LINEAR SEARCH USING RECURSION

Linear search is a straightforward algorithm that checks each element in a list sequentially until the target element is found or the list ends. This method can be implemented recursively, where the function calls itself with a reduced portion of the list until the base condition is met.

Advantages:

Efficient: O(log n) time complexity.

Simple Code: Easy to implement using recursion.

No Extra Space: Doesn’t require extra space besides recursion stack.

Disadvantages:

Recursion Overhead: More memory usage due to recursive calls.

Stack Overflow: Deep recursion can cause stack overflow in large datasets.

Requires Sorted Data.

Time Complexity:

Best Case: O(1)

Worst Case: O(log n)

Applications:

Searching Sorted Arrays: Ideal for efficient searching in sorted data.

Database Indexing: Used in databases where data is pre-sorted and needs quick lookups.

Range Searching: Often used to find the boundaries of ranges in sorted data.

Efficient Lookup: Used in scenarios where fast lookups in sorted data are essential, such as in searching algorithms.

LINEAR SEARCH

Linear search is a straightforward search algorithm used to find the position of a target element within a list or array. It involves sequentially checking each element, starting from the first, until the desired element is found or the end of the list is reached.

Steps to Perform Linear Search

Start at the First Element: Begin with the first element of the list.

Compare the Element: Check if the current element matches the target value.

Move to the Next Element: If it doesn't match, move to the next element.

Repeat: Continue steps 2 and 3 until the target is found or all elements have been checked.

Return the Result: If the target is found, return its position; otherwise, indicate that it is not present in the list.

Advantages:

Simplicity: Easy to understand and implement.

No Sorting Required: Can be used on unsorted data.

Works on Any Data Structure: Can be applied to arrays, linked lists, or any collection.

Disadvantages:

Inefficient for Large Data: Time complexity of O(n) makes it slow for large datasets.

Not Optimal: Other algorithms like binary search are faster on sorted data.

No Early Exit: In the worst case, it checks every element before finding the target.

Time Complexity:

Best Case: O(1)

Worst Case: O(n)

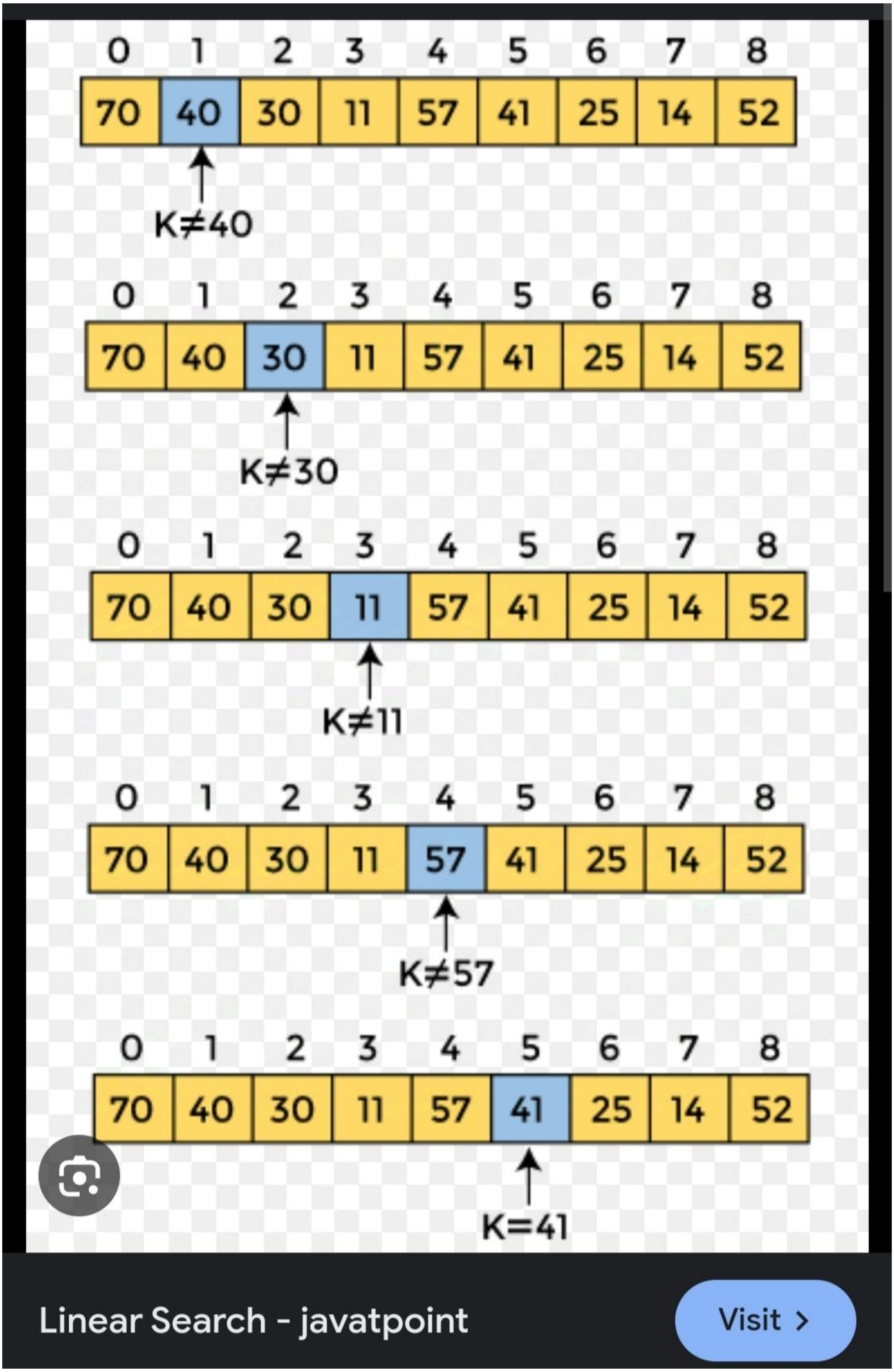
Applications:

