbour, then one of that neighbour's neighbours, and keeps going deeper until it reaches a vertex with no unvisited neighbours. Then it backtracks and tries another direction.
DFS uses a **stack** (or recursion, which uses the call stack implicitly). Think of exploring a maze by always taking the first available turn, going as far as you can, then backing up when you hit a dead end.

Example 286: Analogy: Exploring a Cave System

Imagine exploring a cave with many tunnels branching off. You always take the first unexplored tunnel and go as deep as possible. When you hit a dead end, you back up to the last junction and take the next unexplored tunnel. DFS works exactly this way.

8.2.1 DFS Discovery and Finish Timestamps

Concept: Discovery and Finish Times

In DFS, we maintain a global counter (clock). Two timestamps are recorded for each vertex v:

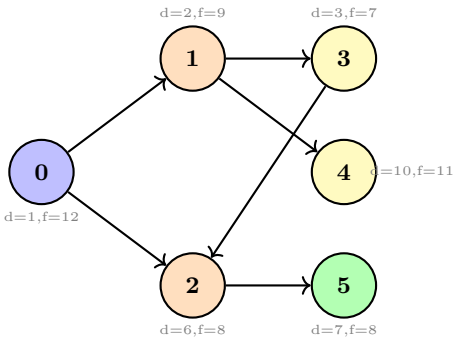
- **Discovery time** $d[v]$: the clock value when v is first visited (entered).
- **Finish time** $f[v]$: the clock value when DFS has finished exploring all of v's descendants (exited).

For any vertex v, $d[v] < f[v]$. If u is an ancestor of v in the DFS tree, then $d[u] < d[v] < f[v] < f[u]$, meaning the interval $[d[v], f[v]]$ is completely inside $[d[u], f[u]]$.
This **parenthesis theorem** is very important for classifying edges.

8.2.2 DFS Step-by-Step Trace with Timestamps

Example 287: DFS Trace on Directed Graph

Graph with 6 vertices, edges: $0 \to 1, 0 \to 2, 1 \to 3, 1 \to 4, 2 \to 5, 3 \to 2$. Start from vertex 0. Neighbours visited in increasing order.



DFS Trace with Timestamps: clock starts at 0

Step	Action	Stack (top right)	Timestamps Updated
1	Enter 0, d[0]=1	0	d[0]=1
2	Enter 1 (first unvisited nbr of 0), d[1]=2	0,1	d[1]=2
3	Enter 3 (first unvisited nbr of 1), d[3]=3	0,1,3	d[3]=3
4	Enter 2 (first unvisited nbr of 3), d[2]=4	0,1,3,2	d[2]=4
5	Enter 5 (first unvisited nbr of 2), d[5]=5	0,1,3,2,5	d[5]=5
6	No unvisited nbr of 5. Exit 5, f[5]=6	0,1,3,2	f[5]=6
7	No more unvisited nbr of 2. Exit 2, f[2]=7	0,1,3	f[2]=7
8	No more unvisited nbr of 3. Exit 3, f[3]=8	0,1	f[3]=8
9	Enter 4 (next unvisited nbr of 1), d[4]=9	0,1,4	d[4]=9
10	No unvisited nbr of 4. Exit 4, f[4]=10	0,1	f[4]=10
11	No more unvisited nbr of 1. Exit 1, f[1]=11	0	f[1]=11
12	Nbr 2 already visited. Exit 0, f[0]=12		f[0]=12

DFS Order: 0, 1, 3, 2, 5, 4. Timestamps: d/f as shown above.

8.2.3 DFS Edge Classification

Concept: Four Types of Edges in DFS

During DFS on a **directed graph**, every edge (u, v) falls into exactly one of four categories:

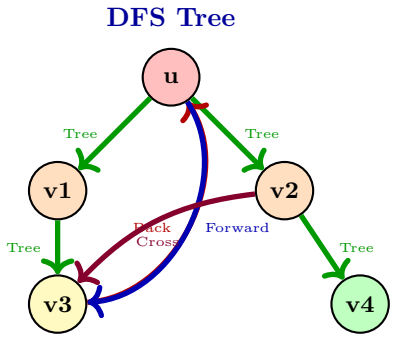
- **Tree Edge:** (u, v) is a tree edge if v was discovered for the first time through u. It is the edge used to discover a new vertex. It appears in the DFS tree.
- **Back Edge:** (u, v) is a back edge if v is an **ancestor** of u in the DFS tree (i.e., v is still on the recursion stack when we see this edge). Back edges indicate a **cycle** in a directed graph.
- **Forward Edge:** (u, v) is a forward edge if v is a **descendant** of u in the DFS tree, but v was not discovered directly through u (it was discovered through some other path). Identified by $d[u] < d[v]$ and v already finished.
- **Cross Edge:** (u, v) is a cross edge if v is in a different DFS subtree (not an ancestor or descendant of u). Identified by $d[v] < d[u]$ and v already finished.

Example 288: Quick Identification Rule

When DFS is exploring vertex u and encounters edge (u, v):

- If v is **unvisited**: Tree Edge.
- If v is **currently on the DFS stack** (visited but not finished): Back Edge.
- If v is **already finished** and $d[u] < d[v]$: Forward Edge.
- If v is **already finished** and $d[u] > d[v]$: Cross Edge.

DFS Edge Classification Summary



Edge Types

TreeEdge	v unvisited when (u, v) explored
BackEdge	v is ancestor of u (on stack)
ForwardEdge	v is descendant, $d[u] < d[v]$, v finished
CrossEdge	v in diff subtree, $d[v] < d[u]$, v finished

Key GATE Fact:
Back edge exists in directed graph $\Leftrightarrow$ cycle exists
Undirected DFS: only tree edges and back edges (no forward or cross)

Algorithm: DFS(G)

```
clock ← 0
for each vertex u ∈ V do
    if u is not visited then
        DFS-Visit(u)
    end if
end for
```

Algorithm: DFS-Visit(u)

```
Mark u as visited (on stack)
clock ← clock + 1; d[u] ← clock
for each neighbour v of u do
    if v is not visited then
        parent[v] ← u
        DFS-Visit(v)                          ▷ Tree edge
    else if v is on stack then
        record (u, v) as Back Edge            ▷ cycle detected
    end if
end for
clock ← clock + 1; f[u] ← clock
Mark u as finished (off stack)
```

Python implementation: dfs.py

```python
def dfs_full(graph, n):
    visited  = [False] * n
    on_stack = [False] * n    # currently in recursion stack
    d        = [0] * n        # discovery time
    f        = [0] * n        # finish time
    parent   = [-1] * n
    clock    = [0]            # use list so nested function can modify it
    order    = []
    has_cycle = [False]

    def dfs_visit(u):
        visited[u]  = True
        on_stack[u] = True
        clock[0]   += 1
        d[u]        = clock[0]
        order.append(u)

        for v in graph[u]:
            if not visited[v]:
                parent[v] = u
                dfs_visit(v)
```

```
            elif on_stack[v]:
                has_cycle[0] = True    # back edge = cycle

        clock[0]    += 1
        f[u]         = clock[0]
        on_stack[u] = False

    for i in range(n):
        if not visited[i]:
            dfs_visit(i)

    return order, d, f, parent, has_cycle[0]


# --- Driver Code ---
# Directed graph: 0->1, 0->2, 1->3, 1->4, 2->5, 3->2
n     = 6
graph = {0:[1,2], 1:[3,4], 2:[5], 3:[2], 4:[], 5:[]}

order, d, f, parent, cycle = dfs_full(graph, n)
print("DFS order:", order)
print("Discovery:", d)
print("Finish:   ", f)
print("Has cycle:", cycle)

# Output:
# DFS order: [0, 1, 3, 2, 5, 4]
# Discovery: [1, 2, 4, 3, 9, 5]
# Finish:    [12, 11, 8, 8, 10, 6]
# Has cycle: False
```

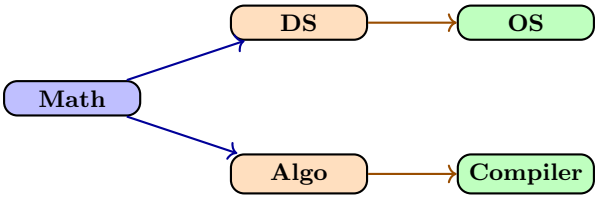
8.3 Topological Sort

Concept: What is Topological Sort

A **topological sort** (or topological ordering) of a **DAG** (Directed Acyclic Graph) is a linear ordering of all its vertices such that for every directed edge (u, v), vertex u appears before vertex v in the ordering.
Think of scheduling tasks where some tasks must be completed before others. For example, you must wear socks before shoes, wear shirt before jacket. A topological sort gives a valid sequence of completing all tasks.
Topological sort is only defined for DAGs. If the graph has a cycle, no topological ordering exists.

Example 289: Course Prerequisites Example

Suppose courses have prerequisites: Math before DS, Math before Algo, DS before OS, Algo before Compiler. A topological sort gives a valid order to take these courses without violating any prerequisite.



Valid topological orders:

Math → DS → Algo → OS → Compiler

Math → Algo → DS → Compiler → OS

8.3.1 Method 1: DFS-Based Topological Sort

Concept: DFS Topological Sort

When DFS finishes a vertex u (assigns finish time), we push u onto a stack. After DFS completes for all vertices, we pop the stack to get the topological order. The vertex with the largest finish time comes first in the topological order.
Why it works: If there is an edge (u, v), then DFS on u must complete all of v's work before u finishes. So $f[v] < f[u]$, meaning u finishes after v, so u appears before v when we output in decreasing finish time order.

Example 290: DFS Topological Sort Trace

Graph: $0 \to 1, 0 \to 2, 1 \to 3, 2 \to 3$. This is a DAG.
DFS from 0: discover 0, then 1, then 3. Finish 3 (push 3). Back to 1, finish 1 (push 1). Back to 0, go to 2, discover 2. Finish 2 (wait, 3 already finished). Finish 2 (push 2). Finish 0 (push 0).
Stack (top to bottom): 0, 2, 1, 3. Pop gives: **0, 2, 1, 3**.
Verify: edge $0 \to 1$: 0 before 1 ✓, edge $0 \to 2$: 0 before 2 ✓, edge $1 \to 3$: 1 before 3 ✓, edge $2 \to 3$: 2 before 3 ✓.

Algorithm: Topological-Sort-DFS(G)

```
Mark all vertices unvisited
Initialize empty stack S
for each vertex u ∈ V do
    if u is not visited then
        Topo-DFS-Visit(u, S)
    end if
end for
while S is not empty do
    print Pop(S)                ▷ decreasing finish time = topological order
end while
```

Algorithm: Topo-DFS-Visit(u, S)

```
Mark u as visited
for each neighbour v of u do
    if v is not visited then
        Topo-DFS-Visit(v, S)
    end if
end for
Push(S, u)                      ▷ push AFTER all descendants are processed
```

8.3.2 Method 2: Kahn's Algorithm (BFS with In-Degree)

Concept: Kahn's Algorithm

Kahn's algorithm is an iterative BFS-based method. The key insight is: in a topological ordering, the first vertex must have in-degree 0 (no prerequisites). After removing it, the next vertex with in-degree 0 can come next, and so on.
Steps:

1. Compute the in-degree of every vertex.
2. Add all vertices with in-degree 0 to a queue.
3. While the queue is not empty:
   (a) Dequeue vertex u, add it to the topological order.
   (b) For each neighbour v of u: decrease v's in-degree by 1. If v's in-degree becomes 0, enqueue v.
4. If the topological order contains fewer than $|V|$ vertices, the graph has a cycle.

This also detects cycles: if we cannot empty the queue before visiting all vertices, a cycle exists.

Kahn's Algorithm Trace: Graph $0 \to 2, 0 \to 1, 1 \to 3, 2 \to 3, 3 \to 4$

Initial in-degrees:

$v0$: 0 $v1$: 1 $v2$: 1 $v3$: 2 $v4$: 1

Step	Dequeue	Queue	In-degree updates
Init	—	`[0]`	Only 0 has in-degree 0
1	**0**	`[1, 2]`	in[1]: 1→0, in[2]: 1→0; enqueue 1, 2
2	**1**	`[2, 3]`	in[3]: 2→1; enqueue 3? No (in[3]=1)
3	**2**	`[3]`	in[3]: 1→0; now enqueue 3
4	**3**	`[4]`	in[4]: 1→0; enqueue 4
5	**4**	`[ ] empty`	No outgoing edges from 4. Done

Topological Order: 0, 1, 2, 3, 4

Algorithm: Kahn-Topological-Sort(G)

```
Compute indegree[v] for all v ∈ V
Initialize empty queue Q
for each vertex v ∈ V do
    if indegree[v] = 0 then
        Enqueue(Q, v)
    end if
end for
count ← 0, result ← empty list
while Q is not empty do
    u ← Dequeue(Q)
    Append u to result
    count ← count + 1
    for each neighbour v of u do
        indegree[v] ← indegree[v] − 1
        if indegree[v] = 0 then
            Enqueue(Q, v)
        end if
    end for
end while
if count ≠ |V| then
    error "Graph has a cycle — no topological sort"
end if
return result
```

Example 291: Multiple Valid Topological Orderings

A DAG may have many valid topological orderings. The order in which we add zero-in-degree vertices to the queue (and which queue we use) determines the specific ordering produced. For the graph $0 \to 2, 0 \to 1, 1 \to 3, 2 \to 3$:

- If we process 1 before 2 at step 2: order is 0, 1, 2, 3.
- If we process 2 before 1 at step 2: order is 0, 2, 1, 3.

Both are valid. The DAG does not have a unique topological ordering unless it is a Hamiltonian path (each vertex has at most one zero-in-degree vertex at any time during Kahn's).

Python implementation: topological-sort.py

```
from collections import deque

# ============================================================
# Method 1: DFS-Based Topological Sort
# ============================================================
def topo_sort_dfs(graph, n):
    visited = [False] * n
    stack   = []

    def dfs_visit(u):
        visited[u] = True
        for v in graph[u]:
            if not visited[v]:
                dfs_visit(v)
        stack.append(u)    # push AFTER all descendants

    for i in range(n):
        if not visited[i]:
            dfs_visit(i)

    return stack[::-1]    # reverse = topological order


# ============================================================
# Method 2: Kahn's Algorithm (BFS with in-degree)
# ============================================================
def kahn_topo_sort(graph, n):
    indegree = [0] * n
    for u in range(n):
        for v in graph[u]:
            indegree[v] += 1

    queue = deque()
    for v in range(n):
        if indegree[v] == 0:
            queue.append(v)

    result = []
    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    if len(result) != n:
        print("Cycle detected! No topological sort possible.")
        return []
    return result


# --- Driver Code ---
# DAG: 0->2, 0->1, 1->3, 2->3, 3->4
```

```
n     = 5
graph = {0:[1,2], 1:[3], 2:[3], 3:[4], 4:[]}

print("DFS Topo Sort:", topo_sort_dfs(graph, n))
# Output: [0, 2, 1, 3, 4] or [0, 1, 2, 3, 4]

print("Kahn's Topo Sort:", kahn_topo_sort(graph, n))
# Output: [0, 1, 2, 3, 4]

# Test with cycle
graph_cycle = {0:[1], 1:[2], 2:[0]}
print("Cycle graph:", kahn_topo_sort(graph_cycle, 3))
# Output: Cycle detected! []
```

BFS vs DFS vs Topological Sort: Summary

Property	BFS	DFS	Topo Sort (Kahn)
Data structure	Queue (FIFO)	Stack / Recursion	Queue + in-degree
Time complexity	$O(V+E)$ (adj list)	$O(V+E)$ (adj list)	$O(V+E)$
Traversal style	Level by level	Deep first, backtrack	In-degree driven
Shortest path	Yes (unweighted)	No	No
Cycle detection	Indirect (components)	Back edge = cycle	result size $<$ V
Requires DAG	No	No	Yes
Applications	Shortest path, levels	Cycle detect, topo sort	Task scheduling, build

Example 292: GATE Exam Tips

- **BFS/DFS tracing:** Always maintain a queue/stack carefully. Follow the adjacency list order strictly. Commonly asked: what is the BFS/DFS order, what are the timestamps, what type is each edge.
- **Edge classification:** On a directed graph DFS, a back edge means a cycle. On an undirected graph DFS, there are no forward or cross edges, only tree and back edges.
- **Topological sort:** Only on DAGs. Multiple valid orderings may exist. Kahn's algorithm also detects cycles. DFS-based gives reverse of finish time.
- **Complexity:** $O(V+E)$ on adjacency list, $O(V^2)$ on adjacency matrix. Always specify which representation.
- **BFS shortest path:** Only gives shortest in terms of edge count (unweighted graphs). Not for weighted graphs.

8.4 Practice Problems

Problem 273 ***[MSQ]*** *Consider a graph with 6 vertices and edges:* $(0,1),(0,2),(1,3),(2,3),(3,4),(4,5)$. *BFS is run from vertex 0 with neighbours visited in increasing order. What is the BFS order of vertices visited?*

(A) 0, 1, 2, 3, 4, 5

(B) 0, 2, 1, 3, 4, 5

(C) 0, 1, 3, 2, 4, 5

(D) 0, 1, 2, 4, 3, 5

Problem 274 ***[NAT]*** *Consider a directed graph with 5 vertices* $V = \{0,1,2,3,4\}$ *represented by the following adjacency list, where adjacent vertices are ordered in increasing numerical order:*

$Adj[0] = [1,2]$
$Adj[1] = [3]$
$Adj[2] = [3]$
$Adj[3] = [4]$
$Adj[4] = []$

A Depth-First Search (DFS) is run starting from vertex 0*, exploring neighbors in the exact order specified by the adjacency list. The discovery time* $d[v]$ *and finish time* $f[v]$ *are recorded using a global clock that starts at* 1 *and increments by* 1 *for each discovery and finish event.*
The finish time of vertex 2 is _____.

Problem 275 ***[MSQ]*** *Which of the following statements are TRUE regarding Breadth-First Search (BFS) on an undirected graph?*

(A) BFS always finds the shortest path in terms of edge count between two connected vertices.

(B) BFS can be used to find all connected components of an undirected graph.

(C) BFS runs in $O(V^2)$ *time when the graph is represented using an adjacency list.*

(D) The BFS traversal starting from a vertex forms a spanning tree for the connected component containing that vertex.

Problem 276 ***[MCQ]*** *During DFS of a directed graph, edge* (u,v) *is classified as a cross edge when:*

(A) v *is not yet visited when* (u,v) *is explored.*

(B) v *is an ancestor of* u *and currently on the DFS stack.*

(C) v *is already finished and* $d[u] > d[v]$.

(D) v *is a descendant of* u *and* $d[u] < d[v]$ *with* v *already finished.*

Problem 277 ***[NAT]*** *A DAG has vertices* $\{A,B,C,D,E\}$ *with edges:* $A \to C, A \to B, B \to D, C \to D, D \to E$. *The number of distinct valid topological orderings is* _____.

Problem 278 ***[MCQ]*** *A DFS is performed on a directed acyclic graph (DAG). For an edge* $(p,q) \in E$ *that is classified as a* ***forward edge****, which of the following timestamp relations must hold?*

(A) $d[q] < d[p] < f[p] < f[q]$

(B) $d[p] < d[q] < f[q] < f[p]$

(C) $d[p] < f[p] < d[q] < f[q]$

(D) $d[q] < f[q] < d[p] < f[p]$

Problem 279 ***[MSQ]*** *A DFS is run on an undirected connected graph* G*, producing a DFS tree* T. *Which of the following are TRUE?*

(A) Every non-tree edge of G *with respect to* T *must be a back edge.*

(B) G *can never have a cross edge with respect to* T.

(C) A forward edge can exist with respect to T.

(D) For every edge $(u, v) \in G$, one of u, v must be an ancestor of the other in T.

Problem 280 ***[NAT]*** *DFS is performed on an undirected graph starting at an unspecified vertex. Each recursive call to visit a vertex is made only after confirming the vertex has not been visited. For the graph with vertex set $\{1, 2, 3, 4, 5, 6, 7\}$ and edges $\{(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 5)\}$, the maximum possible recursion depth (including the initial call) over all choices of starting vertex and neighbour-visit order is _____.*

Problem 281 ***[MSQ]*** *Let G be a simple undirected graph, T_D a DFS tree of G, and T_B a BFS tree of G (both rooted at the same vertex). Consider the statements: (I) No edge of G is a cross edge with respect to T_D. (II) For every edge (u, v) of G, if u is at depth i and v at depth j in T_B, then $|i - j| \leq 1$. Which of the following are TRUE?*

(A) Statement (I) must necessarily be true for every such G and T_D.

(B) Statement (II) must necessarily be true for every such G and T_B.

(C) Statement (I) can be false for some graph G.

(D) Both statements are simultaneously guaranteed true for all simple undirected graphs.

Problem 282 ***[MCQ]*** *Consider the following algorithm `twoPassBFS` applied to a tree T with 9 vertices arranged as a path $1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9$, with one extra branch: vertex 5 also connects to vertex 10.*

```
1. Pick any vertex v. Run BFS from v. Let u = farthest vertex from v.
2. Run BFS from u. Let z = farthest vertex from u.
3. Output distance between u and z.
```

What value does this algorithm output?

(A) 7

(B) 8

(C) 9

(D) 10

Problem 283 ***[MCQ]*** *Consider a complete binary tree with 15 nodes. Let A be the set of the first 4 vertices visited by BFS starting from the root, and B be the set of the first 4 vertices visited by DFS (preorder) starting from the root, always visiting the left child before the right child. The value of $|A - B|$ is:*

(A) 0

(B) 1

(C) 2

(D) 3

Problem 284 ***[MCQ]*** *Let G be a directed graph and T a DFS spanning tree of G rooted at vertex v, such that T is **also** a valid BFS tree of G rooted at v. Which of the following must be TRUE for every such G and T?*

(A) G has no back edges and no cross edges with respect to T.

(B) G has no forward edges with respect to T, but may have cross edges.

(C) Every edge of G must be an edge of T (i.e., $G = T$).

(D) G must be acyclic.

Problem 285 ***[MCQ]*** *The most efficient known algorithm to find the number of connected components of an undirected graph with n vertices and m edges has worst-case time complexity:*

(A) $\Theta(n)$

(B) $\Theta(m)$

(C) $\Theta(m + n)$

(D) $\Theta(mn)$

Problem 286 ***[MCQ]*** *Consider a directed graph with vertices $\{P, Q, R, S, T\}$ and edges $P \to Q$, $Q \to R$, $R \to P$, $R \to S$, $S \to T$, $T \to S$. The strongly connected components of this graph are:*

(A) $\{P, Q, R\}, \{S, T\}$

(B) $\{P, Q, R, S, T\}$

(C) $\{P\}, \{Q\}, \{R\}, \{S, T\}$

(D) $\{P, Q, R, S\}, \{T\}$

Problem 287 ***[MSQ]*** *Consider the DAG with vertices* $\{P, Q, R, S, T\}$ *and edges* $P \to Q$, $P \to R$, $Q \to S$, $R \to S$, $S \to T$. *Which of the following is NOT a valid topological ordering?*

(A) P, Q, R, S, T

(B) P, R, Q, S, T

(C) Q, P, R, S, T

(D) P, Q, S, R, T

Problem 288 ***[MCQ]*** *Let* G *be a directed graph with vertices* $\{A, B, C, D\}$ *and edges* $A \to B$, $B \to C$, $C \to D$, $D \to A$, $A \to C$. *Which of the following is/are correct about* G*?*

(A) G *does not have a topological order.*

(B) A DFS starting at A *classifies exactly one edge as a back edge.*

(C) G *is strongly connected.*

(D) For every pair of vertices u, v*, there is a directed path from* u *to* v*.*

Problem 289 ***[MCQ]*** *Consider a simple undirected unweighted graph where the vertex set consists of all bit strings of length 4 (so* $|V| = 16$*), with an edge between* u *and* v *iff they differ in exactly one bit. If a BFS is run from the all-zero vertex* 0000*, the distance* $d(0000, v)$ *for a vertex* v *equals:*

(A) The position of the first 1 bit in v

(B) The number of 1 bits in v

(C) Always 4, regardless of v

(D) The decimal value of v *modulo 4*

Problem 290 ***[MCQ]*** *A DFS is run on an undirected graph* G *with 8 vertices, producing discovery/finish time pairs:* $A(1, 16)$, $B(2, 9)$, $C(3, 6)$, $D(4, 5)$, $E(7, 8)$, $F(10, 15)$, $G(11, 12)$, $H(13, 14)$. *Based purely on these timestamps, which pair is guaranteed to be an ancestor-descendant pair in the DFS tree?*

(A) B *and* F

(B) A *and* F

(C) C *and* E

(D) D *and* G

Problem 291 ***[MCQ]*** *Consider an undirected graph where BFS from source* s *assigns distances* $d(s) = 0$*, and two vertices* p, q *satisfy* $d(p) = 3$ *and* $d(q) = 3$. *Which of the following is impossible?*

(A) (p, q) *is an edge of* G *but not a tree edge of the BFS tree.*

(B) p *is the parent of* q *in the BFS tree.*

(C) p *and* q *have the same parent in the BFS tree.*

(D) (p, q) *is an edge that is a cross edge with respect to the BFS tree.*

Problem 292 ***[MCQ]*** *In an undirected graph* G*, BFS from source* s *produces tree* T. *Let* (x, y) *be a non-tree edge of* G *with respect to* T*, where* x *is discovered before* y. *Which of the following is the* ***tightest*** *correct bound relating* $d(x)$ *and* $d(y)$*?*

(A) $d(y) = d(x) + 1$ *or* $d(y) = d(x)$

(B) $d(y) \geq d(x)$

(C) $d(y) \leq d(x) + 1$ *and* $d(y) \geq d(x) - 1$

(D) $d(y) > d(x)$ *always*

8.5 GATE PYQs

GATEPYQ 201 (GATE DA 2026) *Consider a directed graph $G = (V, E)$, where V is the finite set of vertices and E is the set of directed edges between the vertices. G may contain cycles but there is no self-loop. Further, G may not be strongly connected.*

Let G^R be the graph obtained by reversing the directions of all the edges in G without changing the set of vertices.

Assume that Breadth First Search (BFS) or Depth First Search (DFS) from any given vertex v of a graph visits only the reachable vertices from v in that graph.

Which of the following statements must always be true, regardless of the structure of G?

A. *If u is a reachable vertex in the BFS of G^R from v, then u is also a reachable vertex in the DFS of G from v.*

B. *In G^R, the BFS traversal from v will visit exactly the same set of vertices as the DFS from v in G.*

C. *The order of vertices visited in the BFS of G^R from v is the reverse of the order of vertices visited in the DFS of G from v.*

D. *If u is a reachable vertex in the DFS of G from v, then v is also a reachable vertex in the BFS of G^R from u.*

GATEPYQ 202 (GATE CSE 2026) *Consider the following pseudocode for depth-first search (DFS) algorithm which takes a directed graph $G(V, E)$ as input, where $d[v]$ and $f[v]$ are the discovery time and finishing time, respectively, of the vertex $v \in V$.*

```
DFS(G):
    unmark all v in V
    t <- 0
    for each v in V
        if v is unmarked
            t <- Explore(G, v, t)
        end if
    end for

Explore(G, v, t):
    mark v
    t <- t + 1
    d[v] <- t
    for each (v, w) in E
        if w is unmarked
            t <- Explore(G, w, t)
        end if
    end for
    t <- t + 1
    f[v] <- t
    return t
```

Suppose that the input directed graph $G(V, E)$ is a directed acyclic graph (DAG). For an edge $(u, v) \in E$, which of the following options will NEVER be correct?

A. $d[u] < d[v] < f[v] < f[u]$

B. $d[v] < d[u] < f[u] < f[v]$

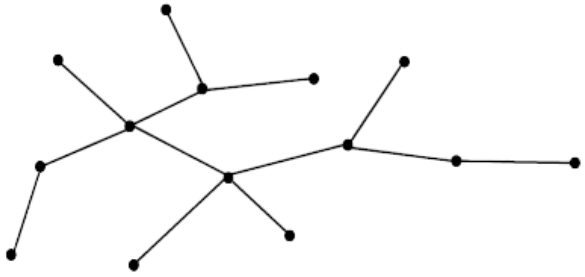
C. $d[v] < f[v] < d[u] < f[u]$

D. $d[u] < d[v] < f[u] < f[v]$

GATEPYQ 203 (GATE CSE 2025) *Consider the following algorithm `someAlgo` that takes an undirected graph G as input.*

```
someAlgo(G):
1. Let v be any vertex in G. Run BFS on G starting at v. Let u be a vertex in G
   at maximum distance from v as given by the BFS.
2. Run BFS on G again with u as the starting vertex. Let z be the vertex at
   maximum distance from u as given by the BFS.
3. Output the distance between u and z in G.
```

The output of `someAlgo(T)` for the tree shown in the given figure is _____.



GATEPYQ 204 (GATE CSE 2025) *Which of the following statements regarding Breadth First Search (BFS) and Depth First Search (DFS) on an undirected simple graph G is/are TRUE?*

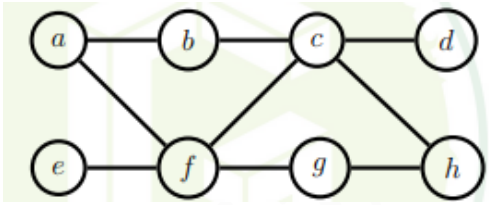
A. *A DFS tree of G is a Shortest Path tree of G.*

B. *Every non-tree edge of G with respect to a DFS tree is a forward/back edge.*

C. *If (u, v) is a non-tree edge of G with respect to a BFS tree, then the distances from the source vertex s to u and v in the BFS tree are within ± 1 of each other.*

D. *Both BFS and DFS can be used to find the connected components of G.*

GATEPYQ 205 (GATE DA 2025) *Let G be a simple, unweighted, and undirected graph. A subset of the vertices and edges of G are shown below.*



It is given that $a-b-c-d$ is a shortest path between a and d; $e-f-g-h$ is a shortest path between e and h; $a-f-c-h$ is a shortest path between a and h. Which of the following is/are NOT the edges of G?

A. (b, d)

B. (b, g)

C. (b, h)

D. (e, g)

GATEPYQ 206 (GATE DA 2025) *Consider a directed graph $G = (V, E)$, where $V = \{0, 1, 2, \ldots, 100\}$ and $E = \{(i, j) : 0 < j - i \leq 2,$ for all $i, j \in V\}$. Suppose the adjacency list of each vertex is in decreasing order of vertex number, and depth-first search (DFS) is performed at vertex 0. The number of vertices that will be discovered after vertex 50 is _____.*

GATEPYQ 207 (GATE DA 2024) *Consider performing depth-first search (DFS) on an undirected and unweighted graph G starting at vertex s. For any vertex u in G, $d[u]$ is the length of the shortest path from s to u. Let (u, v) be an edge in G such that $d[u] < d[v]$. If the edge (u, v) is explored first in the direction from u to v during the above DFS, then (u, v) becomes a _____ edge.*

A. *tree*

B. *cross*

C. *back*

D. *gray*

GATEPYQ 208 (GATE DA 2024) *Consider a state space where the start state is number 1. The successor function for the state numbered n returns two states numbered $n+1$ and $n+2$. Assume that the states in the unexpanded state list are expanded in the ascending order of numbers and the previously expanded states are not added to the unexpanded state list.*

Which ONE of the following statements about breadth-first search (BFS) and depth-first search (DFS) is true, when reaching the goal state number 6?

A. *BFS expands more states than DFS.*

B. *DFS expands more states than BFS.*

C. *Both BFS and DFS expand equal number of states.*

D. *Both BFS and DFS do not reach the goal state number 6.*

GATEPYQ 209 (GATE CSE 2024) *Let G be a directed graph and T a depth first search (DFS) spanning tree in G that is rooted at a vertex v. Suppose T is also a breadth first search (BFS) tree in G, rooted at v. Which of the following statements is/are TRUE for every such graph G and tree T ?*

A. *There are no back-edges in G with respect to the tree T*

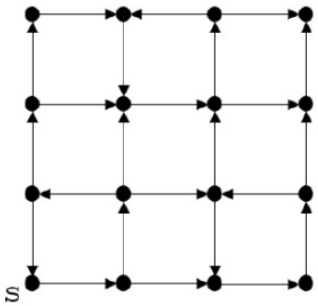
B. *There are no cross-edges in G with respect to the tree T*

C. *There are no forward-edges in G with respect to the tree T*

D. *The only edges in G are the edges in T*

GATEPYQ 210 (GATE CSE 2021) *Consider a complete binary tree with 7 nodes. Let A denote the set of first 3 elements obtained by performing Breadth-First Search (BFS) starting from the root. Let B denote the set of first 3 elements obtained by performing Depth-First Search (DFS) starting from the root.*
The value of $|A - B|$ is _____.

GATEPYQ 211 (GATE CSE 2021) *Consider the following directed graph:*



Which of the following is/are correct about the graph?

A. *The graph does not have a topological order*

B. *A depth-first traversal starting at vertex S classifies three directed edges as back edges*

C. *The graph does not have a strongly connected component*

D. *For each pair of vertices u and v, there is a directed path from u to v*

GATEPYQ 212 (GATE CSE 2018) *Let G be a simple undirected graph. Let T_D be a depth first search tree of G. Let T_B be a breadth first search tree of G. Consider the following statements.*
(I) No edge of G is a cross edge with respect to T_D. (A cross edge in G is between two nodes neither of which is an ancestor of the other in T_D.)
(II) For every edge (u, v) of G, if u is at depth i and v is at depth j in T_B, then $|i - j| = 1$.
Which of the statements above must necessarily be true?

A. *I only*

B. *II only*

C. *Both I and II*

D. *Neither I nor II*

GATEPYQ 213 (GATE CSE 2016) *In an adjacency list representation of an undirected simple graph $G = (V, E)$, each edge (u, v) has two adjacency list entries: $[v]$ in the adjacency list of u, and $[u]$ in the adjacency list of v. These are called twins of each other. A twin pointer is a pointer from an adjacency list entry to its twin. If $|E| = m$ and $|V| = n$, and the memory size is not a constraint, what is the asymptotic complexity of the most efficient algorithm to set the twin pointer in each entry in each adjacency list?*

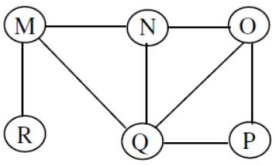
A. $\Theta(n^2)$

B. $\Theta(n+m)$

C. $\Theta(m^2)$

D. $\Theta(n^4)$

GATEPYQ 214 (GATE CSE 2017) *The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?*



A. *MNOPQR*

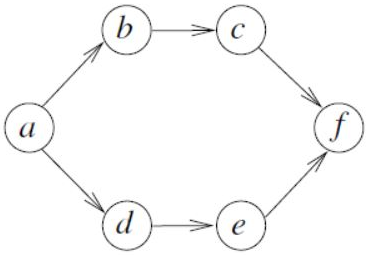
B. *NQMPOR*

C. *QMNROP*

D. *POQNMR*

GATEPYQ 215 (GATE CSE 2016) *Breadth First Search(BFS) is started on a binary tree beginning from the root vertex. There is a vertex t at a distance four from the root. If t is the n-th vertex in this BFS traversal, then the maximum possible value of n is _____.*

GATEPYQ 216 (GATE CSE 2016) *Consider the following directed graph:*



The number of different topological orderings of the vertices of the graph is _____.

GATEPYQ 217 (GATE CSE 2015) *Let $G = (V, E)$ be a simple undirected graph, and s be a particular vertex in it called the source. For $x \in V$, let $d(x)$ denote the shortest distance in G from s to x. A breadth first search (BFS) is performed starting at s. Let T be the resultant BFS tree. If (u, v) is an edge of G that is not in T, then which one of the following CANNOT be the value of $d(u) - d(v)$?*

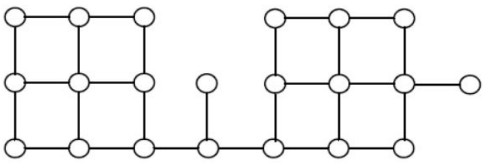
A. *-1*

B. *0*

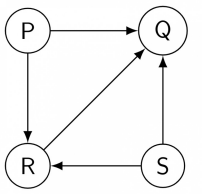
C. *1*

D. *2*

GATEPYQ 218 (GATE CSE 2014) *Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is _____.*



GATEPYQ 219 (GATE CSE 2014) *Consider the directed graph given below.*



Which one of the following is TRUE?

A. *The graph does not have any topological ordering*

B. *Both PQRS and SRQP are topological orderings*

C. *Both PSRQ and SPRQ are topological orderings.*

D. *PSRQ is the only topological ordering*

GATEPYQ 220 (GATE CSE 2014) *Let G be a graph with n vertices and m edges. What is the tightest upper bound on the running time of Depth First Search on G, when G is represented as an adjacency matrix?*

A. $\Theta(n)$

B. $\Theta(n+m)$

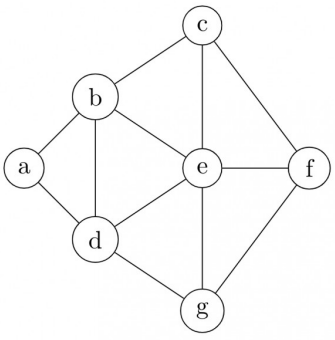
C. $\Theta(n^2)$

D. $\Theta(m^2)$

GATEPYQ 221 (GATE IT 2008) *Consider the following sequence of nodes for the undirected graph given below:*

1. *a b e f d g c*
2. *a b e f c g d*
3. *a d g e b c f*
4. *a d b c g e f*

A Depth First Search (DFS) is started at node a. The nodes are listed in the order they are first visited. Which of the above is/are possible output(s)?



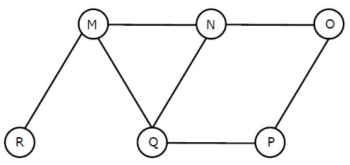
A. *1 and 3 only*

B. *2 and 3 only*

C. *2, 3 and 4 only*

D. *1, 2 and 3 only*

GATEPYQ 222 (GATE CSE 2008) *The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is*



A. *MNOPQR*

B. *NQMPOR*

C. *QMNPRO*

D. *QMNPOR*

GATEPYQ 223 (GATE CSE 2008) *The most efficient algorithm for finding the number of connected components in an undirected graph on n vertices and m edges has time complexity*

A. $\Theta(n)$

B. $\Theta(m)$

C. $\Theta(m+n)$

D. $\Theta(mn)$

GATEPYQ 224 (GATE IT 2007) *A depth-first search is performed on a directed acyclic graph. Let d[u] denote the time at which vertex u is visited for the first time and f[u] the time at which the DFS call to the vertex u terminates. Which of the following statements is always TRUE for all edges (u, v) in the graph?*

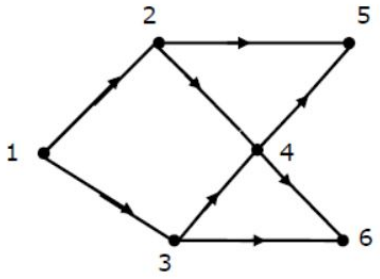
A. $d[u] < d[v]$

B. $d[u] < f[v]$

C. $f[u] < f[v]$

D. $f[u] > f[v]$

GATEPYQ 225 (GATE CSE 2007) *Consider the DAG with $V = \{1, 2, 3, 4, 5, 6\}$, shown below.*



Which of the following is NOT a topological ordering?

A. *1 2 3 4 5 6*

B. *1 3 2 4 5 6*

C. *1 3 2 4 6 5*

D. *3 2 4 1 6 5*

GATEPYQ 226 (GATE IT 2006) *Consider the depth-first-search of an undirected graph with 3 vertices P, Q, and R. Let discovery time d(u) represent the time instant when the vertex u is first visited, and finish time f(u) represent the time instant when the vertex u is last visited. Given that*

$d(P) = 5$ *units* $f(P) = 12$ *units*
$d(Q) = 6$ *units* $f(Q) = 10$ *units*
$d(R) = 14$ *units* $f(R) = 18$ *units*

Which one of the following statements is TRUE about the graph?

A. *There is only one connected component*

B. *There are two connected components, and P and R are connected*

C. *There are two connected components, and Q and R are connected*

D. There are two connected components, and P and Q are connected

GATEPYQ 227 (GATE CSE 2004) *Level order traversal of a rooted tree can be done by starting from the root and performing:*

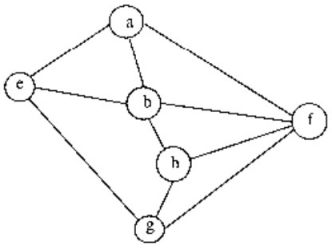
A. preorder traversal

B. in-order traversal

C. depth first search

D. breadth first search

GATEPYQ 228 (GATE CSE 2003) *Consider the following graph*



Among the following sequences

I. a b e g h f

II. a b f e h g

III. a b f h g e

IV. a f g h b e

Which are depth first traversals of the above graph?

A. I, II and IV only

B. I and IV only

C. II, III and IV only

D. I, III and IV only

GATEPYQ 229 (GATE CSE 2001) *Consider the undirected unweighted graph G. Let a breadth-first traversal of G be done starting from a node r. Let $d(r,u)$ and $d(r,v)$ be the lengths of the shortest paths from r to u and v respectively in G. If u is visited before v during the breadth-first traversal, which of the following statements is correct?*

A. $d(r,u) < d(r,v)$

B. $d(r,u) \leq d(r,v)$

C. $d(r,u) > d(r,v)$

D. None of the above

GATEPYQ 230 (GATE CSE 2000) *Let G be an undirected graph. Consider a depth-first traversal of G, and let T be the resulting depth-first search tree. Let u be a vertex in G and let v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following statement is always true?*

A. (u,v) must be an edge in G, and u is a descendant of v in T

B. (u,v) must be an edge in G, and v is a descendant of u in T

C. If (u,v) is not an edge in G then u is a leaf in T

D. If (u,v) is not an edge in G then u and v must have the same parent in T

GATEPYQ 231 (GATE CSE 2000) *The most appropriate matching for the following pairs is:*

X: depth first search	*1: heap*
Y: breadth first search	*2: queue*
Z: sorting	*3: stack*

A. X - 1, Y - 2, Z - 3

B. X - 3, Y - 1, Z - 2

C. X - 3, Y - 2, Z - 1

D. X - 2, Y - 3, Z - 1

gatexam.in

8.6 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
66	Breadth First Search & Properties	https://www.youtube.com/watch?v=JoXSu83SIM8	
67	Depth First Search & Properties	https://www.youtube.com/watch?v=qFk1vXXJ1xU	
68	Topological Sort (DFS Method & Kahn's Algorithm)	https://www.youtube.com/watch?v=dtbba0apE2Y	
69	GATE PYQs on DFS, BFS & Topological Sort	https://www.youtube.com/watch?v=yWjl9meVE3k	
70	Practice Problems on DFS, BFS & Topological Sort	https://www.youtube.com/watch?v=Rm3ETHxQ6Po	

Chapter 9

Shortest Path Algorithms

Concept: The Shortest Path Problem

Given a weighted graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, the shortest path problem asks: what is the minimum total weight path from one vertex to another (or to all other vertices)? The weight of a path is the sum of the weights of its edges. We study three major algorithms, each suited to different situations depending on whether weights can be negative, and whether we want one source or all pairs.
Think of this as planning a road trip. Each road (edge) has a travel cost (weight), like distance, time, or toll. We want to find the cheapest route from our starting city to a destination, or sometimes to every other city at once.