Small Data Sets: Works well for small datasets or when the data is unsorted.

Unsorted Data: Used when searching through unsorted data.

Linear Traversal: For simple tasks like searching through a list, array, or linked list.

First Occurrence Search: To find the first occurrence of an element in an array or list



0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K≠40

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K≠30

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K≠11

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K≠57

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K=41

Linear Search - javatpoint

BINARY SEARCH

Binary search is an efficient searching algorithm used to find the position of a target element within a sorted list or array. It works by repeatedly dividing the search interval in half and eliminating one half of the list based on comparisons.

Steps to Perform Binary Search

Start with Sorted Data: Binary search only works on sorted arrays.

Identify Middle Element: Calculate the index of the middle element.

Compare Target with Middle:

If the target is equal to the middle element, return its position.

If the target is less than the middle element, search the left half of the array.

If the target is greater than the middle element, search the right half of the array.

Repeat: Continue dividing the search interval until the target is found or the interval is empty.

Return Result: If the target is found, return its position; otherwise, indicate it is not present.

Advantages:

Efficient: O(log n) time complexity, fast for large sorted datasets.

Minimal Comparisons: Fewer comparisons than linear search.

Disadvantages:

Requires Sorted Data.

Not Suitable for Linked Lists (non-sequential access).

More Complex Implementation.

Time Complexity:

Best Case: O(1)

Worst Case: O(log n)

Applications:

Searching in Sorted Arrays: Ideal for searching in sorted data structures.

Database Indexing: Used in databases where data is sorted for quick access.

Finding Boundaries: Often used in problems that require finding upper or lower boundaries (e.g., range searching).

Efficient Lookup: Used in situations where efficient lookups of sorted data are required.

BINARY SEARCH USING RECURSION

Binary search is an efficient algorithm for finding a target value within a sorted array or list by repeatedly dividing the search interval in half. Implementing binary search recursively involves the function calling itself with updated parameters until the target is found or the search interval is empty.

Advantages:

Efficient: O(log n) time complexity.

Simple to Implement: Easy to code with recursion.

No Extra Space: Only uses recursion stack.

Disadvantages:

Recursion Overhead: Increases memory usage.

Requires Sorted Data.

Stack Overflow: Deep recursion can cause overflow in large datasets.

Time Complexity:

Best Case: O(1)

Worst Case: O(log n)

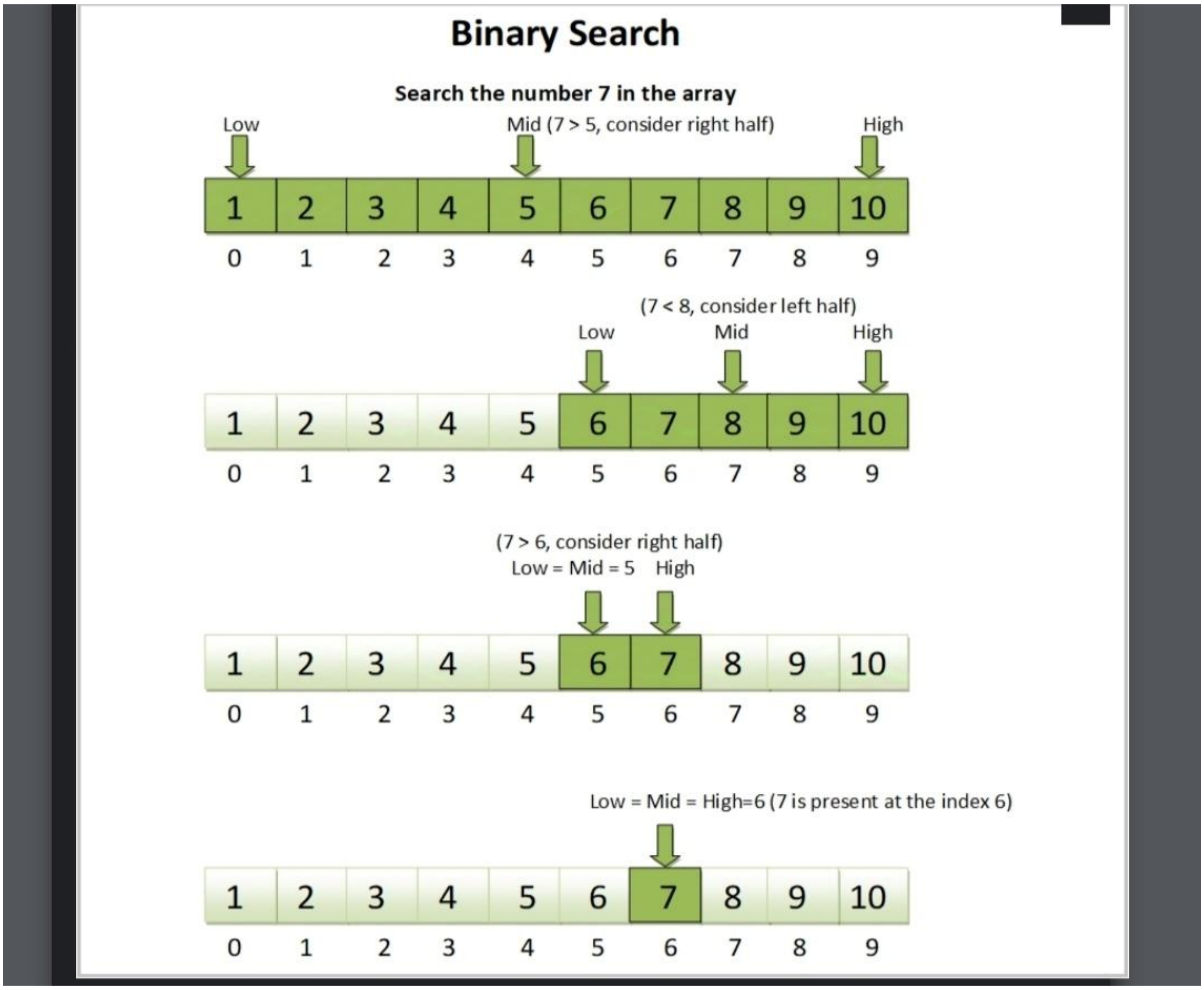
Applications:

Searching Sorted Arrays: Ideal for efficient searching in sorted data.

Database Indexing: Used in databases where data is sorted and requires fast lookups.