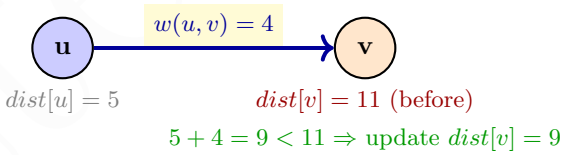
Concept: Relaxation — The Core Operation

All shortest path algorithms are built around a single basic operation called **relaxation**. For an edge (u, v) with weight $w(u, v)$, relaxing this edge means: "Can we improve our current best known distance to v by going through u?"

$$\text{if } dist[u] + w(u,v) < dist[v]: \quad dist[v] \leftarrow dist[u] + w(u,v), \quad parent[v] \leftarrow u$$

Every algorithm in this chapter (Dijkstra, Bellman-Ford, Floyd-Warshall) is just a different strategy for deciding **which edges to relax, and in what order**.

Relaxation: Trying to Improve $dist[v]$ via u



9.1 Single-Source: Dijkstra's Algorithm

Concept: Dijkstra's Algorithm — The Idea

Dijkstra's algorithm finds the shortest path from a single source vertex s to **every** other vertex in the graph, but it requires all edge weights to be **non-negative**. The core strategy is **greedy**: at every step, we pick the **unvisited** vertex with the **smallest** known distance, mark it as finalized, and relax all of its outgoing edges.
Think of it like water spreading outward from a source, but unlike BFS (which spreads in equal steps), Dijkstra always expands to the "nearest" unexplored point first, where "nearest" accounts for edge weights.

Example 293: Analogy: Dijkstra as a Careful Explorer

Imagine you are standing in a city, and you want to find the cheapest taxi fare to every other location. You always first finalize the fare to whichever place is currently cheapest to reach (among places not yet finalized), because once a place is the cheapest remaining option, there is no way a longer, more expensive route could ever beat it

(since all fares are non-negative, going through a costlier intermediate stop can never reduce the total cost).

Theorem: Why the Greedy Choice is Safe (Correctness Intuition)

At each step, Dijkstra picks the unvisited vertex u with minimum tentative distance $dist[u]$, and declares this distance **final**. This is correct **only because all weights are non-negative**. If u has the smallest tentative distance among all unvisited vertices, then any other path to u through a vertex that is not yet finalized would have to pass through an edge with non-negative weight, making that alternate path's length **at least** as long as $dist[u]$ already is. So no shorter path to u can possibly be discovered later.

Algorithm: Dijkstra(G, w, s)

```
1:  for each vertex v ∈ V do
2:      dist[v] ← ∞
3:      parent[v] ← NIL
4:  end for
5:  dist[s] ← 0
6:  S ← ∅                                          ▷ set of finalized vertices
7:  Q ← V                                          ▷ min-priority queue keyed by dist
8:  while Q ≠ ∅ do
9:      u ← Extract-Min(Q)                         ▷ unvisited vertex with smallest dist
10:     S ← S ∪ {u}
11:     for each vertex v adjacent to u do
12:         if dist[u] + w(u, v) < dist[v] then    ▷ relax
13:             dist[v] ← dist[u] + w(u, v)
14:             parent[v] ← u
15:             Decrease-Key(Q, v, dist[v])
16:         end if
17:     end for
18: end while
```

Concept: Time Complexity of Dijkstra

The running time depends entirely on how the priority queue is implemented:

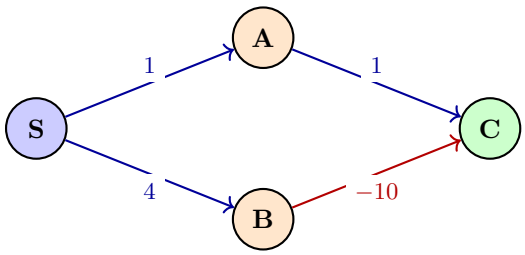
- **Simple array:** Finding the minimum each time takes $O(V)$ (linear scan), done V times, giving $O(V^2)$ for Extract-Min calls. Each edge relaxation is $O(1)$, giving $O(E)$ total for relaxations. Total: $O(V^2+E) = O(V^2)$ (since $E \leq V^2$).
- **Binary min-heap:** Extract-Min takes $O(\log V)$, done V times: $O(V \log V)$. Decrease-Key takes $O(\log V)$, done at most $O(E)$ times: $O(E \log V)$. Total: $O((V + E) \log V)$.
- **Fibonacci heap** (advanced): Extract-Min is $O(\log V)$ amortized, Decrease-Key is $O(1)$ amortized, giving $O(E + V \log V)$, the best known bound.

For **dense** graphs ($E \approx V^2$), the simple array is actually competitive or better. For **sparse** graphs ($E \approx V$), the heap-based version is much faster.

Concept: Why Dijkstra Fails on Negative Weights

Dijkstra's greedy strategy relies on the assumption that once a vertex is finalized (marked visited with its minimum distance locked in), no future relaxation can ever improve it. This assumption **breaks** if negative edge weights exist, because a path through a vertex we haven't visited yet (which currently looks "farther") could later take a very negative edge and become shorter overall than what we already finalized.

Why Dijkstra Fails with Negative Weights: Counter-Example



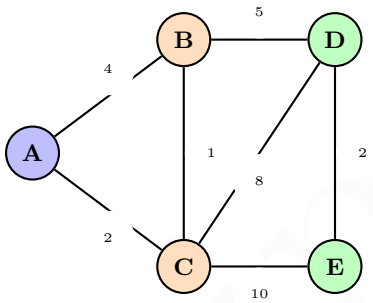
Dijkstra finalizes A first (dist=1), then finalizes C via A (dist=2), **before even considering B.**

But real shortest path S→B→C has cost $4 + (-10) = -6$, much better than 2!

9.1.1 Dijkstra: Step-by-Step Trace (Standard Example)

Example 294: Trace: Dijkstra on a Simple Graph, Source = A

Graph: vertices $\{A, B, C, D, E\}$, edges with weights: A–B:4, A–C:2, B–C:1, B–D:5, C–D:8, C–E:10, D–E:2.



Example 295: Dijkstra Distance Table Trace

Step	Extract-Min	$dist[A]$	$dist[B]$	$dist[C]$	$dist[D]$	$dist[E]$
Init	—	0	∞	∞	∞	∞
1	A (dist 0)	0	**4**	**2**	∞	∞
2	C (dist 2)	0	**3**	2	**10**	**12**
3	B (dist 3)	0	3	2	**8**	12
4	D (dist 8)	0	3	2	8	**10**
5	E (dist 10)	0	3	2	8	10

Explanation of each step:

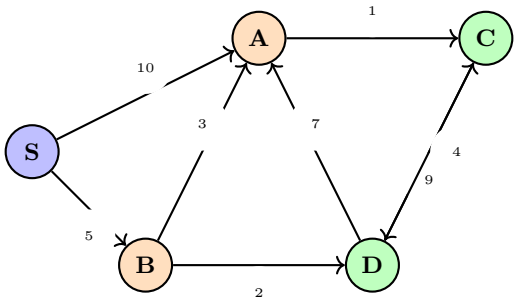
- **Step 1:** Extract A (dist 0). Relax $A \to B$: $0 + 4 = 4 < \infty$, set $dist[B] = 4$. Relax $A \to C$: $0 + 2 = 2 < \infty$, set $dist[C] = 2$.
- **Step 2:** Smallest unvisited is C (dist 2). Relax $C \to B$: $2 + 1 = 3 < 4$, improve $dist[B] = 3$. Relax $C \to D$: $2 + 8 = 10$, set $dist[D] = 10$. Relax $C \to E$: $2 + 10 = 12$, set $dist[E] = 12$.
- **Step 3:** Smallest unvisited is B (dist 3). Relax $B \to D$: $3 + 5 = 8 < 10$, improve $dist[D] = 8$.
- **Step 4:** Smallest unvisited is D (dist 8). Relax $D \to E$: $8 + 2 = 10 < 12$, improve $dist[E] = 10$.
- **Step 5:** Smallest unvisited is E (dist 10). No outgoing edges to relax further.

Final shortest distances from A: $B = 3, C = 2, D = 8, E = 10$.

9.1.2 Dijkstra: Complex Trace (Directed Graph with Reordering)

Example 296: Trace: Dijkstra on a Directed Graph, Source = S (GATE-style)

Directed graph: $S \to A(10)$, $S \to B(5)$, $B \to A(3)$, $B \to D(2)$, $A \to C(1)$, $D \to A(7)$, $D \to C(9)$, $C \to D(4)$. This graph tests whether the relaxation order causes re-evaluation.



Example 297: Full Trace

Step	Extract-Min	$dist[S]$	$dist[A]$	$dist[B]$	$dist[D]$	$dist[C]$
Init	—	0	∞	∞	∞	∞
1	S (0)	0	**10**	**5**	∞	∞
2	B (5)	0	**8**	5	**7**	∞
3	D (7)	0	**8**	5	7	**16**
4	A (8)	0	8	5	7	**9**
5	C (9)	0	8	5	7	9

Key observation: Notice that $dist[A]$ was first set to 10 (via $S \to A$), then improved to 8 (via $S \to B \to A$), demonstrating that a vertex's tentative distance can be relaxed **multiple times** before it is finally extracted. Once A is extracted (finalized) at distance 8, it will never change again, this is the core greedy guarantee.

Concept: Dijkstra — Assumptions, Properties, Advantages, Limitations

- **Assumption:** All edge weights must be non-negative. This is non-negotiable; the algorithm's correctness proof depends on it.
- **Property:** Once a vertex is extracted from the priority queue (finalized), its distance value is guaranteed optimal and never changes again.
- **Property:** Dijkstra works for both directed and undirected graphs, as long as weights are non-negative.
- **Advantage:** Much faster than Bellman-Ford for non-negative weights, especially with a good priority queue: $O((V+E)\log V)$ vs $O(VE)$.
- **Advantage:** Naturally produces a shortest-path tree (via the parent pointers), giving paths to *every* reachable vertex from the source in one run.
- **Limitation:** Cannot handle negative edge weights at all, even without negative cycles; it can give incorrect (too large) results.
- **Limitation:** Does not directly extend to all-pairs shortest paths; running it from every vertex gives $O(V(V+E)\log V)$, which is often worse than Floyd-Warshall for dense graphs.

Python implementation: dijkstra.py

```
import heapq

def dijkstra(graph, n, source):
```

```
    """
    graph: dict of u -> list of (v, weight)
    Returns dist[] and parent[] arrays.
    """
    dist   = [float('inf')] * n
    parent = [-1] * n
    dist[source] = 0

    pq = [(0, source)]           # (distance, vertex)
    visited = [False] * n

    while pq:
        d, u = heapq.heappop(pq)
        if visited[u]:
            continue             # stale entry, skip
        visited[u] = True

        for v, w in graph[u]:
            if not visited[v] and dist[u] + w < dist[v]:
                dist[v]   = dist[u] + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))

    return dist, parent


# --- Driver Code ---
# Graph: S=0, A=1, B=2, D=3, C=4
graph = {
    0: [(1, 10), (2, 5)],
    1: [(4, 1)],
    2: [(1, 3), (3, 2)],
    3: [(1, 7), (4, 9)],
    4: [(3, 4)],
}

dist, parent = dijkstra(graph, 5, 0)
print("Distances from S:", dist)      # [0, 8, 5, 7, 9]
print("Parents:", parent)
```

9.2 Single-Source: Bellman-Ford Algorithm

Concept: Bellman-Ford — The Idea

Bellman-Ford also finds shortest paths from a single source, but unlike Dijkstra, it works even when edge weights are **negative** (as long as there is no negative weight cycle reachable from the source). Instead of greedily picking the closest vertex, Bellman-Ford takes a simpler, more brute-force approach: relax **every single edge**, and repeat this entire process $V - 1$ times.

Theorem: Why $V - 1$ Iterations Suffice

A shortest path between any two vertices in a graph with V vertices can have at most $V - 1$ edges (since a simple path visits at most V vertices, hence at most $V - 1$ edges; a path with a cycle is never shorter than the same path with the cycle removed, as long as there's no negative cycle).
Each full pass over all edges guarantees that the shortest path using at most k edges is correctly computed after the k-th pass. So after $V - 1$ passes, we are guaranteed to have found the shortest path using at most $V - 1$ edges, which covers every possible simple shortest path in the graph.

Algorithm: Bellman-Ford(G, w, s)

```
for each vertex v ∈ V do
    dist[v] ← ∞
    parent[v] ← NIL
end for
dist[s] ← 0
for i = 1 to |V| − 1 do                          ▷ repeat V-1 times
    for each edge (u, v) ∈ E with weight w(u, v) do
        if dist[u] + w(u, v) < dist[v] then
            dist[v] ← dist[u] + w(u, v)
            parent[v] ← u
        end if
    end for
end for                                          ▷ Check for negative weight cycles
for each edge (u, v) ∈ E with weight w(u, v) do
    if dist[u] + w(u, v) < dist[v] then
        return "Negative weight cycle detected"
    end if
end for
return dist[]
```

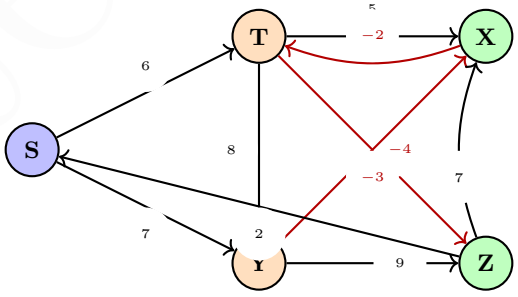
Concept: Negative Weight Cycle Detection

After the $V-1$ main iterations, we do **one more** pass over all edges. If any edge can still be relaxed (i.e., $dist[u] + w(u,v) < dist[v]$), this means the shortest path estimate is *still* improving beyond what $V-1$ edges should allow. This can only happen if there is a **negative weight cycle** reachable from the source, since such a cycle allows us to keep looping around it, making the "shortest path" infinitely small (undefined).

9.2.1 Bellman-Ford: Step-by-Step Trace

Example 298: Trace: Bellman-Ford on a Graph with Negative Edges, Source = S

Directed graph: $S \to A(6)$, $S \to C(7)$, $A \to C(8)$, $A \to D(5)$, $A \to B(-4)$, $C \to D(-3)$, $C \to E(9)$, $D \to B(-1)$, $B \to D(7)$ wait — let's use the classic CLRS example: $S \to T(6), S \to Y(7), T \to X(5), T \to Y(8), T \to Z(-4), X \to T(-2), Y \to X(-3), Y \to Z(9), Z \to X(7), Z \to S(2)$.



Example 299: Bellman-Ford Trace: Relaxing All Edges Each Pass

Processing order of edges: $(S,T), (S,Y), (T,X), (T,Y), (T,Z), (X,T), (Y,X), (Y,Z), (Z,X), (Z,S)$.

Pass	$dist[S]$	$dist[T]$	$dist[X]$	$dist[Y]$	$dist[Z]$
Init	0	∞	∞	∞	∞
1	0	6	11	7	2
2	0	2	4	7	-2
3	0	2	4	7	-2
4	0	2	4	7	-2

Pass 1: (S,T): $0+6=6 \to dist[T]=6$. (S,Y): $0+7=7 \to dist[Y]=7$. (T,X): $6+5=11 \to dist[X]=11$. (T,Y): $6+8=14$, no improvement. (T,Z): $6-4=2 \to dist[Z]=2$. (X,T): $11-2=9$, no improvement. (Y,X): $7-3=4<11 \to dist[X]=4$. (Y,Z): $7+9=16$, no improvement. (Z,X): $2+7=9$, no improvement. (Z,S): $2+2=4$, no improvement.
Pass 2: (T,Z): now $dist[T]=6$ still, so no change yet in this pass's early edges. Key change: (X,T): $4-2=2<6 \to dist[T]=2$. Later edges then use updated $dist[T]=2$: (T,Z) re-examined only in pass 3 since edges are processed once per pass in fixed order, so by pass 2 end, $dist[T]=2$ propagates further in Pass 3: (T,Z): $2-4=-2<2 \to dist[Z]=-2$.
Pass 3, 4: No further changes; values have converged.
Final shortest distances from S: $T=2, X=4, Y=7, Z=-2$.
Verification pass (negative cycle check): Re-examine all edges once more; no edge can be relaxed further, confirming **no negative weight cycle** exists.

Concept: Bellman-Ford — Assumptions, Properties, Advantages, Limitations

- **Assumption:** The graph can have negative edge weights, but must not have a negative weight cycle reachable from the source (or the algorithm correctly detects this instead of producing wrong output).
- **Property:** After k passes, $dist[v]$ correctly equals the shortest path to v using at most k edges.
- **Property:** The order in which edges are processed within a single pass does not affect the final correctness, though it may affect how fast convergence happens (sometimes fewer than $V-1$ passes are needed in practice).
- **Advantage:** Handles negative weights, unlike Dijkstra.
- **Advantage:** Can explicitly detect negative weight cycles, a very useful diagnostic capability (e.g., detecting arbitrage opportunities in currency exchange graphs).
- **Advantage:** Simple to implement; works with just an edge list, no priority queue needed.
- **Limitation:** Much slower than Dijkstra for the same non-negative-weight graph: $O(VE)$ vs $O((V+E)\log V)$.
- **Limitation:** If a negative cycle is reachable from the source, the notion of "shortest path" to vertices affected by that cycle is undefined (distances can be made arbitrarily small by looping).

Python implementation: bellman-ford.py

```
def bellman_ford(edges, n, source):
    """
    edges: list of (u, v, weight) tuples
    Returns dist[], parent[], and whether a negative cycle exists.
    """
    dist   = [float('inf')] * n
    parent = [-1] * n
    dist[source] = 0

    # Relax all edges V-1 times
    for _ in range(n - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v]   = dist[u] + w
                parent[v] = u
                updated = True
        if not updated:           # early exit: no change this pass
            break

    # One more pass: detect negative weight cycle
    has_negative_cycle = False
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            has_negative_cycle = True
            break

    return dist, parent, has_negative_cycle
```

```
# --- Driver Code ---
# S=0, T=1, X=2, Y=3, Z=4
edges = [
    (0,1,6), (0,3,7), (1,2,5), (1,3,8), (1,4,-4),
    (2,1,-2), (3,2,-3), (3,4,9), (4,2,7), (4,0,2),
]

dist, parent, neg_cycle = bellman_ford(edges, 5, 0)
print("Distances:", dist)              # [0, 2, 4, 7, -2]
print("Negative cycle?", neg_cycle)    # False
```

9.3 All-Pairs: Floyd-Warshall Algorithm

Concept: Floyd-Warshall — The Idea

Floyd-Warshall solves the **all-pairs shortest path** problem: finding the shortest distance between **every** pair of vertices (i, j), all at once. It uses **dynamic programming**, based on a simple but powerful idea: for every pair (i, j), we consider whether the shortest path should pass through an intermediate vertex k, trying every possible k, one at a time.

Concept: The DP Formulation

Let $d^{(k)}[i][j]$ denote the shortest path from i to j **using only vertices** $\{1, 2, \ldots, k\}$ **as intermediate vertices** (the endpoints i and j themselves are always allowed). The recurrence is:

$$d^{(k)}[i][j] = \min\left(d^{(k-1)}[i][j],\ d^{(k-1)}[i][k] + d^{(k-1)}[k][j]\right)$$

In words: the shortest path either does **not** use vertex k at all (so it's the same as before), or it **does** pass through k (so it splits into a path from i to k, plus a path from k to j, both using only vertices up to $k-1$). We take the smaller of these two options. The base case is $d^{(0)}[i][j] = w(i, j)$ (direct edge weight, or ∞ if no edge, and 0 if $i = j$).
Since each $d^{(k)}$ only depends on $d^{(k-1)}$, we can use a single 2D array and update it in-place as k increases from 1 to V.

Algorithm: Floyd-Warshall(W)

```
d ← W                                   ▷ initialize with direct edge weights, d[i][i] = 0, ∞ if no edge
for k = 1 to |V| do
    for i = 1 to |V| do
        for j = 1 to |V| do
            if d[i][k] + d[k][j] < d[i][j] then
                d[i][j] ← d[i][k] + d[k][j]
            end if
        end for
    end for
end for
return d
```

Concept: Time and Space Complexity

Three nested loops, each running V times, gives $\Theta(V^3)$ time. We store a $V \times V$ matrix, giving $\Theta(V^2)$ space. This is efficient for **dense** graphs where we want all-pairs distances; running Dijkstra from every vertex would take $O(V \cdot (V + E) \log V)$, which for dense graphs ($E \approx V^2$) becomes $O(V^3 \log V)$, actually worse than Floyd-Warshall's clean $O(V^3)$.

9.3.1 Floyd-Warshall: Step-by-Step Matrix Trace

Example 300: Trace: Floyd-Warshall on a 4×4 Graph

Vertices $\{1, 2, 3, 4\}$, directed edges: $1 \to 2(3), 1 \to 3(8), 1 \to 4(-4), 2 \to 4(1), 2 \to 3(7), 3 \to 1(2), 4 \to 3(-5), 4 \to 1(6)$. This matches the classic CLRS Floyd-Warshall example.

Example 301: Initial Matrix $d^{(0)}$

$$d^{(0)} = \begin{pmatrix} 0 & 3 & 8 & -4 \\ \infty & 0 & 7 & 1 \\ 2 & \infty & 0 & \infty \\ 6 & \infty & -5 & 0 \end{pmatrix}$$

Example 302: After $k = 1$ (allowing vertex 1 as intermediate)

Check if any $d[i][1] + d[1][j] < d[i][j]$. For example, $d[3][2]$: direct is ∞, via vertex 1: $d[3][1] + d[1][2] = 2 + 3 = 5$. Update $d[3][2] = 5$. Similarly $d[3][4]$: $d[3][1] + d[1][4] = 2 + (-4) = -2 < \infty$, update.

$$d^{(1)} = \begin{pmatrix} 0 & 3 & 8 & -4 \\ \infty & 0 & 7 & 1 \\ 2 & 5 & 0 & -2 \\ 6 & 9 & -5 & 0 \end{pmatrix}$$

(Row 4, column 2: $d[4][1] + d[1][2] = 6 + 3 = 9 < \infty$, so updated.)