Range Searching: Useful for finding specific ranges or boundaries in sorted data.

Efficient Lookups: Used in various algorithms that require quick searching in sorted datasets, such as in dictionaries or search engines.



Binary Search

Search the number 7 in the array

Low

Mid (7 > 5, consider right half)

High

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

(7 < 8, consider left half)

Low

Mid

High

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

(7 > 6, consider right half)

Low = Mid = 5

High

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Low = Mid = High=6 (7 is present at the index 6)

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

QUICK SORT

Quick Sort is a **divide-and-conquer** sorting algorithm that selects a **pivot** element, partitions the array around the pivot, and recursively sorts the sub-arrays. It is widely used due to its efficiency and in-place sorting capabilities.

ALGO:

Divide: Select a 'pivot' element from the array. The choice of pivot can vary; common strategies include picking the first element, the last element, the middle element, or a random element.

Partition: Rearrange the array elements so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. This step is known as partitioning.

Conquer: Recursively apply the above steps to the sub-arrays formed by dividing at the pivot. The base case of the recursion is arrays of size zero or one, which are inherently sorted.

Advantages:

Efficient: Average time complexity O(n log n).

In-place Sorting: O(log n) extra space.

Cache Efficient: Works well with modern CPUs.

Disadvantages:

Worst-Case $O(n^2)$: When pivot is poorly chosen.

Not Stable: Doesn’t preserve relative order of equal elements.

Recursive Overhead: Can cause stack overflow for large datasets.

Time Complexity:

Best/Average Case: O(n log n)

Worst Case: $O(n^2)$

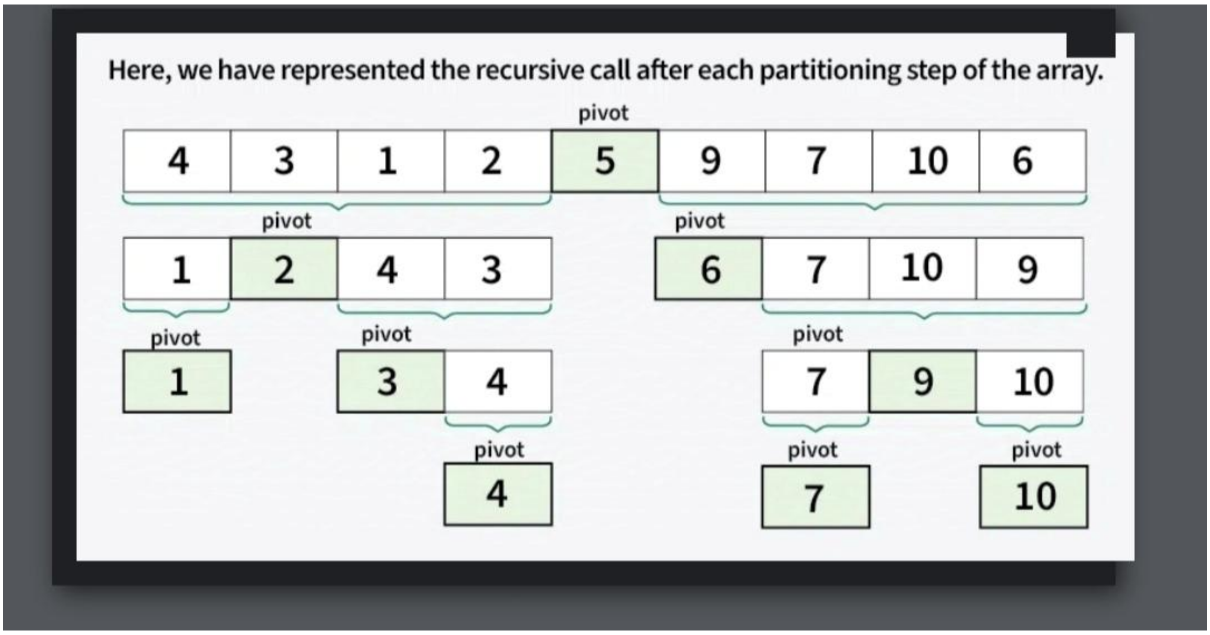
Applications:

General-Purpose Sorting.

Large Datasets.

Database Indexing.

Parallel Computing.



Here, we have represented the recursive call after each partitioning step of the array.

MERGE SORT

Merge Sort is a **divide-and-conquer** sorting algorithm that splits the data into smaller parts, sorts them, and then merges the sorted parts back together. It is efficient for large datasets and provides stable sorting.

ALGORITHM:-

Divide: Split the unsorted list into halves until each sublist contains a single element.

Conquer: Merge the sublists back together in sorted order. This is the recursive part where two sorted sublists are merged.

Combine: Continue merging the sublists until the entire list is reassembled in sorted order.

Advantages:-

Efficient: Time complexity of O(n log n) in all cases.

Stable: Preserves the relative order of equal elements.

Works well for large datasets: Suitable for external sorting (large data that doesn't fit in memory).

Parallelizable: Can be easily parallelized due to its divide-and-conquer nature.

Disadvantages

Space Complexity: Requires O(n) extra space for temporary arrays.

Overhead: Recursive calls can add overhead and consume more memory.

Not Adaptive: Doesn't perform better with partially sorted data.

Slower for Small Lists: Less efficient than simpler algorithms like insertion sort for small datasets.

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Applications

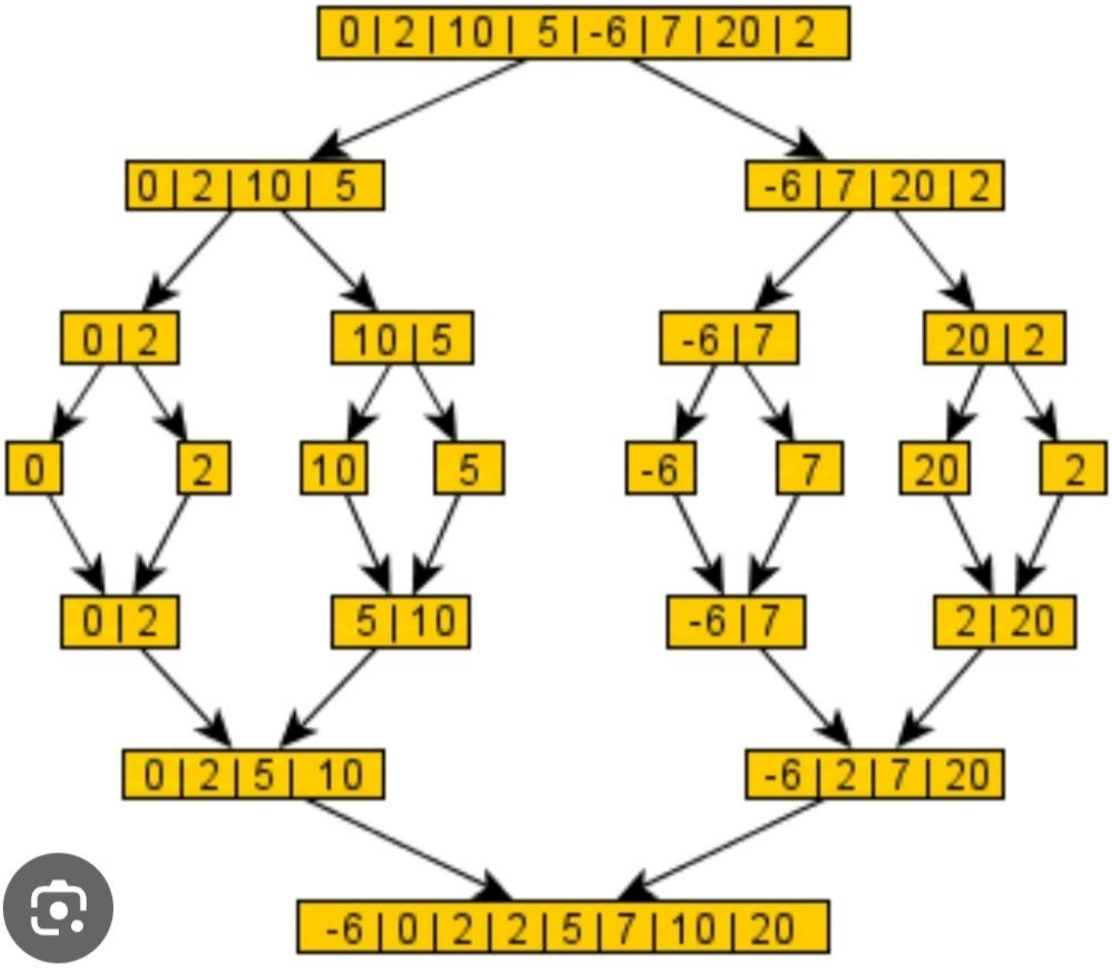
Large Data Sorting: For sorting data that doesn't fit into memory.