Example 303: After $k = 2$ (allowing vertex 2 as intermediate)

Check $d[i][2] + d[2][j]$. For instance $d[1][3]$: direct is 8, via vertex 2: $d[1][2] + d[2][3] = 3 + 7 = 10$, not better. $d[3][3]$ via 2: $5 + \infty$, no change. $d[1][4]$ via 2: $3 + 1 = 4$, not better than -4. No improvements occur at this step for most entries except checking carefully all combinations; the matrix stays effectively the same at this stage for this particular graph.

$$d^{(2)} = \begin{pmatrix} 0 & 3 & 8 & -4 \\ \infty & 0 & 7 & 1 \\ 2 & 5 & 0 & -2 \\ 6 & 9 & -5 & 0 \end{pmatrix}$$

Example 304: After $k = 3$ (allowing vertex 3 as intermediate)

Check $d[i][3] + d[3][j]$. $d[2][1]$: direct ∞, via 3: $d[2][3] + d[3][1] = 7 + 2 = 9$. Update $d[2][1] = 9$. $d[4][2]$: via 3: $d[4][3] + d[3][2] = -5 + 5 = 0 < 9$, update $d[4][2] = 0$. $d[2][4]$: via 3: $7 + (-2) = 5$, not better than 1.

$$d^{(3)} = \begin{pmatrix} 0 & 3 & 8 & -4 \\ 9 & 0 & 7 & 1 \\ 2 & 5 & 0 & -2 \\ 6 & 0 & -5 & 0 \end{pmatrix}$$

Example 305: After $k = 4$ (allowing vertex 4 as intermediate) — Final Matrix

Check $d[i][4] + d[4][j]$. $d[1][3]$: via 4: $d[1][4] + d[4][3] = -4 + (-5) = -9 < 8$. Update $d[1][3] = -9$. $d[2][1]$: via 4: $1 + 6 = 7 < 9$. Update $d[2][1] = 7$. $d[2][3]$: via 4: $1 + (-5) = -4 < 7$. Update $d[2][3] = -4$. $d[3][2]$: via 4:

$-2+0=-2<5$. Update $d[3][2]=-2$.

$$d^{(4)}=\begin{pmatrix}0 & 3 & -9 & -4\\ 7 & 0 & -4 & 1\\ 2 & -2 & 0 & -2\\ 6 & 0 & -5 & 0\end{pmatrix}$$

This is the final all-pairs shortest distance matrix. Reading row i, column j gives the shortest distance from i to j.

Concept: Detecting Negative Weight Cycles via Floyd-Warshall

After running the algorithm to completion, check the **diagonal** entries $d[i][i]$. In a graph with no negative weight cycles, every $d[i][i]$ should remain 0 (the trivial zero-length path from i to itself). If any $d[i][i]<0$ after all iterations, it means there is a cycle through i with negative total weight, since the "shortest path" from i back to itself has become negative.

Concept: Floyd-Warshall — Assumptions, Properties, Advantages, Limitations

- **Assumption:** Works correctly with negative edge weights, but like Bellman-Ford, requires no negative weight cycles for the result to be meaningful (otherwise, detect it via negative diagonal entries).
- **Property:** The order of considering intermediate vertices $k=1,2,\ldots,V$ matters for correctness; we must process them in a complete, increasing sequence (any fixed full ordering works, not necessarily by vertex label).
- **Property:** Can be used on both directed and undirected graphs (undirected simply means $w(i,j)=w(j,i)$).
- **Advantage:** Gives **all pairs** shortest paths in a single clean run, with simple nested-loop code, no complex data structures.
- **Advantage:** Can detect negative weight cycles anywhere in the graph (not just those reachable from one source, unlike Bellman-Ford which only checks from a chosen source).
- **Advantage:** Simple to extend for computing **transitive closure** (reachability) by replacing min/+ with OR/AND operations.
- **Limitation:** $O(V^3)$ time can be too slow for very large sparse graphs; running Dijkstra V times (with a heap) is often faster when $E \ll V^2$.
- **Limitation:** $O(V^2)$ space can also be prohibitive for very large V.

Python implementation: floyd-warshall.py

```
def floyd_warshall(W, n):
    """
    W: n x n matrix, W[i][j] = weight of edge i->j,
       float('inf') if no edge, 0 if i == j
    Returns the all-pairs shortest distance matrix.
    """
    d = [row[:] for row in W]    # deep copy

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]

    # Check for negative weight cycles
    for i in range(n):
        if d[i][i] < 0:
            print(f"Negative weight cycle detected through vertex {i}")

    return d
```

```
# --- Driver Code ---
INF = float('inf')
# Vertices: 1,2,3,4 -> indices 0,1,2,3
W = [
    [0,   3,   8,  -4],
    [INF, 0,   7,   1],
    [2,   INF, 0,   INF],
    [6,   INF, -5,  0],
]

result = floyd_warshall(W, 4)
for row in result:
    print(row)

# Output:
# [0, 3, -9, -4]
# [7, 0, -4, 1]
# [2, -2, 0, -2]
# [6, 0, -5, 0]
```

9.4 Choosing the Right Algorithm

Shortest Path Algorithm Selection Guide

Scenario	Algorithm	Time Complexity	Key Constraint
Unweighted graph	**BFS**	$O(V+E)$	All edges weight 1
Single-source, non-negative	**Dijkstra**	$O((V+E)\log V)$	$w(e) \geq 0$ for all e
Single-source, negative OK	**Bellman-Ford**	$O(VE)$	No negative cycle (or detect)
All-pairs	**Floyd-Warshall**	$O(V^3)$	No negative cycle (or detect)
All-pairs, sparse, non-neg	**Dijkstra** $\times V$	$O(V(V+E)\log V)$	Faster than F-W if $E \ll V^2$

Concept: Key Decision Questions

1. **Are all weights 1 (unweighted)?** Use BFS. Simplest and fastest, $O(V+E)$.
2. **Single source needed, and are all weights non-negative?** Use Dijkstra. It is the fastest correct algorithm for this case.
3. **Single source needed, but negative weights possible?** Use Bellman-Ford. Slower, but correctly handles negatives and can detect negative cycles.
4. **Need distances between every pair of vertices?** Use Floyd-Warshall if the graph is dense or V is small-to-moderate. If the graph is sparse and weights are non-negative, running Dijkstra from each vertex can be faster.

9.5 Practice Problems

Problem 293 ***[MCQ]*** *Consider running Dijkstra's algorithm on the following directed graph with source S: edges $S \to A(4)$, $S \to B(1)$, $B \to A(2)$, $B \to C(5)$, $A \to C(1)$. What is the order in which vertices are extracted (finalized) from the priority queue?*

(A) S, A, B, C

(B) S, B, A, C

(C) S, B, C, A

(D) S, A, C, B

Problem 294 ***[MSQ]*** *Which of the following statements about Dijkstra's algorithm are TRUE?*

(A) Once a vertex is extracted from the priority queue, its shortest distance value is finalized and never changes again.

(B) Dijkstra's algorithm can be used directly on graphs with negative edge weights if there is no negative cycle.

(C) Using a simple array instead of a heap gives $O(V^2)$ time, which can be better than the heap-based version for dense graphs.

(D) Dijkstra's algorithm, run from every vertex, is always asymptotically faster than Floyd-Warshall for all graph densities.

Problem 295 ***[MSQ]*** *Consider running Bellman-Ford on a directed graph. Which of the following are TRUE?*

(A) If after $V-1$ passes, the V-th pass still relaxes some edge, a negative weight cycle reachable from the source exists.

(B) Bellman-Ford correctly computes shortest paths even when some edges have negative weight, as long as there's no negative cycle reachable from the source.

(C) The time complexity of Bellman-Ford is always strictly worse than Dijkstra's, for any graph.

(D) Bellman-Ford can be used to detect negative weight cycles anywhere in the graph, regardless of the chosen source vertex.

Problem 296 ***[NAT]*** *Bellman-Ford is run on a graph with $V = 5$ vertices and $E = 8$ edges. The algorithm performs relaxation checks on every edge during every pass, for $V-1$ passes, plus one final verification pass. The total number of edge relaxation attempts (checks) performed across all passes (including the verification pass) is _____.*

Problem 297 ***[MCQ]*** *Consider a directed graph with a cycle $A \to B \to C \to A$ with edge weights $3, -8, 2$ respectively. If this cycle is reachable from the source vertex in a Bellman-Ford run, what happens?*

(A) Bellman-Ford converges normally since the cycle weight sum is $3 + (-8) + 2 = -3$, which doesn't matter.

(B) Bellman-Ford detects this as a negative weight cycle, since the total cycle weight -3 is negative.

(C) Bellman-Ford ignores cycles entirely and only considers simple paths.

(D) Bellman-Ford incorrectly reports this as a positive weight cycle.

Problem 298 ***[MCQ]*** *Floyd-Warshall is run on a graph with $V = 4$ vertices. After processing $k = 1$ and $k = 2$ (but before processing $k = 3$), the matrix entry $d[3][4]$ represents:*

(A) The shortest path from vertex 3 to vertex 4 using any vertices as intermediates.

(B) The shortest path from vertex 3 to vertex 4 using only vertices $\{1, 2\}$ as intermediates.

(C) The direct edge weight only, with no intermediates considered.

(D) The shortest path from vertex 3 to vertex 4 using only vertex $\{3\}$ as an intermediate.

Problem 299 ***[NAT]*** *Floyd-Warshall is run on a 4×4 weighted directed graph. Initial matrix (vertices 1-4, ∞ denotes no direct edge):*

$$W = \begin{pmatrix} 0 & 5 & \infty & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

After running Floyd-Warshall to completion, the shortest distance from vertex 1 to vertex 4 is _____.

Problem 300 ***[MCQ]*** *A graph's Floyd-Warshall algorithm is run to completion, and the resulting matrix has $d[2][2] = -5$. What does this indicate?*

(A) A computational error, since diagonal entries must always be 0.

(B) There is a cycle passing through vertex 2 with total negative weight.

(C) Vertex 2 is unreachable from itself.

(D) The graph must be undirected.

Problem 301 ***[MSQ]*** *Which of the following are TRUE about the Floyd-Warshall algorithm?*

(A) It correctly handles negative edge weights, as long as there are no negative weight cycles.

(B) Its time complexity is $\Theta(V^3)$ regardless of the number of edges E.

(C) It requires the intermediate vertices to be considered in the order $k = 1, 2, \ldots, V$, and this specific numeric order cannot be changed.

(D) It can be adapted to compute the transitive closure of a directed graph.

Problem 302 ***[MCQ]*** *For a sparse graph with $V = 1000$ vertices and $E = 2000$ edges, all with non-negative weights, which approach is most efficient for computing all-pairs shortest paths?*

(A) Floyd-Warshall

(B) Running Dijkstra from each vertex with a heap.

(C) Running Bellman-Ford from each vertex.

(D) Both A and C are equally efficient

Problem 303 ***[MCQ]*** *Consider the following claim: "If we run Dijkstra's algorithm on a graph where all edge weights are increased by the same positive constant c, the shortest path tree remains the same." Is this claim TRUE?*

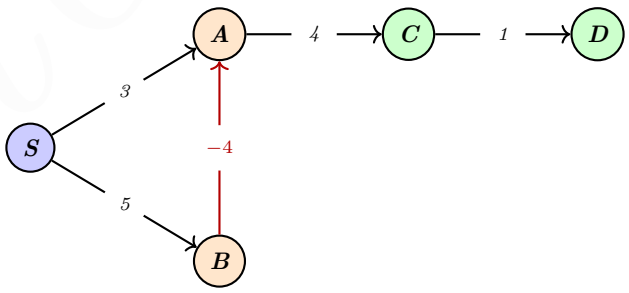
(A) True; adding a constant to every edge weight never changes which path is shortest, since every path's total increases by the same amount proportional to constant times number of edges.

(B) False in general, because paths with different numbers of edges get different total increases, potentially changing which path is shortest.

(C) True, but only for trees.

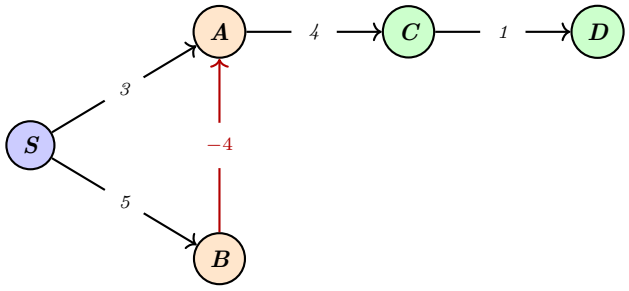
(D) False, because Dijkstra cannot handle any modification to edge weights at all.

Problem 304 ***[NAT]*** *Consider the directed graph below. It has one negative edge $B \to A$ and no cycle. A standard Dijkstra implementation, which never re-extracts a vertex once it has been finalized, is run from S.*

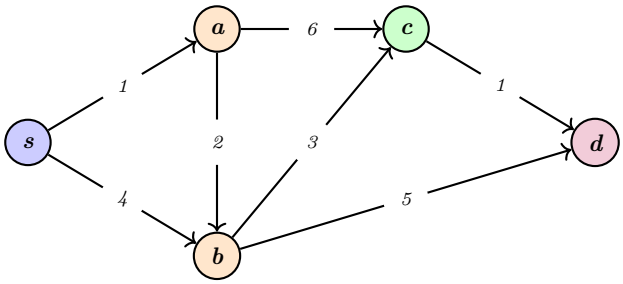


The value of $dist[D]$ computed by this Dijkstra run is _____.

Problem 305 ***[NAT]*** *For the graph given, the **true** shortest-path distance from S to D by Bellman-Ford Algorithm is* _____.



Problem 306 ***[MSQ]*** *For the graph above, the* ***removal*** *of which single edge (keeping all others) strictly increases the shortest distance from s to d?*



(A) $a \to b$

(B) $b \to d$

(C) $a \to c$

(D) $c \to d$

Problem 307 ***[MSQ]*** *Let G be a connected graph with positive edge weights, and T the shortest path tree from s computed by Dijkstra. For which transformations of the edge weights is T (as a set of shortest paths) guaranteed to remain valid?*

(A) Multiply every weight by 5.

(B) Add 5 to every weight.

(C) Replace every weight w by w^2.

(D) Multiply every weight by 0.5.

Problem 308 ***[MCQ]*** *All edges of a connected graph have the same weight* 5*. Which algorithm gives single-source shortest distances in the* ***best*** *time?*

(A) Bellman-Ford

(B) Floyd-Warshall

(C) BFS (then multiply distances by 5)

(D) Dijkstra with an array

Problem 309 ***[MSQ]*** *Which of the following statements about Dijkstra's algorithm on a connected graph with positive weights are TRUE?*

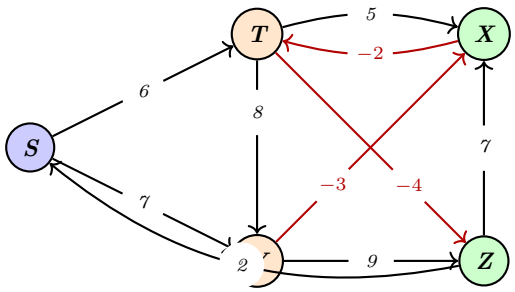
(A) The sequence of extracted distances $dist[u_1] \le dist[u_2] \le \cdots$ is non-decreasing.

(B) If all edge weights are distinct, the shortest path between two vertices is unique.

(C) The number of Decrease-Key operations is at most $|E|$.

(D) The shortest path tree is always a minimum spanning tree.

Problem 310 ***[NAT]*** *In the graph, edge (Z, S) has weight w (originally 2). All other weights stay the same. The smallest integer value of w for which the graph has* ***no negative weight cycle*** *is* _____.



Problem 311 ***[MSQ]*** *Which of the following statements about Bellman-Ford are TRUE?*

(A) After k passes, $dist[v]$ is at most the weight of the shortest path to v that uses at most k edges.

(B) The order of relaxing edges within a pass affects the number of passes needed, but not the final correct answer.

(C) A negative cycle that is not reachable from the source is detected by the V-th pass.

(D) Worst-case time is $\Theta(VE)$.

Problem 312 ***[NAT]*** *A directed graph has 3 vertices and edges $1 \to 2$ (weight 2), $2 \to 3$ (weight -4), $3 \to 1$ (weight 1). After Floyd-Warshall completes, the number of vertices i with $d[i][i] < 0$ is _____.*

Problem 313 ***[MCQ]*** *Single-source shortest paths are needed on a graph with $V = 10^5$ vertices, $E = 3 \times 10^5$ edges, all weights non-negative. The best choice is:*

(A) Floyd-Warshall

(B) Bellman-Ford

(C) Dijkstra with a binary heap

(D) Dijkstra with an array

gatexam.in

9.6 GATE PYQs

GATEPYQ 232 (GATE CSE 2026) *Let $G(V, E)$ be an undirected, edge-weighted graph with integer weights. The weight of a path is the sum of the weights of the edges in that path. The length of a path is the number of edges in that path.*
Let $s \in V$ be a vertex in G. For every $u \in V$ and for every $k \geq 0$, let $d_k(u)$ denote the weight of a shortest path (in terms of weight) from s to u of length at most k. If there is no path from s to u of length at most k, then $d_k(u) = \infty$.
Consider the statements:

S1: *For every $k \geq 0$ and $u \in V$, $d_{k+1}(u) \leq d_k(u)$.*

S2: *For every $(u, v) \in E$, if (u, v) is part of a shortest path (in terms of weight) from s to v, then for every $k \geq 0$, $d_k(u) \leq d_k(v)$.*

Which one of the following options is correct?

A. *Only S1 is true*

B. *Only S2 is true*

C. *Both S1 and S2 are true*

D. *Neither S1 nor S2 is true*

GATEPYQ 233 (GATE CSE 2025) *Let $G(V, E)$ be an undirected and unweighted graph with 100 vertices. Let $d(u, v)$ denote the number of edges in a shortest path between vertices u and v in V. Let the maximum value of $d(u, v), u, v \in V$ such that $u \neq v$, be 30. Let T be any breadth-first-search tree of G. Which ONE of the given options is CORRECT for every such graph G?*

A. *The height of T is exactly 15.*

B. *The height of T is exactly 30.*

C. *The height of T is at least 15.*

D. *The height of T is at least 30.*

GATEPYQ 234 (GATE CSE 2020) *Let $G = (V, E)$ be a directed, weighted graph with weight function $w : E \to \mathbb{R}$. For some function $f : V \to \mathbb{R}$, for each edge $(u, v) \in E$, define $w'(u, v)$ as $w(u, v) + f(u) - f(v)$.*
Which one of the options completes the following sentence so that it is TRUE?
"The shortest paths in G under w are shortest paths under w' too, _____."

A. *for every $f : V \to \mathbb{R}$*

B. *if and only if $\forall u \in V, f(u)$ is positive*

C. *if and only if $\forall u \in V, f(u)$ is negative*

D. *if and only if $f(u)$ is the distance from s to u in the graph obtained by adding a new vertex s to G and edges of zero weight from s to every vertex of G*

GATEPYQ 235 (GATE CSE 2016) *Consider the weighted undirected graph with 4 vertices, where the weight of edge $\{i, j\}$ is given by the entry W_{ij} in the matrix W.*

$$W = \begin{pmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{pmatrix}$$

The largest possible integer value of x, for which at least one shortest path between some pair of vertices will contain the edge with weight x is _____.

GATEPYQ 236 (GATE CSE 2016) *The Floyd-Warshall algorithm for all-pair shortest paths computation is based on*

A. *Greedy paradigm*

B. *Divide-and-Conquer paradigm*

C. *Dynamic Programming paradigm*

D. Neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm

GATEPYQ 237 (GATE CSE 2013) *What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of n vertices?*

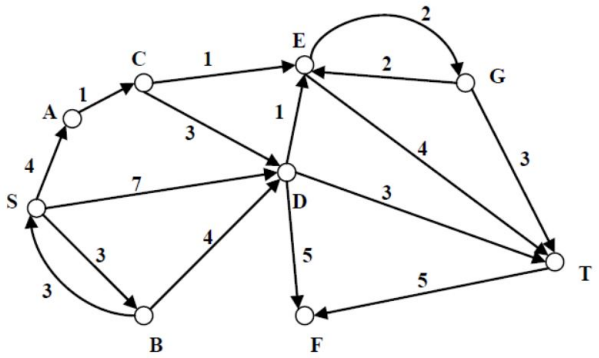
A. $\Theta(n^2)$

B. $\Theta(n^2 \log n)$

C. $\Theta(n^3)$

D. $\Theta(n^3 \log n)$

GATEPYQ 238 (GATE CSE 2012) *Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered.*



A. $S \to D \to T$

B. $S \to B \to D \to T$

C. $S \to A \to C \to D \to T$

D. $S \to A \to C \to E \to T$

GATEPYQ 239 (GATE CSE 2010) *Consider a complete undirected graph with vertex set $\{0, 1, 2, 3, 4\}$. Entry W_{ij} in the matrix W below is the weight of the edge $\{i, j\}$.*

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

What is the minimum possible weight of a path P from vertex 1 to vertex 2 in this graph such that P contains at most 3 edges?