Linked List Sorting: Effective for sorting linked lists.

Parallel Processing: Can be parallelized for distributed systems.

Stable Sorting: Used when preserving the relative order of equal elements is needed.

Data Stream Processing: Useful in sorting data streams.


0 | 2 | 10 | 5 | -6 | 7 | 20 | 2
0 | 2 | 10 | 5
-6 | 7 | 20 | 2
0 | 2
10 | 5
-6 | 7
20 | 2
0
2
10
5
-6
7
20
2
0 | 2
5 | 10
-6 | 7
2 | 20
0 | 2 | 5 | 10
-6 | 2 | 7 | 20
-6 | 0 | 2 | 2 | 5 | 7 | 10 | 20

0/1 KNAPSACK PROBLEM

The 0/1 Knapsack Problem is a combinatorial optimization problem where you need to select items to maximize the total value without exceeding the weight capacity of the knapsack. The "0/1" indicates that you can either include or exclude an item (no fractional inclusion).

PROBLEM STATEMENT

The **0/1 Knapsack Problem** is a **combinatorial optimization problem** in which you are given:

- **n items**, each with:
  - w[i] → **weight** of the i^th item
  - v[i] → **value** of the i^th item
- A **knapsack** with a maximum weight capacity of W.

Advantages:-

Optimal Solution: Guarantees the best possible solution using Dynamic Programming (DP).

Real-World Applications: Used in resource allocation, cargo loading, and investment planning.

Efficient for Small Datasets: Works efficiently for small to medium-sized datasets.

Disadvantages

No Fractional Selection: Cannot include partial items, unlike the Fractional Knapsack.

Not Suitable for Continuous Optimization: Limited to discrete problems, unsuitable for continuous optimization.

Memory Overhead: Requires extra memory for the DP table, which can be inefficient for large datasets.

Time Complexity:

Time Complexity: $O(n \times W)$

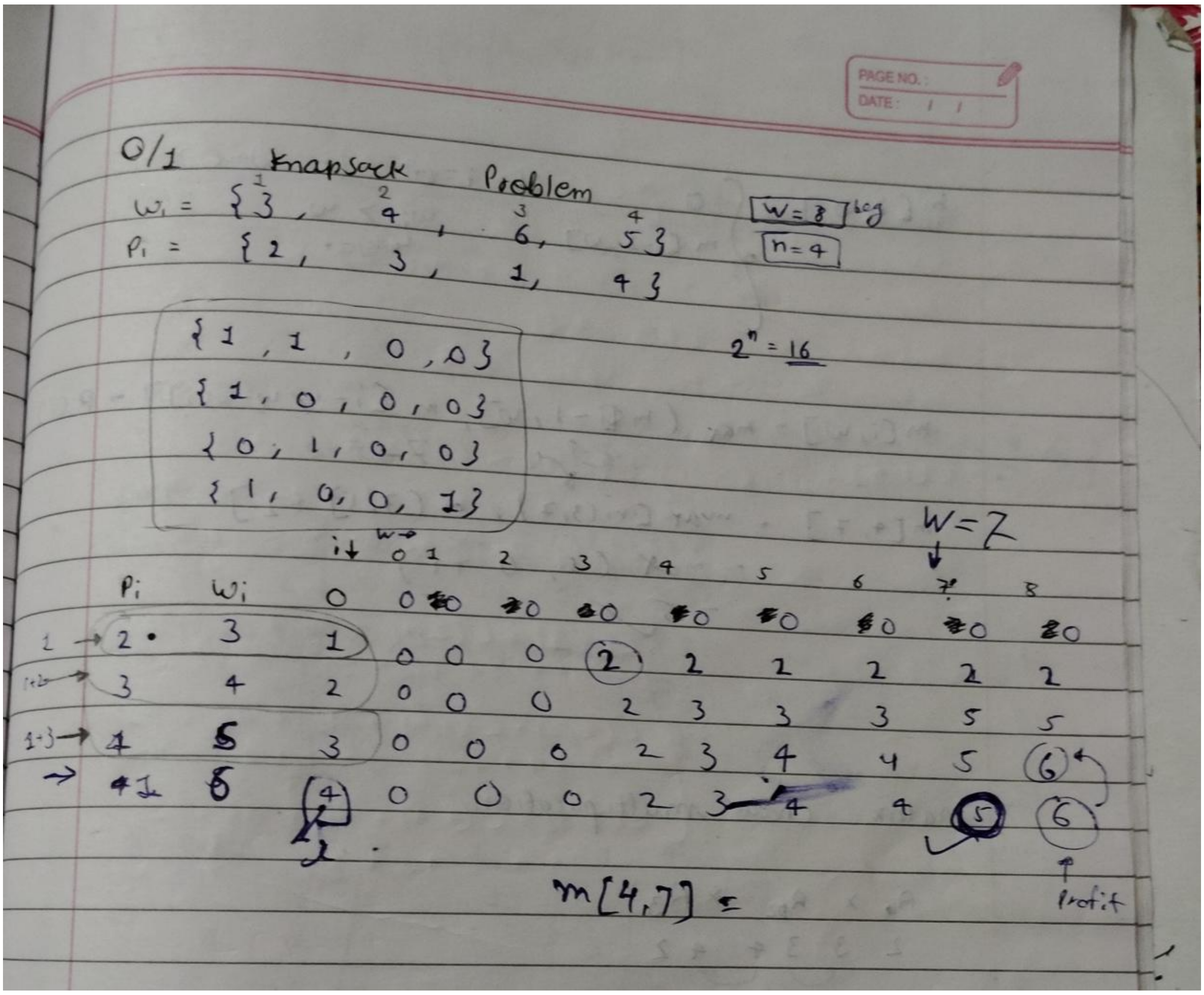
Space Complexity: $O(n \times W)$

Applications

Resource Allocation: Optimal selection of resources with limited capacity.

Cargo Loading: Maximizing the value of goods transported with a weight limit.

Project Selection: Selecting projects with maximum ROI under budget constraints.



0/1 Knapsack Problem

$w_i = \{\overset{1}{3}, \overset{2}{4}, \overset{3}{6}, \overset{4}{5}\}$ $\boxed{W=8}$ kg

$p_i = \{2, 3, 1, 4\}$ $\boxed{n=4}$

$\{1, 1, 0, 0\}$
$\{1, 0, 0, 0\}$
$\{0, 1, 0, 0\}$
$\{1, 0, 0, 1\}$

$2^n = \underline{16}$

W=7

	p_i	w_i	i↓ w→	0	1	2	3	4	5	6	7	8
			0	0	0	0	0	0	0	0	0	0
1 →	2	3	1	0	0	0	2	2	2	2	2	2
1+2 →	3	4	2	0	0	0	2	3	3	3	5	5
1+3 →	4	5	3	0	0	0	2	3	4	4	5	6
→	4 1	6	4	0	0	0	2	3	4	4	5	6

1 .

Profit

m[4,7] =

MATRIX CHAIN MULTIPLICATION

Matrix Chain Multiplication is a dynamic programming algorithm used to find the most efficient way to multiply a sequence of matrices by minimizing the total number of scalar multiplications.

PROBLEM STATEMENT

Given a sequence of matrices A1,A2,A3,…,An with dimensions p0×p1,p1×p2,…,pn−1 , the goal is to find the optimal parenthesization that minimizes the number of multiplications.

For example:

For matrices with dimensions: 10×30, 30×5, 5×60, the optimal order is:
(A1×(A2×A3))
Not:
((A1×A2)×A3)
Since the first order requires **15,000** multiplications, while the second requires **27,000**.