A. 7

B. 8

C. 9

D. 10

GATEPYQ 240 (GATE CSE 2009) *Which of the following statement(s) is/are correct regarding Bellman-Ford shortest path algorithm?*

P: *Always finds a negative weighted cycle, if one exists.*

Q: *Finds whether any negative weighted cycle is reachable from the source.*

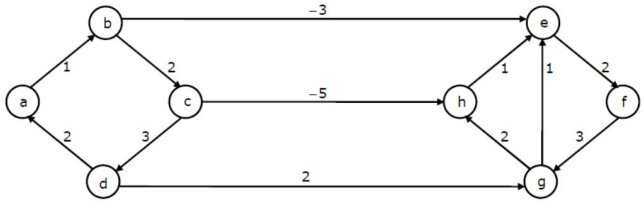
A. P only

B. Q only

C. *both P and Q*

D. *Neither P nor Q*

GATEPYQ 241 (GATE CSE 2008) *Dijkstra's single source shortest path algorithm when run from vertex a in the above graph, computes the correct shortest path distance to*



A. *only vertex a*

B. *only vertices a, e, f, g, h*

C. *only vertices a, b, c, d*

D. *all the vertices*

GATEPYQ 242 (GATE IT 2007) *Consider a weighted, undirected graph with positive edge weights and let (u, v) be an edge in the graph. It is known that the shortest path from the source vertex s to u has weight 53 and the shortest path from s to v has weight 65. Which one of the following statements is always TRUE?*

A. $Weight(u, v) \leq 12$

B. $Weight(u, v) = 12$

C. $Weight(u, v) \geq 12$

D. $Weight(u, v) > 12$

GATEPYQ 243 (GATE CSE 2006) *To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is*

A. *Queue*

B. *Stack*

C. *Heap*

D. *B-Tree*

GATEPYQ 244 (GATE CSE 2005) *Let $G(V, E)$ be an undirected graph with positive edge weights. Dijkstra's single source shortest path algorithm can be implemented using the binary heap data structure with time complexity:*

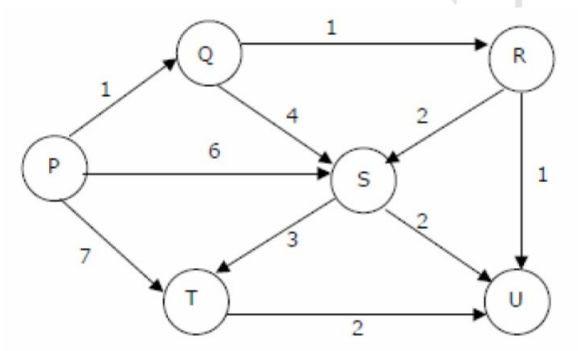
A. $O(|V|^2)$

B. $O(|E| + |V| \log |V|)$

C. $O(|V| \log |V|)$

D. $O((|E| + |V|) \log |V|)$

GATEPYQ 245 (GATE CSE 2004) *Suppose we run Dijkstra's single source shortest-path algorithm on the following edge-weighted directed graph with vertex P as the source.*



In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?

A. P, Q, R, S, T, U

B. P, Q, R, U, S, T

C. P, Q, R, U, T, S

D. P, Q, T, R, U, S

gatexam.in

9.7 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
71	Dijkstra's Algorithm: Single Source Shortest Path	https://www.youtube.com/watch?v=L9onxFQOoi8	
72	Bellman-Ford Algorithm	https://www.youtube.com/watch?v=UaregbEVGb8	
73	Floyd-Warshall: All-Pairs Shortest Path & Transitive Closure	https://www.youtube.com/watch?v=hTmKNIvOkQc	
74	GATE PYQs on Shortest Paths	https://www.youtube.com/watch?v=_GTk657uO6I	
75	Practice Problems on Shortest Paths	https://www.youtube.com/watch?v=f4LHhAO-x8k	

Chapter 10

Searching and Sorting

Concept: Searching and Sorting — The Big Picture

Searching means finding whether a given value (the **key**) is present in a collection, and if so, where. **Sorting** means rearranging the elements of a collection into increasing (or decreasing) order. The two are closely linked. Sorting costs time once, but afterwards searching becomes much faster (binary search instead of linear search).
Think of a library. If the books are thrown in a heap, you must check every book to find one title (linear search). If the books are arranged alphabetically, you open the middle shelf and decide to go left or right (binary search). Arranging the books is sorting.
In this chapter all sorting algorithms are **comparison-based**: they learn about the order of elements only by comparing two elements at a time.

Concept: Four Properties Used to Compare Sorting Algorithms

- **Stable:** Equal elements keep their original relative order. If two records have the same key and record X comes before record Y in the input, then X still comes before Y in the output. This matters when sorting by multiple keys (sort by marks, then by name).
- **In-place:** The algorithm uses only $O(1)$ extra memory (some books also allow $O(\log n)$ for recursion stack). It sorts inside the same array.
- **Adaptive:** The algorithm runs faster when the input is already partly sorted. For example, it takes $O(n)$ time on a sorted array.
- **Online:** The algorithm can sort the list as new elements keep arriving, without seeing the whole input first.

Concept: Inversions — The Measure of Disorder

In an array $A[0..n-1]$, a pair of positions (i, j) with $i < j$ and $A[i] > A[j]$ is called an **inversion**. It is a pair that is in the wrong order.

- A sorted array has 0 inversions.
- A reverse-sorted array of distinct elements has the maximum, $\binom{n}{2} = \frac{n(n-1)}{2}$ inversions.
- Every swap of two **adjacent** elements changes the inversion count by exactly 1.

Therefore any algorithm that moves elements only by adjacent swaps (bubble sort, insertion sort) needs at least as many swaps or shifts as the number of inversions.

Example 306: Counting Inversions

Array $A = [3, 1, 4, 2]$.

- $(3, 1)$: inversion, since $3 > 1$.
- $(3, 2)$: inversion.

- $(4, 2)$: inversion.
- $(3, 4)$, $(1, 4)$, $(1, 2)$: not inversions.

Total inversions $= 3$. The maximum possible for $n = 4$ is $\frac{4 \cdot 3}{2} = 6$.

10.1 Search Algorithms

10.1.1 Linear Search

Concept: Linear Search

Linear search checks the elements one by one from the beginning, until the key is found or the array ends. It does not need the array to be sorted. It works on arrays and linked lists.

- **Best case:** key at the first position, 1 comparison, $\Theta(1)$.
- **Worst case:** key at the last position or absent, n comparisons, $\Theta(n)$.
- **Average case:** if the key is present and equally likely at any position, $\frac{1 + 2 + \cdots + n}{n} = \frac{n+1}{2}$ comparisons, $\Theta(n)$.

Algorithm: Linear-Search(A, n, key)

```
1: for i = 0 to n − 1 do
2:     if A[i] = key then
3:         return i
4:     end if
5: end for
6: return −1
```

Modified Version: Sentinel Linear Search

In the normal loop, every iteration does **two** tests: "is $i < n$?" and "is $A[i] = key$?". In sentinel search, we place the key itself at the end, $A[n] = key$, so that the loop is guaranteed to stop. Now only one test per iteration is needed. After the loop, we check whether the position found is $< n$. The complexity is still $\Theta(n)$, but the constant becomes smaller. The algorithm needs one extra slot in the array.

Algorithm: Sentinel-Linear-Search(A, n, key)

```
1: last ← A[n − 1]
2: A[n − 1] ← key                          ▷ place sentinel at the last position
3: i ← 0
4: while A[i] ≠ key do
5:     i ← i + 1
6: end while
7: A[n − 1] ← last                         ▷ restore the original element
8: if i < n − 1 or A[n − 1] = key then
9:     return i
10: end if
11: return −1
```

Python implementation: linear-search.py

```
def linear_search(a, key):
    for i in range(len(a)):
        if a[i] == key:
            return i
    return -1
```

```python
def sentinel_search(a, key):
    n = len(a)
    last = a[n - 1]
    a[n - 1] = key               # sentinel
    i = 0
    while a[i] != key:           # only ONE test per iteration
        i += 1
    a[n - 1] = last              # restore
    if i < n - 1 or last == key:
        return i
    return -1


print(linear_search([7, 3, 9, 4], 9))      # 2
print(sentinel_search([7, 3, 9, 4], 5))    # -1
```

10.1.2 Binary Search

Concept: Binary Search

Binary search needs a **sorted** array (with random access). It compares the key with the **middle** element.

- If the key equals the middle element, we are done.
- If the key is smaller, the key can only be in the left half, so we discard the right half.
- If the key is larger, the key can only be in the right half, so we discard the left half.

Each step throws away half of the remaining elements. After k steps, at most $n/2^k$ elements remain. The search ends when this becomes less than 1, i.e., after about $\log_2 n$ steps.

Example 307: Analogy: Guess the Number

Your friend thinks of a number between 1 and 1000. You ask "Is it more than 500?" and he answers yes or no. Every answer halves the possibilities. In at most 10 questions ($2^{10} = 1024 \geq 1000$) you find the number. Binary search is exactly this game.

Algorithm: Binary-Search-Iterative(A, n, key)

1: $lo \leftarrow 0$, $hi \leftarrow n - 1$
2: **while** $lo \leq hi$ **do**
3: $mid \leftarrow lo + \lfloor (hi - lo)/2 \rfloor$
4: **if** $A[mid] = key$ **then**
5: **return** mid
6: **else if** $A[mid] < key$ **then**
7: $lo \leftarrow mid + 1$
8: **else**
9: $hi \leftarrow mid - 1$
10: **end if**
11: **end while**
12: **return** -1

Algorithm: Binary-Search-Recursive(A, lo, hi, key)

```
if lo > hi then
    return −1
end if
mid ← lo + ⌊(hi − lo)/2⌋
if A[mid] = key then
    return mid
else if A[mid] < key then
    return Binary-Search-Recursive(A, mid + 1, hi, key)
else
    return Binary-Search-Recursive(A, lo, mid − 1, key)
end if
```

Example 308: Pseudocode: Binary Search (C-like)

BinarySearch(A, n, key):

1. lo = 0, hi = n - 1
2. while lo ¡= hi:
3. mid = lo + (hi - lo) / 2
4. if A[mid] == key: return mid
5. else if A[mid] ¡ key: lo = mid + 1
6. else: hi = mid - 1
7. return -1

Concept: Recurrence and Complexity of Binary Search

One comparison, then one subproblem of half the size:

$$T(n) = T(n/2) + \Theta(1), \qquad T(1) = \Theta(1)$$

By Master theorem ($a = 1$, $b = 2$, $n^{\log_2 1} = 1 = \Theta(f(n))$, Case 2): $T(n) = \Theta(\log n)$.

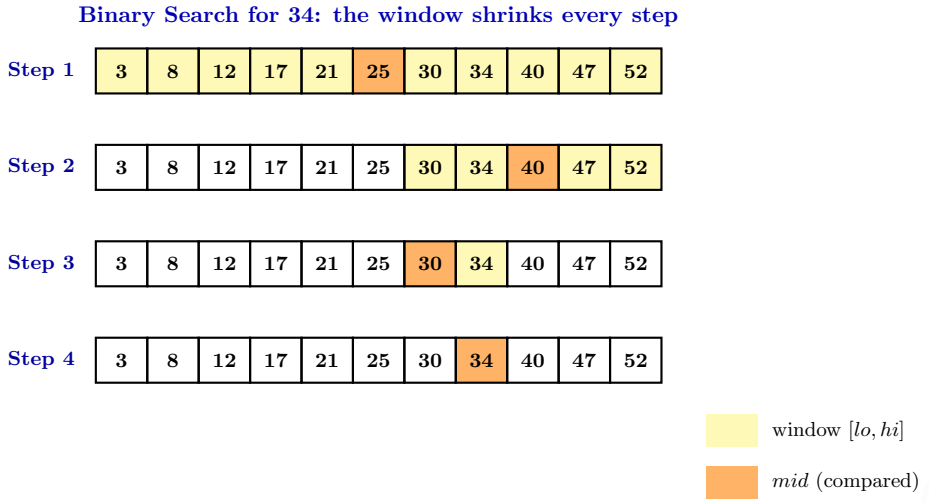
- **Best case:** the key is at the first middle, $\Theta(1)$.
- **Worst and average case:** $\Theta(\log n)$.
- **Space:** iterative $O(1)$; recursive $O(\log n)$ for the call stack.

Example 309: Trace: Binary Search for 34

$A = [3, 8, 12, 17, 21, 25, 30, 34, 40, 47, 52]$ (indices 0 to 10, $n = 11$).

- Iteration 1: $lo = 0$, $hi = 10$, $mid = 5$, $A[5] = 25 < 34$, so $lo = 6$.
- Iteration 2: $lo = 6$, $hi = 10$, $mid = 8$, $A[8] = 40 > 34$, so $hi = 7$.
- Iteration 3: $lo = 6$, $hi = 7$, $mid = 6$, $A[6] = 30 < 34$, so $lo = 7$.
- Iteration 4: $lo = 7$, $hi = 7$, $mid = 7$, $A[7] = 34$. Found at index 7.

Total iterations $= 4 = \lfloor \log_2 11 \rfloor + 1$.



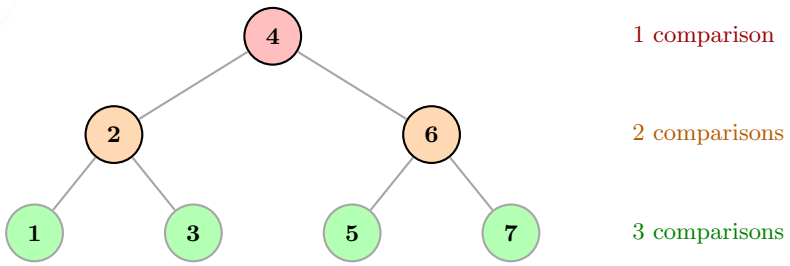
Concept: Exact Number of Comparisons in Binary Search

Counting convention: one comparison per iteration (the three-way test $A[mid]$ against the key). GATE questions usually count iterations this way, but always read the question.

- **Worst case** (successful or unsuccessful): $\lfloor \log_2 n \rfloor + 1$.
- For $n = 2^k - 1$ (a perfect decision tree): the key at the first middle needs 1 comparison, the two keys at the next level need 2 each, the four keys at the next level need 3 each, and so on. In general, 2^{j-1} keys need exactly j comparisons.
- **Average successful search** for $n = 2^k - 1$:

$$\frac{\sum_{j=1}^{k} j \cdot 2^{j-1}}{n} = \frac{(k-1)2^k + 1}{2^k - 1}$$

which is about $\log_2 n - 1$ for large n.



Example 310: Exact Count for $n = 15$

Level 1 has 1 key (1 comparison), level 2 has 2 keys (2 comparisons each), level 3 has 4 keys (3 each), level 4 has 8 keys (4 each).

$$\text{Total} = 1(1) + 2(2) + 4(3) + 8(4) = 1 + 4 + 12 + 32 = 49$$

Average for a successful search $= \frac{49}{15} \approx 3.27$. Worst case $= \lfloor \log_2 15 \rfloor + 1 = 4$.

Concept: Assumptions, Pitfalls and Limitations of Binary Search

- **Needs a sorted array.** On an unsorted array the answer is meaningless.
- **Needs random access** ($O(1)$ access to the middle). On a linked list, reaching the middle costs $O(n)$, so binary search loses its advantage.
- **Overflow:** In fixed-width integers, $mid = (lo + hi)/2$ can overflow. Use $mid = lo + (hi - lo)/2$. (Python integers do not overflow, but the habit is important.)
- **Duplicates:** Plain binary search returns *some* index of the key, not necessarily the first or last. To find the first occurrence (lower bound), do not stop at equality; keep searching on the left.
- **Off-by-one:** The loop condition $lo \leq hi$ with updates $mid \pm 1$ must go together. Mixing $lo < hi$ with $hi = mid$ is a different, valid style, but mixing the two styles by mistake gives infinite loops.
- **Modified version, interpolation search:** instead of the middle, probe at $lo + \frac{(key - A[lo])(hi - lo)}{A[hi] - A[lo]}$. For uniformly distributed keys the average is $O(\log \log n)$, but the worst case is $O(n)$.

Python implementation: binary-search.py

```
def binary_search(a, key):
    lo, hi = 0, len(a) - 1
    while lo <= hi:
        mid = lo + (hi - lo) // 2
        if a[mid] == key:
            return mid
        elif a[mid] < key:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1


def binary_search_rec(a, lo, hi, key):
    if lo > hi:
        return -1
    mid = lo + (hi - lo) // 2
    if a[mid] == key:
        return mid
    if a[mid] < key:
        return binary_search_rec(a, mid + 1, hi, key)
    return binary_search_rec(a, lo, mid - 1, key)


def lower_bound(a, key):
    """Index of the FIRST element >= key (works with duplicates)."""
    lo, hi = 0, len(a)
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if a[mid] < key:
            lo = mid + 1
        else:
            hi = mid            # do not stop: keep looking left
    return lo


def count_iterations(a, key):
    lo, hi, steps = 0, len(a) - 1, 0
    while lo <= hi:
        steps += 1
        mid = lo + (hi - lo) // 2
```

```
        if a[mid] == key:
            return steps
        elif a[mid] < key:
            lo = mid + 1
        else:
            hi = mid - 1
    return steps


A = [3, 8, 12, 17, 21, 25, 30, 34, 40, 47, 52]
print(binary_search(A, 34))          # 7
print(count_iterations(A, 34))       # 4
print(lower_bound([1, 2, 2, 2, 3], 2))    # 1
```

10.1.3 Binary Search on the Answer

Concept: Binary Search on the Answer (Application Pattern)

Sometimes there is no sorted array, but the **answer itself** lies in a range of numbers and has a monotone yes/no property. For example, if speed k works, then every speed larger than k also works. Such a predicate looks like

$$\underbrace{\text{No, No, No,}}_{\text{infeasible}}\ \underbrace{\text{Yes, Yes, Yes}}_{\text{feasible}}$$

We binary search for the **first Yes**. Each step calls a feasibility check feasible(mid) costing C, so the total cost is $O(C \cdot \log(\text{range size}))$.
Template: lo = smallest possible answer, hi = a value that is surely feasible. While $lo < hi$: $mid = lo + (hi - lo)/2$; if feasible(mid) then $hi = mid$ else $lo = mid + 1$. At the end lo is the answer.

Example 311: Example 1: Floor of Square Root

Find $\lfloor \sqrt{x} \rfloor$ for $x = 50$. The largest m with $m^2 \le x$. Search $m \in [0, 50]$ with predicate "$m^2 > x$" (the first Yes is $\lfloor \sqrt{x} \rfloor + 1$). Since $7^2 = 49 \le 50 < 64 = 8^2$, the answer is 7. This takes about $\log_2 50 \approx 6$ steps instead of 7 linear steps, and for $x = 10^{18}$ it takes about 60 steps instead of 10^9.

Example 312: Example 2: Minimum Eating Speed

Piles $= [3, 6, 7, 11]$ bananas, and we have $h = 8$ hours. Each hour we eat from one pile at speed k (a pile of p takes $\lceil p/k \rceil$ hours). Find the minimum integer k that finishes in h hours.
Feasible(k) means $\sum \lceil p_i/k \rceil \le 8$.

- $k = 4$: $1 + 2 + 2 + 3 = 8 \le 8$, feasible.
- $k = 3$: $1 + 2 + 3 + 4 = 10 > 8$, not feasible.

Since feasibility is monotone in k, binary search over $k \in [1, 11]$ gives the minimum $k = 4$.

Python implementation: binary-search-on-answer.py

```
import math

def isqrt_floor(x):
    lo, hi = 0, x + 1                # first m with m*m > x
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if mid * mid > x:            # feasible (Yes)
            hi = mid
        else:
            lo = mid + 1
    return lo - 1
```

```python
def min_speed(piles, h):
    def feasible(k):
        return sum(math.ceil(p / k) for p in piles) <= h

    lo, hi = 1, max(piles)
    while lo < hi:
        mid = lo + (hi - lo) // 2
        if feasible(mid):
            hi = mid
        else:
            lo = mid + 1
    return lo


print(isqrt_floor(50))                  # 7
print(min_speed([3, 6, 7, 11], 8))      # 4
```

10.2 Basic (Quadratic) Sorting

10.2.1 Selection Sort

Concept: Selection Sort

Selection sort repeatedly **selects** the minimum element of the unsorted part and puts it at the front of the unsorted part. After pass i, the first i positions hold the i smallest elements in their final places.
Analogy: Arranging cards in hand by repeatedly picking the smallest card from the remaining pile and placing it next in a row.

Algorithm: Selection-Sort(A, n)

```
for i = 0 to n − 2 do
    min ← i
    for j = i + 1 to n − 1 do
        if A[j] < A[min] then
            min ← j
        end if
    end for
    exchange A[i] with A[min]
end for
```

Example 313: Trace: Selection Sort on $[64, 25, 12, 22, 11]$

- Pass 1: minimum of all $= 11$ (index 4). Swap with index 0: $[11, 25, 12, 22, 64]$. Comparisons: 4.
- Pass 2: minimum of $[25, 12, 22, 64] = 12$. Swap with index 1: $[11, 12, 25, 22, 64]$. Comparisons: 3.
- Pass 3: minimum of $[25, 22, 64] = 22$. Swap with index 2: $[11, 12, 22, 25, 64]$. Comparisons: 2.
- Pass 4: minimum of $[25, 64] = 25$, already at index 3. Self-swap (no change). Comparisons: 1.

Total comparisons $= 4 + 3 + 2 + 1 = 10 = \frac{5 \cdot 4}{2}$. Swaps performed $= 4$ (only 3 if we skip self-swaps).

Properties of Selection Sort

- **Comparisons:** always exactly $\dfrac{n(n-1)}{2}$, for every input (sorted, reversed or random). So best = average = worst $= \Theta(n^2)$.
- **Swaps:** at most $n - 1$, so $O(n)$. This is the *minimum number of writes* among the simple sorts, useful when writing to memory is expensive.

- **Not adaptive:** a sorted array still costs $\Theta(n^2)$ comparisons.
- **In-place:** yes, $O(1)$ extra space.
- **Not stable (in the swap form):** the long-distance swap can jump an element over equal elements. Example: $[4_a, 4_b, 3]$. Pass 1 swaps 4_a with 3 and gives $[3, 4_b, 4_a]$, so 4_a and 4_b are reversed.
- **Not online:** it must see all elements to find the minimum.

Modified Versions of Selection Sort

- **Stable selection sort:** instead of swapping, *shift* the elements between i and min one place to the right and insert the minimum at position i. This keeps equal elements in order, but now the writes become $O(n^2)$.
- **Bidirectional (double) selection sort:** in each pass find both the minimum and the maximum and place them at the two ends. Passes halve, but comparisons stay $\Theta(n^2)$.
- **Heap sort** is selection sort with a smarter way (a heap) of finding the minimum or maximum, reducing $\Theta(n^2)$ to $\Theta(n \log n)$.

Python implementation: selection-sort.py

```python
def selection_sort(a):
    n = len(a)
    comparisons = swaps = 0
    for i in range(n - 1):
        m = i
        for j in range(i + 1, n):
            comparisons += 1
            if a[j] < a[m]:
                m = j
        if m != i:
            a[i], a[m] = a[m], a[i]
            swaps += 1
    return comparisons, swaps


def stable_selection_sort(a):
    n = len(a)
    for i in range(n - 1):
        m = i
        for j in range(i + 1, n):
            if a[j] < a[m]:           # strict: first minimum is chosen
                m = j
        key = a[m]
        while m > i:                  # shift instead of swap
            a[m] = a[m - 1]
            m -= 1
        a[i] = key


A = [64, 25, 12, 22, 11]
print(selection_sort(A), A)           # (10, 3) [11, 12, 22, 25, 64]
```

10.2.2 Bubble Sort

Concept: Bubble Sort

Bubble sort repeatedly walks through the array and compares **adjacent** elements. If a pair is in the wrong order, it swaps them. In each pass, the largest remaining element "bubbles up" to the end like an air bubble rising in water. After pass i, the last i positions hold the i largest elements in final places.

Algorithm: Bubble-Sort(A, n) — with early exit

```
for i = 0 to n − 2 do
    swapped ← false
    for j = 0 to n − 2 − i do
        if A[j] > A[j + 1] then
            exchange A[j] with A[j + 1]
            swapped ← true
        end if
    end for
    if swapped = false then
        break                                   ▷ already sorted, stop
    end if
end for
```

Example 314: Trace: Bubble Sort on $[5, 1, 4, 2, 8]$

- **Pass 1** (4 comparisons): $(5, 1)$ swap $\to [1, 5, 4, 2, 8]$; $(5, 4)$ swap $\to [1, 4, 5, 2, 8]$; $(5, 2)$ swap $\to [1, 4, 2, 5, 8]$; $(5, 8)$ no swap. Result $[1, 4, 2, 5, 8]$, 3 swaps.
- **Pass 2** (3 comparisons): $(1, 4)$ no; $(4, 2)$ swap $\to [1, 2, 4, 5, 8]$; $(4, 5)$ no. 1 swap.
- **Pass 3** (2 comparisons): $(1, 2)$ no; $(2, 4)$ no. No swap, so stop.

Total comparisons $= 4+3+2 = 9$. Total swaps $= 4 =$ number of inversions of the input $((5, 1), (5, 4), (5, 2), (4, 2))$. Passes executed $= 3$ (two to sort, one to confirm).

Properties of Bubble Sort

- **Swaps** = number of inversions I exactly (each swap fixes exactly one inversion).
- **Comparisons:**
  - Without early exit: always $\dfrac{n(n-1)}{2}$.
  - With early exit: best case (sorted) $n - 1$, worst case (reversed) $\dfrac{n(n-1)}{2}$.
- **Time:** best $\Theta(n)$ (only with the early-exit flag), average and worst $\Theta(n^2)$.
- **Number of passes to sort** = maximum, over all elements, of the number of larger elements before it (how far an element must move *left*). Add one more pass if the early-exit flag version must confirm. In the example above: 2 for the element 2 (the larger elements 5 and 4 are before it), so 2 passes sort it, plus 1 confirming pass.
- **Stable:** yes, because we swap only when $A[j] > A[j + 1]$ strictly.
- **In-place:** yes. **Adaptive:** yes, with the flag. **Online:** no.
- **Hare and tortoise behaviour:** large elements near the front ("rabbits") move to the end quickly, but small elements near the end ("turtles") move left only **one place per pass**. A single small element at the end of an otherwise sorted array forces $n - 1$ passes.

Modified Versions of Bubble Sort

- **Early-exit flag** (shown above) makes it adaptive: $\Theta(n)$ on sorted input.
- **Last-swap-position optimisation:** remember the index of the last swap in a pass. Everything after it is already sorted, so the next pass only needs to go up to that index. Good for arrays whose tail is already sorted.
- **Cocktail shaker sort (bidirectional bubble sort):** alternate passes left-to-right (pushes the largest to the end) and right-to-left (pushes the smallest to the front). It handles the "turtle" problem. Still $\Theta(n^2)$ worst case, stable, in-place.

Python implementation: bubble-sort.py

```
def bubble_sort(a):
    n = len(a)
    comparisons = swaps = passes = 0
    for i in range(n - 1):
        swapped = False
        passes += 1
        for j in range(n - 1 - i):
            comparisons += 1
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swaps += 1
                swapped = True
        if not swapped:
            break
    return comparisons, swaps, passes


def bubble_sort_last_swap(a):
    """Shrinks the range using the position of the last swap."""
    n = len(a)
    end = n - 1
    while end > 0:
        last = 0
        for j in range(end):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                last = j
        end = last


def cocktail_sort(a):
    lo, hi = 0, len(a) - 1
    swapped = True
    while swapped and lo < hi:
        swapped = False
        for j in range(lo, hi):                # forward pass
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
                swapped = True
        hi -= 1
        for j in range(hi, lo, -1):            # backward pass
            if a[j - 1] > a[j]:
                a[j - 1], a[j] = a[j], a[j - 1]
                swapped = True
        lo += 1


A = [5, 1, 4, 2, 8]
print(bubble_sort(A), A)      # (9, 4, 3) [1, 2, 4, 5, 8]
```

10.2.3 Insertion Sort

Concept: Insertion Sort

Insertion sort keeps a sorted prefix $A[0..i-1]$ and takes the next element (the **key**), $A[i]$. It shifts the larger elements of the prefix one place to the right and drops the key into the gap. This is exactly how we arrange playing cards in hand: pick up one card at a time and insert it at the correct place among the cards already held.

Algorithm: Insertion-Sort(A, n)

```
1: for i = 1 to n − 1 do
2:     key ← A[i]
3:     j ← i − 1
4:     while j ≥ 0 and A[j] > key do
5:         A[j + 1] ← A[j]                ▷ shift right
6:         j ← j − 1
7:     end while
8:     A[j + 1] ← key
9: end for
```

Example 315: Trace: Insertion Sort on $[5, 2, 4, 6, 1, 3]$

Count the comparisons of the form $A[j] > key$.

- $key = 2$: compare with 5 ($>$, shift). Reached the left end. Array $[2, 5, 4, 6, 1, 3]$. Comparisons 1, shifts 1.
- $key = 4$: compare with 5 (shift), with 2 (stop). Array $[2, 4, 5, 6, 1, 3]$. Comparisons 2, shifts 1.
- $key = 6$: compare with 5 (stop). Array unchanged. Comparisons 1, shifts 0.
- $key = 1$: compare with $6, 5, 4, 2$ (all shift), left end reached. Array $[1, 2, 4, 5, 6, 3]$. Comparisons 4, shifts 4.
- $key = 3$: compare with $6, 5, 4$ (shift), with 2 (stop). Array $[1, 2, 3, 4, 5, 6]$. Comparisons 4, shifts 3.

Total comparisons $= 1 + 2 + 1 + 4 + 4 = 12$. Total shifts $= 9 =$ inversions of the input.
General formula: **shifts** $= I$ (number of inversions); **comparisons** $\leq I + (n-1)$. Here $9+5 = 14$, minus 2 because two keys (2 and 1) reached the left end without a failing comparison, giving 12.

Properties of Insertion Sort

- **Best case** (already sorted): $n - 1$ comparisons, 0 shifts, $\Theta(n)$.
- **Worst case** (reverse sorted): comparisons $=$ shifts $= \dfrac{n(n-1)}{2}$, $\Theta(n^2)$.
- **Average case:** about $\dfrac{n(n-1)}{4}$ inversions, so $\Theta(n^2)$.
- **Running time** $= \Theta(n + I)$, directly proportional to the number of inversions.
- **Nearly sorted input:** if every element is at most k places away from its final position, the time is $O(nk)$.
- **Stable:** yes (we shift only elements *strictly* greater than the key).
- **In-place:** yes. **Adaptive:** yes. **Online:** yes, a new element simply gets inserted into the sorted prefix.
- **Fastest** of the simple sorts for small or nearly sorted arrays; production sorts (TimSort, introsort) use it for small subarrays.

Modified Versions of Insertion Sort

- **Binary insertion sort:** find the insertion position with **binary search** on the sorted prefix. Comparisons drop to $O(n \log n)$, but the shifting still costs $O(n^2)$ moves, so the total time is still $\Theta(n^2)$. It loses the $\Theta(n)$ best case (it always does about $\log i$ comparisons per key). It remains stable if we search for the position after the last equal element.
- **Shell sort:** insertion sort on elements that are h apart, for a decreasing sequence of gaps h, ending with $h = 1$. It lets elements jump long distances and beats $\Theta(n^2)$ in practice. Not stable.
- **Linked-list insertion sort:** no shifting is needed (just relink), but finding the position is still linear.

Python implementation: insertion-sort.py

```python
import bisect

def insertion_sort(a):
    comparisons = shifts = 0
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0:
            comparisons += 1
            if a[j] > key:
                a[j + 1] = a[j]
                shifts += 1
                j -= 1
            else:
                break
        a[j + 1] = key
    return comparisons, shifts


def binary_insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]
        pos = bisect.bisect_right(a, key, 0, i)    # after equal elements: stable
        for j in range(i, pos, -1):                # still O(i) shifting
            a[j] = a[j - 1]
        a[pos] = key


A = [5, 2, 4, 6, 1, 3]
print(insertion_sort(A), A)     # (12, 9) [1, 2, 3, 4, 5, 6]
```

10.2.4 Comparison of the Three Quadratic Sorts

Property	Selection	Bubble (flag)	Insertion
Comparisons, best	$\frac{n(n-1)}{2}$	$n-1$	$n-1$
Comparisons, worst	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
Swaps / moves, worst	$n-1$	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
Best-case time	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$
Stable	No	Yes	Yes
In-place	Yes	Yes	Yes
Adaptive	No	Yes	Yes
Online	No	No	Yes
Swaps equal inversions?	No	Yes	Yes (shifts)

10.3 Divide and Conquer Sorting

10.3.1 Merge Sort

Concept: Merge Sort

Merge sort follows the divide-and-conquer paradigm:

1. **Divide:** split the array into two halves.
2. **Conquer:** sort each half recursively (a single element is already sorted).
3. **Combine: merge** the two sorted halves into one sorted array.

The real work is in the merge step. Merging two sorted lists is easy: keep one pointer at the front of each list, repeatedly take the smaller front element, and move that pointer.

Analogy: Two sorted piles of exam papers (by roll number) on a table. To make one sorted pile, repeatedly compare the top papers of the two piles and take the one with the smaller roll number.

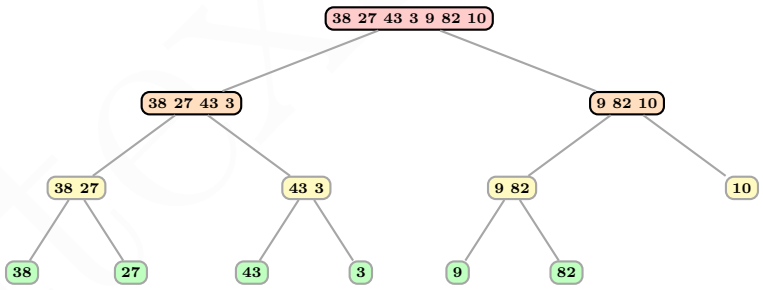
Algorithm: Merge-Sort(A, p, r)

```
1: if p < r then
2:     q ← ⌊(p + r)/2⌋
3:     Merge-Sort(A, p, q)
4:     Merge-Sort(A, q + 1, r)
5:     Merge(A, p, q, r)
6: end if
```

Algorithm: Merge(A, p, q, r) — CLRS style with sentinels

```
1: n1 ← q − p + 1, n2 ← r − q
2: create arrays L[1..n1 + 1] and R[1..n2 + 1]
3: for i = 1 to n1 do L[i] ← A[p + i − 1]
4: end for
5: for j = 1 to n2 do R[j] ← A[q + j]
6: end for
7: L[n1 + 1] ← ∞, R[n2 + 1] ← ∞                          ▷ sentinels
8: i ← 1, j ← 1
9: for k = p to r do
10:     if L[i] ≤ R[j] then                              ▷ ≤ keeps the sort stable
11:         A[k] ← L[i], i ← i + 1
12:     else
13:         A[k] ← R[j], j ← j + 1
14:     end if
15: end for
```

Merge Sort: the DIVIDE phase on $[38, 27, 43, 3, 9, 82, 10]$



Example 316: Trace: The MERGE (combine) Phase

Going up from the leaves:

- $[38] + [27] \to [27, 38]$. $[43] + [3] \to [3, 43]$.
- $[27, 38] + [3, 43] \to [3, 27, 38, 43]$.
- $[9] + [82] \to [9, 82]$. $[9, 82] + [10] \to [9, 10, 82]$.
- Final merge: $[3, 27, 38, 43] + [9, 10, 82]$. Compare 3 vs 9 (take 3), 27 vs 9 (take 9), 27 vs 10 (take 10), 27 vs 82 (take 27), 38 vs 82 (take 38), 43 vs 82 (take 43). Left list is empty; copy the remaining 82 without comparison. Comparisons $= 6$.

Result: $[3, 9, 10, 27, 38, 43, 82]$.

Concept: Analysis of Merge Sort

Recurrence: $T(n) = 2T(n/2) + \Theta(n)$. By Master theorem ($a = 2$, $b = 2$, $n^{\log_2 2} = n = \Theta(f(n))$, Case 2): $T(n) = \Theta(n \log n)$.
By recursion tree: there are $\log_2 n$ levels, and each level does total merge work of about n.
Merge comparisons: merging sorted lists of sizes m and n needs

- at most $m + n - 1$ comparisons (the largest element ends up in one list and the other list is emptied last);
- at least $\min(m, n)$ comparisons (one list entirely smaller than the other).

Comparisons for the whole sort ($n = 2^k$):

- Worst case: $n \log_2 n - n + 1$. For $n = 8$: $8 \cdot 3 - 8 + 1 = 17$.
- Best case (already sorted input): $\frac{n}{2} \log_2 n$. For $n = 8$: 12.

Properties of Merge Sort

- **Time:** $\Theta(n \log n)$ in best, average and worst case. The recurrence does not depend on the input order.
- **Stable:** yes, provided the merge takes from the left list when the elements are equal ($\leq$).
- **Not in-place:** needs $\Theta(n)$ auxiliary array for arrays. (Recursion stack adds $O(\log n)$.)
- **Not adaptive** in the basic form (still $\Theta(n \log n)$ on sorted input; only the comparison constant drops).
- **Not online.**
- **Good for:** linked lists (merging needs only $O(1)$ extra space by relinking), external sorting of data larger than memory, and guaranteed $O(n \log n)$.

Modified Versions of Merge Sort

- **Bottom-up (iterative) merge sort:** merge runs of size 1, then 2, then 4, and so on. No recursion, same $\Theta(n \log n)$.
- **Natural merge sort:** first detect the already-sorted runs in the input and merge those. This is **adaptive**: $\Theta(n)$ if the input is sorted.
- **TimSort** (used by Python): natural merge sort plus insertion sort for short runs. Stable, adaptive, worst case $\Theta(n \log n)$.
- **Merge sort on linked lists:** $O(1)$ extra space, still stable.
- **Counting inversions:** while merging, whenever an element from the *right* list is taken before the remaining x elements of the left list, those x pairs are inversions. This counts inversions in $\Theta(n \log n)$ instead of $\Theta(n^2)$.

Python implementation: merge-sort.py

```
def merge_sort(a):
    if len(a) <= 1:
        return a
    mid = len(a) // 2
    left = merge_sort(a[:mid])
    right = merge_sort(a[mid:])
    return merge(left, right)


def merge(left, right):
    out = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:          # <= keeps the sort stable
            out.append(left[i]); i += 1
```

```
        else:
            out.append(right[j]); j += 1
    out.extend(left[i:])                    # one of these is empty
    out.extend(right[j:])
    return out


def sort_and_count(a):
    """Returns (sorted list, number of inversions) in O(n log n)."""
    if len(a) <= 1:
        return a, 0
    mid = len(a) // 2
    left, c1 = sort_and_count(a[:mid])
    right, c2 = sort_and_count(a[mid:])
    out, i, j, cross = [], 0, 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            out.append(left[i]); i += 1
        else:
            out.append(right[j]); j += 1
            cross += len(left) - i        # left[i:] are all > right[j]
    out.extend(left[i:]); out.extend(right[j:])
    return out, c1 + c2 + cross


def bottom_up_merge_sort(a):
    n, width = len(a), 1
    while width < n:
        for lo in range(0, n, 2 * width):
            mid = min(lo + width, n)
            hi = min(lo + 2 * width, n)
            a[lo:hi] = merge(a[lo:mid], a[mid:hi])
        width *= 2


print(merge_sort([38, 27, 43, 3, 9, 82, 10]))
print(sort_and_count([5, 2, 4, 6, 1, 3])[1])      # 9
```

10.3.2 Quicksort

Concept: Quicksort

Quicksort also uses divide and conquer, but the hard work is in the **divide** step (partition), and the combine step is empty.

1. **Divide (partition):** choose a **pivot** and rearrange the array so that elements $\leq$ pivot come before it and elements $>$ pivot come after it. The pivot is now in (or near) its final place.
2. **Conquer:** sort the left part and the right part recursively.
3. **Combine:** nothing to do, since the array is already in order.

Analogy: Arranging a queue by height. Pick one person as the pivot, ask shorter people to stand on his left and taller people on his right, then repeat the idea inside each group.

Algorithm: Quicksort(A, p, r)

1: **if** $p < r$ **then**
2: $q \leftarrow$ Partition(A, p, r)
3: Quicksort(A, p, $q-1$)
4: Quicksort(A, $q+1$, r)
5: **end if**

Lomuto Partition Scheme (CLRS)

Concept: Lomuto Partition

The pivot is the **last** element $x = A[r]$. We keep an index i such that:

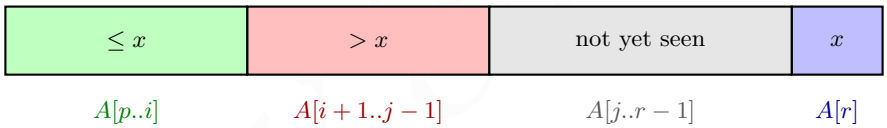
- $A[p..i]$ holds elements $\leq x$ (the "small" region),
- $A[i+1..j-1]$ holds elements $> x$ (the "large" region),
- $A[j..r-1]$ is not yet examined.

For each j, if $A[j] \leq x$, grow the small region by incrementing i and swapping $A[i]$ with $A[j]$. At the end, put the pivot between the two regions by swapping $A[i+1]$ and $A[r]$. The pivot lands at its **final sorted position**, $i+1$. Partition on n elements does exactly $n-1$ comparisons.

Algorithm: Partition(A, p, r) — Lomuto

```
1:  x ← A[r]                                              ▷ pivot
2:  i ← p − 1
3:  for j = p to r − 1 do
4:      if A[j] ≤ x then
5:          i ← i + 1
6:          exchange A[i] with A[j]
7:      end if
8:  end for
9:  exchange A[i + 1] with A[r]
10: return i + 1
```

Invariant of Lomuto Partition during the loop



Example 317: Trace: Lomuto Partition on $[2, 8, 7, 1, 3, 5, 6, 4]$

Pivot $x = 4$ (last element). $i = -1$ at the start (array indexed from 0).

- $j = 0$: $2 \leq 4$. $i = 0$, swap $A[0]$ with itself. $[2, 8, 7, 1, 3, 5, 6, 4]$
- $j = 1$: $8 > 4$. No change.
- $j = 2$: $7 > 4$. No change.
- $j = 3$: $1 \leq 4$. $i = 1$, swap $A[1]$ and $A[3]$. $[2, 1, 7, 8, 3, 5, 6, 4]$
- $j = 4$: $3 \leq 4$. $i = 2$, swap $A[2]$ and $A[4]$. $[2, 1, 3, 8, 7, 5, 6, 4]$
- $j = 5$: $5 > 4$. $j = 6$: $6 > 4$. No change.
- End: swap $A[i+1] = A[3]$ with $A[7]$. $[2, 1, 3, 4, 7, 5, 6, 8]$. Pivot index $= 3$.

Left part $[2, 1, 3]$, right part $[7, 5, 6, 8]$. Comparisons in this partition: $7 = n - 1$.

Hoare Partition Scheme

Concept: Hoare Partition

The original scheme of C.A.R. Hoare. The pivot is $x = A[p]$ (the first element). Two pointers move towards each other:

- i starts at the left and moves right until it finds $A[i] \geq x$.

- j starts at the right and moves left until it finds $A[j] \leq x$.
- If $i < j$, swap $A[i]$ and $A[j]$ and continue. Otherwise return j.

The recursion is then on $A[p..j]$ and $A[j+1..r]$. Important differences from Lomuto:

- The pivot is **not** guaranteed to land at its final position.
- It does about **three times fewer swaps** on average, so it is faster in practice.
- The recursive calls use (p, j) and $(j+1, r)$, not $(p, q-1)$ and $(q+1, r)$.
- With the pivot as the *first* element and this version, using the last element as pivot can loop forever, so keep the pivot choice and the return value consistent.

Algorithm: Hoare-Partition(A, p, r)

```
1: x ← A[p]
2: i ← p − 1, j ← r + 1
3: while true do
4:     repeatj ← j − 1
5:     until A[j] ≤ x
6:     repeati ← i + 1
7:     until A[i] ≥ x
8:     if i < j then
9:         exchange A[i] with A[j]
10:    else
11:        return j
12:    end if
13: end while
```

Example 318: Trace: Hoare Partition on $[5, 3, 8, 4, 2, 7, 1]$

Pivot $x = 5$. Start $i = -1$, $j = 7$.

- j moves left: $A[6] = 1 \leq 5$, stop at $j = 6$. i moves right: $A[0] = 5 \geq 5$, stop at $i = 0$. $i < j$, swap: $[1, 3, 8, 4, 2, 7, 5]$.
- j: $A[5] = 7 > 5$, $A[4] = 2 \leq 5$, stop at $j = 4$. i: $A[1] = 3 < 5$, $A[2] = 8 \geq 5$, stop at $i = 2$. Swap: $[1, 3, 2, 4, 8, 7, 5]$.
- j: $A[3] = 4 \leq 5$, stop at $j = 3$. i: $A[3] = 4 < 5$, $A[4] = 8 \geq 5$, stop at $i = 4$. Now $i \geq j$, return $j = 3$.

Partitions: $[1, 3, 2, 4]$ and $[8, 7, 5]$. Every element of the left part is $\leq 5 \leq$ every element of the right part. Note the pivot 5 is in the right part, not at a fixed final position.

Analysis of Quicksort

Concept: Running Time of Quicksort

Let the partition split n elements into parts of size k and $n-k-1$ (Lomuto). Then

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

- **Worst case:** the split is always $0 : n-1$ (the pivot is always the smallest or largest element).

$$T(n) = T(n-1) + \Theta(n) \;\Rightarrow\; T(n) = \Theta(n^2), \quad \text{comparisons} = \frac{n(n-1)}{2}$$

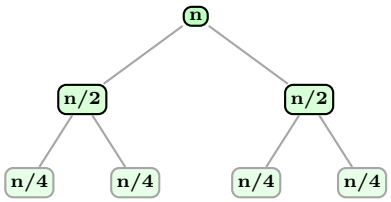
  For $n = 5$ this is $4 + 3 + 2 + 1 = 10$ comparisons.
- **Best case:** the pivot is always the median, $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.
- **Any constant-proportion split is still $\Theta(n \log n)$.** For example, a $9 : 1$ split every time gives $T(n) = T(9n/10) + T(n/10) + n$. The recursion tree has depth about $\log_{10/9} n$ on the long side, and each level costs at most n, so $T(n) = \Theta(n \log n)$. This is why quicksort is good even with an "unlucky but not terrible" pivot.

- **Average case:** for a random input order, $E[C_n] = (n-1) + \frac{2}{n}\sum_{k=0}^{n-1} E[C_k]$, which solves to $\approx 2n \ln n \approx 1.39\, n \log_2 n$ comparisons. So the average is $\Theta(n \log n)$, only about 39% more comparisons than the ideal.

Concept: When is Quicksort Worst? — Traps for GATE

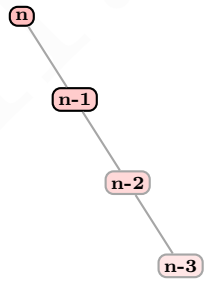
- **Sorted or reverse-sorted input with first or last element as pivot:** every partition is $0 : n-1$, so $\Theta(n^2)$. (Sorted + last-element pivot: the pivot is the maximum. Reverse sorted + last-element pivot: the pivot is the minimum.)
- **All elements equal** with Lomuto's "$\leq$" test: every element goes to the left part, so the split is again $n-1 : 0$ and the time is $\Theta(n^2)$.
- **Middle element as pivot** makes sorted input the *best* case, but an adversary can still construct a bad input for any fixed pivot rule.
- **Pivot is always the median** (found in $O(n)$ time) gives a guaranteed $\Theta(n \log n)$ worst case, but with a big constant.

Best case: balanced



depth $\log n$, total $\Theta(n \log n)$

Worst case: skewed



depth n, total $\Theta(n^2)$

Properties of Quicksort

- **Time:** best and average $\Theta(n \log n)$, worst $\Theta(n^2)$.
- **In-place:** yes, partitioning rearranges within the array. The recursion stack uses $O(\log n)$ on average, but $O(n)$ in the worst case.
- **Not stable:** the long-distance swaps in partition can reorder equal elements. Example with Lomuto on $[3_a, 3_b, 2]$: pivot 2; no element is ≤ 2 except the pivot itself, and the final swap gives $[2, 3_b, 3_a]$. The two 3s are reversed.
- **Not adaptive** in the basic form, and not online.
- **Fast in practice:** good cache behaviour (sequential access) and a small constant factor, usually faster than merge sort on arrays.

Modified Versions of Quicksort

- **Randomised quicksort:** choose the pivot uniformly at random (or swap a random element to the pivot position first). No fixed input is bad; for *every* input the **expected** time is $O(n \log n)$. The worst case $O(n^2)$ still exists but has negligible probability.
- **Median-of-three pivot:** use the median of the first, middle and last elements. Sorted input is handled well.
- **Three-way partitioning (Dutch national flag):** split into $<$ pivot, $=$ pivot, $>$ pivot. Equal elements are placed once and never touched again, so inputs with many duplicates run in $O(n)$ for all-equal input.
- **Smaller-part-first recursion + tail-call elimination:** recurse on the smaller side and loop on the larger side. The stack depth is at most $\log_2 n$, so the stack space is $O(\log n)$ even in the worst case.

- **Introsort:** start with quicksort; if the recursion depth exceeds $2\log_2 n$, switch to heap sort. Guarantees $O(n \log n)$ worst case. Used in C++ `std::sort`.
- **Cutoff to insertion sort** for tiny subarrays (below about 10 elements).

Example 319: Pseudocode: Quicksort with Lomuto Partition (C-like)

Partition(A, p, r):

1. x = A[r], i = p - 1
2. for j = p to r - 1:
3. if A[j] <= x: i = i + 1, swap(A[i], A[j])
4. swap(A[i+1], A[r])
5. return i + 1

Quicksort(A, p, r):

1. if p ¡ r:
2. q = Partition(A, p, r)
3. Quicksort(A, p, q - 1)
4. Quicksort(A, q + 1, r)

Python implementation: quicksort.py

```
import random

def lomuto_partition(a, p, r):
    x = a[r]
    i = p - 1
    for j in range(p, r):
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i + 1], a[r] = a[r], a[i + 1]
    return i + 1


def quicksort(a, p, r):
    if p < r:
        q = lomuto_partition(a, p, r)
        quicksort(a, p, q - 1)
        quicksort(a, q + 1, r)


def hoare_partition(a, p, r):
    x = a[p]
    i, j = p - 1, r + 1
    while True:
        j -= 1
        while a[j] > x:
            j -= 1
        i += 1
        while a[i] < x:
            i += 1
        if i < j:
            a[i], a[j] = a[j], a[i]
        else:
            return j
```

```
def quicksort_hoare(a, p, r):
    if p < r:
        q = hoare_partition(a, p, r)
        quicksort_hoare(a, p, q)          # note: q, not q-1
        quicksort_hoare(a, q + 1, r)


def randomized_quicksort(a, p, r):
    if p < r:
        k = random.randint(p, r)
        a[k], a[r] = a[r], a[k]           # random pivot moved to the end
        q = lomuto_partition(a, p, r)
        randomized_quicksort(a, p, q - 1)
        randomized_quicksort(a, q + 1, r)


def three_way_quicksort(a, lo, hi):
    """Dutch national flag partition: handles duplicates well."""
    if lo >= hi:
        return
    x = a[lo]
    lt, i, gt = lo, lo + 1, hi
    while i <= gt:
        if a[i] < x:
            a[lt], a[i] = a[i], a[lt]; lt += 1; i += 1
        elif a[i] > x:
            a[i], a[gt] = a[gt], a[i]; gt -= 1
        else:
            i += 1
    three_way_quicksort(a, lo, lt - 1)
    three_way_quicksort(a, gt + 1, hi)


A = [2, 8, 7, 1, 3, 5, 6, 4]
print(lomuto_partition(A[:], 0, 7), A)       # pivot index 3
B = [2, 8, 7, 1, 3, 5, 6, 4]
quicksort(B, 0, len(B) - 1)
print(B)                                      # [1, 2, 3, 4, 5, 6, 7, 8]
```

10.3.3 Lower Bound for Comparison-Based Sorting

Theorem: Any Comparison Sort Needs $\Omega(n \log n)$ Comparisons

A comparison-based sorting algorithm can be viewed as a **binary decision tree**. Each internal node is a comparison "is $a_i < a_j$?", and each leaf is one possible output ordering (a permutation of the input).

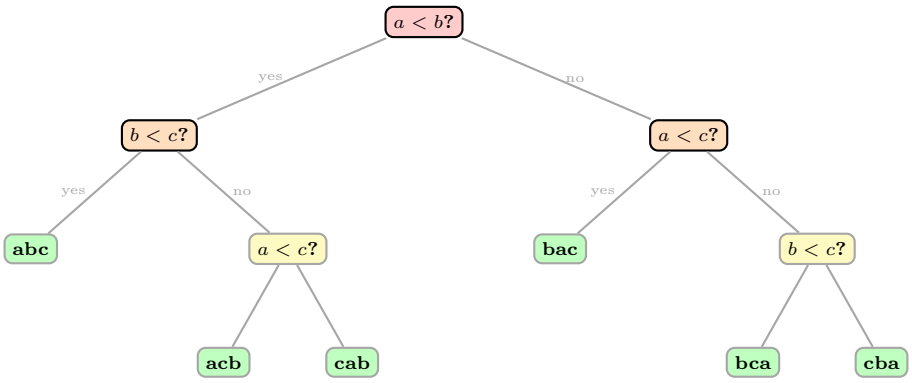
- There are $n!$ possible input orderings, and the algorithm must be able to produce each of them, so the tree has **at least $n!$ leaves**.
- A binary tree of height h has at most 2^h leaves. So $2^h \geq n!$, which gives

  $$h \geq \log_2(n!)$$

- The height h is the number of comparisons in the worst case. Using Stirling's approximation, $\log_2(n!) = n \log_2 n - n \log_2 e + O(\log n) = \Theta(n \log n)$.

Hence every comparison sort needs $\Omega(n \log n)$ comparisons in the worst case. This proves that merge sort (and heap sort) are asymptotically optimal.
Minimum worst-case comparisons to sort n elements is at least $\lceil \log_2 n! \rceil$: for $n = 3$, $\lceil \log_2 6 \rceil = 3$; for $n = 4$, $\lceil \log_2 24 \rceil = 5$; for $n = 5$, $\lceil \log_2 120 \rceil = 7$.
Searching a sorted array: the decision tree for a key among n sorted elements has n successful outcomes, so the height is at least $\lceil \log_2(n+1) \rceil$. Binary search meets this bound.

Decision tree for sorting 3 elements a, b, c: $3! = 6$ leaves, height 3



Example 320: Applying the Lower Bound

Can we sort 6 numbers using only 9 comparisons in the worst case? $6! = 720$ and $2^9 = 512 < 720$. A decision tree with 9 levels has at most 512 leaves, which is less than 720, so it is **impossible**. We need at least $\lceil \log_2 720 \rceil = 10$ comparisons.

10.4 Randomised Quicksort

Concept: Why Randomise the Pivot

In ordinary quicksort the pivot rule is fixed (first, last or middle element). So for every fixed rule there is some input, such as a sorted array, that gives $\Theta(n^2)$ time. A person who knows your rule can always give you that bad input.

In **randomised quicksort**, the pivot is chosen **uniformly at random** from the current subarray. Now the running time depends on the coin tosses of the algorithm and not on the order of the input. No input is bad in itself. For **every** input, the **expected** running time is $\Theta(n \log n)$.

Analogy: A teacher picks a student at random as the "pivot" for splitting the class by height. Because the student is picked by lottery, nobody can arrange the class in advance to make the teacher unlucky.

Algorithm: Randomized-Partition(A, p, r)

1: $i \leftarrow$ Random(p, r) ▷ uniform integer in $[p, r]$
2: exchange $A[r]$ with $A[i]$
3: **return** Partition(A, p, r) ▷ Lomuto partition, pivot is now $A[r]$

Algorithm: Randomized-Quicksort(A, p, r)

1: **if** $p < r$ **then**
2: $q \leftarrow$ Randomized-Partition(A, p, r)
3: Randomized-Quicksort(A, p, $q - 1$)
4: Randomized-Quicksort(A, $q + 1$, r)
5: **end if**

Example 321: Pseudocode: Randomised Quicksort (C-like)

RandomizedPartition(A, p, r):

1. i = random integer in [p, r]
2. swap(A[r], A[i])
3. return Partition(A, p, r)

RandomizedQuicksort(A, p, r):

1. if p ¡ r:

2. q = RandomizedPartition(A, p, r)
3. RandomizedQuicksort(A, p, q - 1)
4. RandomizedQuicksort(A, q + 1, r)

10.4.1 Examples

Example 322: Example 1: One Run on a Sorted Array

Sort $A = [1, 2, 3, 4, 5]$, which is the worst input for last-element pivot.
Call 1: the random index chosen is 2 (value 3). Swap with $A[4]$: $[1, 2, 5, 4, 3]$. Lomuto with pivot 3: elements $1, 2$ are ≤ 3, elements $5, 4$ are not. Final swap gives $[1, 2, 3, 4, 5]$ with pivot index 2. Split sizes $2 : 2$. Comparisons: 4.
Left part $[1, 2]$: random pivot, say index 0. Swap: $[2, 1]$. Pivot 1: $2 > 1$. Final swap gives $[1, 2]$. Comparisons: 1.
Right part $[4, 5]$ similarly costs 1 comparison.
Total comparisons $= 4 + 1 + 1 = 6$, instead of the 10 that last-element pivot needs on this input.

Example 323: Example 2: Number of Comparisons Depends on Luck

For $n = 3$ distinct elements, the first pivot is the smallest, the middle or the largest, each with probability $\frac{1}{3}$.

- Pivot is the middle element (probability $\frac{1}{3}$): the split is $1 : 1$. Comparisons $= 2$ (partition) $+0 + 0 = 2$.
- Pivot is the smallest or largest (probability $\frac{2}{3}$): the split is $0 : 2$. Comparisons $= 2$ (partition) $+1$ (sorting two elements) $= 3$.

Expected comparisons $= \frac{1}{3}(2) + \frac{2}{3}(3) = \frac{8}{3} \approx 2.67$.

Example 324: Example 3: Probability of the Worst Case

For $n = 4$, the maximum number of comparisons is $3 + 2 + 1 = 6$. This happens when every pivot is the minimum or maximum of its subarray. The first call has 2 good choices out of 4, the second has 2 out of 3, the third has 2 out of 2.

$$\Pr[6 \text{ comparisons}] = \frac{2}{4} \cdot \frac{2}{3} \cdot \frac{2}{2} = \frac{1}{3}$$

In general, the probability of the fully skewed run is $\frac{2^{n-1}}{n!}$. For $n = 10$ this is $\frac{512}{3628800} \approx 0.00014$, and it falls extremely fast as n grows.

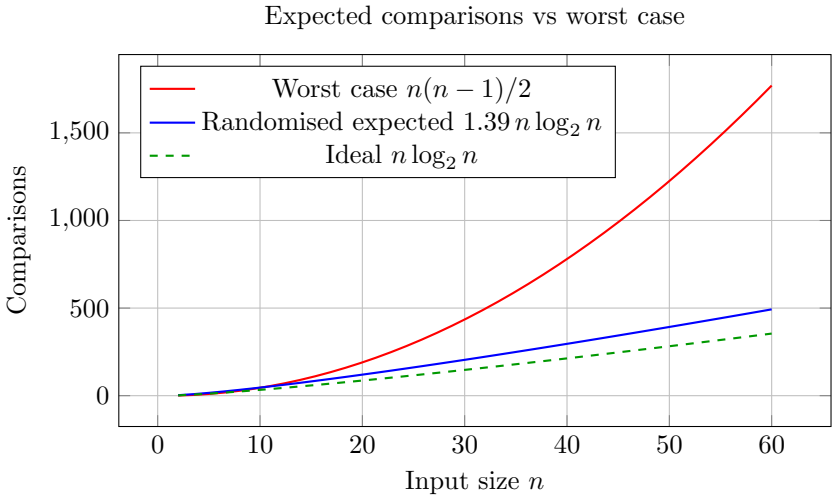
Example 325: Example 4: Probability that Two Elements Are Compared

Sort 6 distinct elements. What is the probability that the elements of rank 2 and rank 5 are ever compared?
They are compared only if one of them is the **first pivot chosen** among the ranks $2, 3, 4, 5$. If any element of rank 3 or 4 is chosen first, it separates them forever. There are $5 - 2 + 1 = 4$ elements in the range, and 2 of them are good choices.

$$\Pr = \frac{2}{4} = \frac{1}{2}$$

10.4.2 Performance

Input	Last-element pivot	Randomised pivot (expected)
Sorted	$\Theta(n^2)$	$\Theta(n \log n)$
Reverse sorted	$\Theta(n^2)$	$\Theta(n \log n)$
Random order	$\Theta(n \log n)$	$\Theta(n \log n)$
All keys equal (Lomuto)	$\Theta(n^2)$	$\Theta(n^2)$



Python implementation: randomized-quicksort.py

```
import random

comparisons = 0

def partition(a, p, r):
    global comparisons
    x = a[r]
    i = p - 1
    for j in range(p, r):
        comparisons += 1
        if a[j] <= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    a[i + 1], a[r] = a[r], a[i + 1]
    return i + 1


def randomized_partition(a, p, r):
    i = random.randint(p, r)          # uniform in [p, r]
    a[r], a[i] = a[i], a[r]
    return partition(a, p, r)


def randomized_quicksort(a, p, r):
    while p < r:                      # loop on the larger part
        q = randomized_partition(a, p, r)
        if q - p < r - q:             # recurse on the smaller part
            randomized_quicksort(a, p, q - 1)
            p = q + 1
        else:
            randomized_quicksort(a, q + 1, r)
            r = q - 1


def average_comparisons(n, trials=2000):
    global comparisons
    total = 0
    for _ in range(trials):
        a = list(range(n))            # sorted input: worst for fixed pivot
        comparisons = 0
        randomized_quicksort(a, 0, n - 1)
        total += comparisons
    return total / trials


for n in (3, 10, 100, 1000):
    print(n, round(average_comparisons(n), 1))
# n=3     ~ 2.67
# n=100   ~ 648     (formula 2(n+1)H(n+1) - 4n)
```

```
# n=1000 ~ 10987
```

10.5 Master Summary

Algorithm	Best	Average	Worst	Extra space	Stable	In-place	Adaptive	Online
Selection sort	n^2	n^2	n^2	$O(1)$	No	Yes	No	No
Bubble sort (flag)	n	n^2	n^2	$O(1)$	Yes	Yes	Yes	No
Insertion sort	n	n^2	n^2	$O(1)$	Yes	Yes	Yes	Yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes	No	No	No
Quicksort (fixed pivot)	$n \log n$	$n \log n$	n^2	$O(\log n)$ avg	No	Yes	No	No
Randomised quicksort	$n \log n$	$n \log n$ (expected)	n^2 (very unlikely)	$O(\log n)$ expected	No	Yes	No	No

Search	Best	Worst	Requirement
Linear search	1	n	none
Binary search	1	$\lfloor \log_2 n \rfloor + 1$	sorted, random access

Example 326: GATE Exam Tips for Searching and Sorting

- **Count the right thing.** Some questions ask comparisons, some ask swaps, some ask passes. Selection sort: comparisons fixed at $\frac{n(n-1)}{2}$. Bubble and insertion: swaps (shifts) = number of inversions.
- **Stability traps.** Stable: bubble, insertion, merge. Not stable: selection, quicksort (and heap sort). Selection sort can be made stable by shifting; merge sort becomes unstable if the merge uses $<$ instead of $\leq$.
- **Quicksort worst case.** Sorted or reverse-sorted input with first or last pivot, and also all-equal elements with Lomuto. Recurrence $T(n) = T(n-1) + n$. A constant split ratio (like $9:1$) is still $\Theta(n \log n)$.
- **Randomised quicksort.** Expected time is $O(n \log n)$ for every input, worst case is still $O(n^2)$. Probability that ranks $i < j$ are compared is $\frac{2}{j-i+1}$.
- **In-place is not the same as no extra stack.** Quicksort is in-place (it needs only $O(\log n)$ stack on average). Merge sort on arrays is not in-place.
- **Binary search iterations** for n elements: at most $\lfloor \log_2 n \rfloor + 1$. Do not forget the $+1$.
- **Lower bound.** For $n!$ outcomes, at least $\lceil \log_2 n! \rceil$ comparisons.
- **Quicksort partition trace.** Know the Lomuto scheme (pivot last) and the Hoare scheme (pivot first, pointers towards each other) by hand. They give different intermediate arrays.

10.6 Practice Problems

Problem 314 *[NAT] Iterative binary search is run on a sorted array of 100 distinct keys. Searching for each of the 100 keys in turn, the number of keys that need **exactly 7** comparisons is _____.*

Problem 315 *[MSQ] Which of the following statements about searching are TRUE?*

(A) Iterative binary search needs $O(1)$ extra space.

(B) Recursive binary search needs $O(\log n)$ stack space in the worst case.

(C) On a sorted array whose elements are all equal to the key, plain binary search succeeds at its first probe.

(D) The worst-case running time of binary search is $O(\log \log n)$.

Problem 316 *[NAT] Standard insertion sort sorts $[3, 8, 2, 7]$. The total number of comparisons is _____.*

Problem 317 *[NAT] An array of $n = 10$ distinct numbers has its first 9 elements in ascending order, and the last element is smaller than all of them. In insertion sort the total number of comparisons is _____.*

Problem 318 *[NAT] Insertion sort using adjacent swaps is run on a uniformly random permutation of 20 distinct elements. The expected number of swaps is _____.*

Problem 319 *[MCQ] Insertion sort is implemented on a **singly linked list**. Nodes are removed one by one from the input list and inserted into a growing sorted output list. The position is found by scanning the output list **from its head**, and relinking costs $O(1)$. What are the running times for an input in strictly **descending** order and for an input in strictly **ascending** order?*

(A) $\Theta(n^2)$ and $\Theta(n^2)$

(B) $\Theta(n)$ and $\Theta(n^2)$

(C) $\Theta(n^2)$ and $\Theta(n)$

(D) $\Theta(n)$ and $\Theta(n)$

Problem 320 *[MSQ] Let I be the number of inversions of an array of n distinct elements sorted by standard insertion sort. Which statements are TRUE?*

(A) The number of shifts equals I.

(B) The number of comparisons is at most $I + (n - 1)$.

(C) The number of comparisons equals $I + (n - 1)$ for every input.

(D) If $I = 0$, the number of comparisons is exactly $n - 1$.

Problem 321 *[MSQ] **Binary insertion sort** finds the insertion position by binary search on the sorted prefix, then shifts. Which statements are TRUE?*

(A) The total number of comparisons is $O(n \log n)$.

(B) The worst-case running time is still $\Theta(n^2)$.

(C) The best-case running time is $\Theta(n)$.

(D) It can be made stable by searching for the position just after the last equal element.

Problem 322 *[NAT] Bubble sort with the early-exit flag (stop after a pass with no swap) is run on $X = [2, 3, 4, 5, 1]$ and on $Y = [5, 1, 2, 3, 4]$. Let c_X and c_Y be the total numbers of comparisons. The value of $c_X - c_Y$ is _____.*

Problem 323 *[NAT] Selection sort (find the minimum of the unsorted part, swap it into place only if it is at a different index) sorts the permutation $[3, 7, 1, 5, 2, 6, 4]$. The number of swaps performed is _____.*

Problem 324 *[MCQ] Records (key, tag) are $[(5, a), (5, b), (2, c), (9, d)]$ and are sorted by key using selection sort that swaps the minimum into position (strict $<$ when finding the minimum). The output order of tags is:*

(A) c, a, b, d

(B) c, b, a, d

(C) a, b, c, d

(D) c, a, d, b

Problem 325 ***[NAT]*** *In Lomuto partition the last element is selcted as pivot and is applied once to* $[9, 4, 7, 1, 8, 2, 6, 5]$. *The* `exchange` *statement is executed (including self-exchanges and the final one)* _____ *times.*

Problem 326 ***[MCQ]*** *The Hoare partition (pivot is first element) is applied to* $[7, 2, 9, 4, 1, 8, 3]$. *What index is returned and what is the array afterwards?*

(A) index 3, $[3, 2, 1, 4, 9, 8, 7]$

(B) index 4, $[3, 2, 1, 4, 9, 8, 7]$

(C) index 3, $[3, 2, 1, 4, 7, 8, 9]$

(D) index 2, $[3, 2, 1, 7, 4, 8, 9]$

Problem 327 ***[MCQ]*** *An array of* n *distinct elements is sorted by (i) pivot as last element and (ii) as first element of the array. Their worst case running times are respectively:*

(A) $\Theta(n^2)$ *and* $\Theta(n \log n)$

(B) $\Theta(n \log n)$ *and* $\Theta(n^2)$

(C) $\Theta(n^2)$ *and* $\Theta(n^2)$

(D) $\Theta(n)$ *and* $\Theta(n \log n)$

Problem 328 ***[NAT]*** *Randomised quicksort sorts 6 distinct numbers. The probability that the elements of rank 2 and rank 5 are ever compared with each other during the run is* _____.

Problem 329 ***[NAT]*** *Randomised quicksort (uniform random pivot in every call) sorts 3 distinct elements. The expected number of comparisons, rounded to 2 decimal places, is* _____.

Problem 330 ***[NAT]*** *Merge sort sorts* $[4, 1, 3, 2, 8, 5, 7, 6]$. *A merge stops comparing as soon as one of its two lists is exhausted, and the rest is copied without comparisons. The total number of comparisons is* _____.

Problem 331 ***[NAT]*** *Merge sort splits* n *elements into* $\lceil n/2 \rceil$ *and* $\lfloor n/2 \rfloor$ *parts. A merge of total size* m *takes at most* $m - 1$ *comparisons. For* $n = 100$, *the worst-case number of comparisons is* _____.

Problem 332 ***[MCQ]*** *In a merge sort on records, the merge step is written with* `if L[i] < R[j]: take L[i] else: take R[j]`. *Which statement is TRUE?*

(A) The output is sorted and the sort is stable.

(B) The output is sorted but the sort is not stable.

(C) The output may be unsorted.

(D) The sort remains stable but runs in $\Theta(n^2)$.

Problem 333 ***[MCQ]*** *For every fixed input of* n *distinct elements, randomised quicksort has expected running time:*

(A) $\Theta(n \log n)$

(B) $\Theta(n^2)$ *on sorted input and* $\Theta(n \log n)$ *otherwise*

(C) $\Theta(n)$

(D) $\Theta(n \log n)$ *only when the input is in random order*

Problem 334 ***[MCQ]*** *The probability that the smallest and the largest of* n *distinct elements are compared with each other during randomised quicksort is:*

(A) $\dfrac{1}{n}$

(B) $\dfrac{2}{n}$

(C) $\dfrac{2}{n-1}$

(D) $\frac{1}{2}$

Problem 335 ***[MCQ]*** *Randomised quicksort and "fixed pivot quicksort on a uniformly random permutation" both have* $\Theta(n \log n)$ *time. The main difference is:*

(A) Randomised quicksort's guarantee holds for every input, while the other depends on the input being random.

(B) Randomised quicksort has $\Theta(n \log n)$ *worst case, the other has* $\Theta(n^2)$ *worst case on random inputs.*

(C) Randomised quicksort is stable, the other is not.

(D) There is no difference.

gatexam.in

10.7 GATE PYQs

GATEPYQ 246 (GATE DA 2026) *Consider that the quick sort algorithm is used to sort an array of n distinct randomly ordered elements. In every call, the pivot is chosen as the first element of the current subarray.*
Let $T(n)$ denote the expected time to sort the array. Assume that the time to partition is linear in the size of the current subarray.
Which of the following recurrence relations correctly represents $T(n)$ in this scenario?

A. $T(n) = T(1) + T(n-1) + O(n)$

B. $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$

C. $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

D. $T(n) = \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-k-1)] + O(n)$

GATEPYQ 247 (GATE DA 2026) *Let A be a sorted array containing 1000 distinct integers. You perform a recursive binary search on A to find an element y. Suppose each comparison checks whether the middle element computed during the current recursive step is equal to, less than, or greater than y.*
The maximum number of comparisons that may have to be performed if y is not an element of A is _____. (Answer in integer)

GATEPYQ 248 (GATE DA 2026) *Consider the problem of sorting the given array in ascending order:*

$P = [1, 2, 3, 5, 4]$

Consider two sorting algorithms Bubble Sort (BS) and Insertion Sort (IS).
Let N_1 be the total number of comparisons done by BS on the elements of P and N_2 be the total number of comparisons done by IS on the elements of P.
Which of the following options is/are correct?

A. $N_1 = 10, N_2 = 4$

B. $N_1 > N_2$

C. *IS on P will perform only one swap*

D. *Both BS and IS on P will make at least one unnecessary comparison (i.e., comparing elements that are already in correct order)*

GATEPYQ 249 (GATE DA 2025) *For which of the following inputs does binary search take time $O(\log n)$ in the worst case?*

A. *An array of n integers in any order*

B. *A linked list of n integers in any order*

C. *An array of n integers in increasing order*

D. *A linked list of n integers in increasing order*

GATEPYQ 250 (GATE DA 2025) *Suppose that insertion sort is applied to the array $[1, 3, 5, 7, 9, 11, x, 15, 13]$ and it takes exactly two swaps to sort the array. Select all possible values of x.*

A. *10*

B. *12*

C. *14*

D. *16*

GATEPYQ 251 (GATE DA 2024) *Consider sorting the following array of integers in ascending order using an in-place Quicksort algorithm that uses the last element as the pivot:*

60 70 80 90 100

The minimum number of swaps performed during this Quicksort is _____.

GATEPYQ 252 (GATE CSE 2026 SET-2) *Consider an array $A = [10, 7, 8, 19, 41, 35, 25, 31]$. Suppose the merge sort algorithm is executed on array A to sort it in increasing order. The merge sort algorithm will carry out a total of 7 merge operations.*

A merge operation on sorted left array L and sorted right array R is said to be void if the output of the merge operation is the elements of array L followed by the elements of array R.

The number of void merge operations among these 7 merge operations is _____.

GATEPYQ 253 (GATE CSE 2021 SET-1) *Consider the following array:*

$$23, 32, 45, 69, 72, 73, 89, 97$$

Which algorithm out of the following options uses the least number of comparisons (among the array elements) to sort the above array in ascending order?

A. *Selection sort*

B. *Merge sort*

C. *Insertion sort*

D. *Quicksort using the last element as pivot*

GATEPYQ 254 (GATE CSE 2019) *An array of 25 distinct elements is to be sorted using quicksort. Assume that the pivot element is chosen uniformly at random. The probability that the pivot element gets placed in the worst possible location in the first round of partitioning (rounded off to 2 decimal places) is _____.*

GATEPYQ 255 (GATE CSE 2016 SET-2) *Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE?*

I. *Quicksort runs in $\Theta(n^2)$ time*

II. *Bubblesort runs in $\Theta(n^2)$ time*

III. *Merge sort runs in $\Theta(n)$ time*

IV. *Insertion sort runs in $\Theta(n)$ time*

A. *I and II only*

B. *I and III only*

C. *II and IV only*

D. *I and IV only*

GATEPYQ 256 (GATE CSE 2016 SET-1) *The worst-case running times of Insertion sort, Mergesort, and Quicksort, respectively, are:*

A. *$\Theta(n \log n)$, $\Theta(n \log n)$, and $\Theta(n^2)$*

B. *$\Theta(n^2)$, $\Theta(n^2)$, and $\Theta(n \log n)$*

C. *$\Theta(n^2)$, $\Theta(n \log n)$, and $\Theta(n \log n)$*

D. *$\Theta(n^2)$, $\Theta(n \log n)$, and $\Theta(n^2)$*

GATEPYQ 257 (GATE CSE 2015 SET-3) *Assume that a mergesort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?*

A. *256*

B. *512*

C. *1024*

D. *2048*

GATEPYQ 258 (GATE CSE 2015 SET-2) *Suppose you are provided with the following function declaration in the C programming language:*

```
int partition(int a[], int n);
```

The function treats the first element of `a[]` *as a pivot, and rearranges the array so that all elements less than or equal to the pivot are in the left part of the array, and all elements greater than the pivot are in the right part. In addition, it moves the pivot so that the pivot is the last element of the left part. The return value is the number of elements in the left part.*
The following partially given function in the C programming language is used to find the kth *smallest element in an array* `a[]` *of size* n *using the* `partition` *function. We assume* $k \leq n$*:*

```
int kth_smallest(int a[], int n, int k)
{
    int left_end = partition(a, n);
    if (left_end + 1 == k) {
        return a[left_end];
    }
    if (left_end + 1 > k) {
        return kth_smallest(________);
    } else {
        return kth_smallest(________);
    }
}
```

The missing argument lists are respectively:

A. `(a, left_end, k)` *and* `(a + left_end + 1, n - left_end - 1, k - left_end - 1)`

B. `(a, left_end, k)` *and* `(a, n - left_end - 1, k - left_end - 1)`

C. `(a + left_end + 1, n - left_end - 1, k - left_end - 1)` *and* `(a, left_end, k)`

D. `(a, n - left_end - 1, k - left_end - 1)` *and* `(a, left_end, k)`

GATEPYQ 259 (GATE CSE 2015 SET-1) *Which one of the following is the recurrence equation for the worst-case time complexity of the Quicksort algorithm for sorting* n (≥ 2) *numbers? (In the recurrence equations below,* c *is a constant.)*

A. $T(n) = 2T(n/2) + cn$

B. $T(n) = T(n-1) + T(1) + cn$

C. $T(n) = 2T(n-1) + cn$

D. $T(n) = T(n/2) + cn$

GATEPYQ 260 (GATE CSE) *You have an array of* n *elements. Suppose you implement quick sort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst-case performance is*

A. $O(n^2)$

B. $O(n \log n)$

C. $\Theta(n \log n)$

D. $O(n^3)$

GATEPYQ 261 (GATE CSE 2014 SET-1) *Let* P *be a quick sort program to sort numbers in ascending order using the first element as the pivot. Let* t_1 *and* t_2 *be the number of comparisons made by* P *for the inputs* [1 2 3 4 5] *and* [4 1 5 3 2] *respectively. Which one of the following holds?*

A. $t_1 = 5$

B. $t_1 < t_2$

C. $t_1 > t_2$

D. $t_1 = t_2$

GATEPYQ 262 (GATE CSE) *Which one of the following is the tightest upper bound that represents the number of swaps required to sort* n *numbers using selection sort?*

A. $O(\log n)$

B. $O(n)$

C. $O(n \log n)$

D. $O(n^2)$

GATEPYQ 263 (GATE CSE 2012) *A list of n strings, each of length n, is sorted into lexicographic order using the merge-sort algorithm. The worst-case running time of this computation is*

A. $O(n \log n)$

B. $O(n^2 \log n)$

C. $O(n^2 + \log n)$

D. $O(n^2)$

GATEPYQ 264 (GATE CSE 2009) *In quick sort, for sorting n elements, the $(n/4)^{th}$ smallest element is selected as pivot using an $O(n)$ time algorithm. What is the worst-case time complexity of the quick sort?*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(n^2 \log n)$

GATEPYQ 265 (GATE CSE 2009) *What is the number of swaps required to sort n elements using selection sort, in the worst case?*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(n^2 \log n)$

GATEPYQ 266 (GATE CSE 2008) *Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let $T(n)$ be the number of comparisons required to sort n elements. Then*

A. $T(n) \leq 2T(n/5) + n$

B. $T(n) \leq T(n/5) + T(4n/5) + n$

C. $T(n) \leq 2T(4n/5) + n$

D. $T(n) \leq 2T(n/2) + n$

GATEPYQ 267 (GATE CSE 2007) *Which of the following sorting algorithms has the lowest worst-case complexity?*

A. *Merge sort*

B. *Bubble sort*

C. *Quick sort*

D. *Selection sort*

GATEPYQ 268 (GATE CSE 2006) *The median of n elements can be found in $O(n)$ time. Which one of the following is correct about the complexity of quick sort, in which median is selected as pivot?*

A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(n^3)$

GATEPYQ 269 (GATE CSE 2006) *Which one of the following in-place sorting algorithms needs the minimum number of swaps?*

A. *Quick-sort*

B. *Insertion sort*

C. *Selection sort*

D. *Heap sort*

GATEPYQ 270 (GATE CSE 2004) *The tightest lower bound on the number of comparisons, in the worst case, for comparison-based sorting is of the order of*

A. n

B. n^2

C. $n \log n$

D. $n \log_2 n$

GATEPYQ 271 (GATE CSE 2003) *In a permutation $a_1 \dots a_n$ of n distinct integers, an inversion is a pair (a_i, a_j) such that $i < j$ and $a_i > a_j$.*
What would be the worst-case time complexity of the Insertion Sort algorithm, if the inputs are restricted to permutations of $1 \dots n$ with at most n inversions?

A. $\Theta(n^2)$

B. $\Theta(n \log n)$

C. $\Theta(n^{1.5})$

D. $\Theta(n)$

GATEPYQ 272 (GATE CSE 2003) *The usual $\Theta(n^2)$ implementation of Insertion Sort to sort an array uses linear search to identify the position where an element is to be inserted into the already sorted part of the array. If, instead, we use binary search to identify the position, the worst-case running time will*

A. *remain* $\Theta(n^2)$

B. *become* $\Theta(n(\log n)^2)$

C. *become* $\Theta(n \log n)$

D. *become* $\Theta(n)$

GATEPYQ 273 (GATE CSE 1999) *A sorting technique is called stable if*

A. *it takes $O(n \log n)$ time*

B. *it maintains the relative order of occurrence of non-distinct elements*

C. *it uses the divide and conquer paradigm*

D. *it takes $O(n)$ space*

GATEPYQ 274 (GATE CSE 1998) *Give the correct matching for the following pairs:*

(A) $O(\log n)$	*(P) Selection sort*
(B) $O(n)$	*(Q) Insertion sort*
(C) $O(n \log n)$	*(R) Binary search*
(D) $O(n^2)$	*(S) Merge sort*

A. *A-R, B-P, C-Q, D-S*

B. *A-R, B-P, C-S, D-Q*

C. *A-P, B-R, C-S, D-Q*

D. *A-P, B-S, C-R, D-Q*

GATEPYQ 275 (GATE CSE 1996) *Quick-sort is run on two inputs shown below to sort in ascending order taking the first element as pivot:*

(i) $1, 2, 3, \ldots, n$

(ii) $n, n-1, n-2, \ldots, 2, 1$

Let C_1 and C_2 be the number of comparisons made for the inputs (i) and (ii) respectively. Then,

A. $C_1 < C_2$

B. $C_1 = C_2$

C. $C_1 > C_2$

D. we cannot say anything for arbitrary n

GATEPYQ 276 (GATE CSE 2001) *Randomized quicksort is an extension of quicksort where the pivot is chosen randomly. What is the worst-case complexity of sorting n numbers using randomized quicksort?*

A. $O(n)$

B. $O(n \log n)$

C. $O(n^2)$

D. $O(n!)$

gatepaini.in

10.8 YouTube Links and QR Codes

Lec. No	Title	URL	QR Code
76	Binary Search	https://www.youtube.com/watch?v=iT4Xs9PEUyk	
77	Bubble Sort	https://www.youtube.com/watch?v=Vp_V_AZbATY	
78	Selection Sort	https://www.youtube.com/watch?v=9w0sYvkYRRE	
79	Insertion Sort	https://www.youtube.com/watch?v=HKuyblDd924	
80	Merge Sort	https://www.youtube.com/watch?v=jhONpn_tr80	
81	Quick Sort	https://www.youtube.com/watch?v=wGUIaGASWnE	
82	GATE PYQs on Searching and Sorting	https://www.youtube.com/watch?v=pr68CYvXczk	
83	Practice Problems on Searching and Sorting	https://www.youtube.com/watch?v=RBApB9p0Y4k	

Chapter 11

Solutions to Practice Problems

Lec. No	Title	URL	QR Code
16	Practice Problems on Time Complexity	https://www.youtube.com/watch?v=znprYcc-0F4	
23	Practice Problems on Stacks	https://www.youtube.com/watch?v=EWJ0_MKidhU	
28	Practice Problems on Queues	https://www.youtube.com/watch?v=IEQK7NnKGvk	
39	Problem Solving on Linked Lists	https://www.youtube.com/watch?v=da6Qz7zD-aA	
50	Practice Problems on Binary Trees & Binary Search Trees	https://www.youtube.com/watch?v=_jLpDuYBMnA	
58	Practice Problems on Hashing	https://www.youtube.com/watch?v=Yt64C-ZypZI	

Lec. No	Title	URL	QR Code
65	Practice Problems on Basics of Graph Theory	https://www.youtube.com/watch?v=gsO3Ns7u4k4	
70	Practice Problems on DFS, BFS & Topological Sort	https://www.youtube.com/watch?v=Rm3ETHxQ6Po	
75	Practice Problems on Shortest Paths	https://www.youtube.com/watch?v=f4LHhAO-x8k	
83	Practice Problems on Searching and Sorting	https://www.youtube.com/watch?v=RBApB9p0Y4k	

Sample

Chapter 12

Solutions to GATE PYQs

Lec. No	Title	URL	QR Code
17	GATE PYQs on Time Complexity	https://www.youtube.com/watch?v=IVUnUrgzQ00	
22	GATE PYQs on Stacks	https://www.youtube.com/watch?v=4OfuDg2jKzk	
26	GATE PYQs on Queues	https://www.youtube.com/watch?v=oXz9Ol6sVtU	
38	GATE PYQs on Linked Lists	https://www.youtube.com/watch?v=_thOnD3Wjmc	
48	GATE PYQs on Binary Trees	https://www.youtube.com/watch?v=2b9b6m4958E	
49	GATE PYQs on Binary Search Trees	https://www.youtube.com/watch?v=DQNgmVK1-lM	

Lec. No	Title	URL	QR Code
57	GATE PYQs on Hashing	https://www.youtube.com/watch?v=5NSSSQGiAxg	
63	GATE PYQs on Basics of Graph Theory	https://www.youtube.com/watch?v=9yihYWBPZ3U	
69	GATE PYQs on DFS, BFS & Topological Sort	https://www.youtube.com/watch?v=yWjl9meVE3k	
74	GATE PYQs on Shortest Paths	https://www.youtube.com/watch?v=_GTk657uO6I	
82	GATE PYQs on Searching and Sorting	https://www.youtube.com/watch?v=pr68CYvXczk	

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.

[2] N. Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, 2nd ed., CareerMonk Publications, 2012.

[3] S. S. Skiena, *The Algorithm Design Manual*, 3rd ed., Springer, 2020.

[4] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Python*, Wiley, 2013.

[5] B. N. Miller and D. L. Ranum, *Problem Solving with Algorithms and Data Structures Using Python*, 2nd ed., Franklin, Beedle & Associates, 2011.

[6] A. Bhargava, *Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People*, 2nd ed., Manning Publications, 2024.

gateaim.in

GateXAIML

Free GATE resources for Data Science & AI and CSE

Website: www.gatexaiml.in — **Email:** contact@gatexaiml.in