ALGORITHM:-

Initialize the Matrix:

- Create a 2D table m[i][j] to store the minimum number of multiplications needed for matrix chain multiplication between matrix i and matrix j.

Base Case:

- For a single matrix, multiplication cost is zero: m[i][i]=0

Iterate for Increasing Chain Length:

- For L=2 to n:
    - For each matrix chain, determine the minimum multiplication cost.

Choose the Optimal Parenthesization:

- Select the minimum cost by considering different ways to split the matrices.
- Use the formula:
  m[i][j]=min(m[i][k]+m[k+1][j]+pi−1×pk×pj)
- Where k is the splitting point.

Advantages:-

Optimized Matrix Multiplication: Minimizes the number of scalar multiplications.
Efficient for Large Chains: Suitable for large matrix chains.

Reusable Subproblems: Uses dynamic programming to avoid redundant computations.

Disadvantages

Space Complexity: Requires O(n^2) space.
Limited Scope: Only applicable for matrix chain multiplication, not general matrix operations.

Time Complexity:

Time Complexity: O(n^3)

Space Complexity: O(n^2)

Applications

Graphics Processing: Used in computer graphics for transforming objects.
Scientific Computing: Matrix operations are common in simulations and data analysis.
Dynamic Programming Problems: Used to solve complex optimization problems involving

$((A_1(A_2 A_3)) A_4)$

$A_1 \quad * \quad A_2 \quad * \quad A_3 \quad * \quad A_4$

$\underset{d_0}{\underset{\downarrow}{\boxed{3}}}\ \underset{d_1}{\underset{\downarrow}{\boxed{2 \qquad 2}}}\ \underset{d_2}{\underset{\downarrow}{\boxed{4 \qquad 4}}}\ \underset{d_3}{\underset{\downarrow}{\boxed{2 \qquad 2}}}\ \underset{d_4}{\underset{\downarrow}{\boxed{5}}}$

$$C[i, J] = \min_{i \le k < J} \left\{ C(i,k) + C(k+1, j) + d_{i-1} * d_k * d_j \right\}$$

Cost matrix

	1	2	3	4
1	0	24 ①	28 ④	58 ⑥
2		0	16 ②	36 ⑤
3			0	40 ③
4				0

K Table

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

$$C[1,2] = \min_{1 \le 1 < 2}^{k=1} \left\{ C(1,1) + C(2,2) + d_0 * d_1 * d_2 \right\}$$

$$= \{ 0 + 0 + 3 * 2 * 4 \}$$

$$= \underline{\underline{24}}$$

$$C[2,3] = \min_{2 \le 2 \le 3}^{k=2} \left\{ C(2,2) + C(3,3) + d_1 * d_2 * d_3 \right\}$$

$$= \{ 0 + 0 * 2 * 4 * 2$$

$$= 16$$

$$C[3,4] = \min_{3 \le 3 < 4} \overset{k=3}{\{} C(3,3) + C(4,4) + d_2 * d_3 * d_4 \}$$

$$= 0 + 0 + 4 \times 2 \times 5$$

$$= 40$$

minimum

$$C[1,3] = \min_{1 \le k < 3} \begin{cases} k=1 & C(1,1) + C(2,3) + d_0 * d_1 * d_3 \Rightarrow \textcircled{28} \\ k=2 & C(1,2) + C(3,3) + d_0 * d_2 * d_3 \Rightarrow 48 \end{cases}$$

$$C[1,4] = \min_{1 \le k \le 4} \begin{cases} k=1 & C(1,1) + C(2,4) + d_0 * d_1 * d_4 \\ & = 0 + 36 + 3 * 2 * 5 \Rightarrow 66 \\ k=2 & C(1,2) + C(3,4) + d_0 * d_2 * d_4 \\ & = 24 + 40 + 3 * 4 * 5 \Rightarrow 124 \\ k=3 & C(1,3) + C(4,4) + d_0 * d_3 * d_4 \\ & = 28 + 0 + 3 * 2 * 5 \Rightarrow \textcircled{58} \end{cases}$$

min.

$$C[2,4] = \min_{2 \le k < 4} \begin{cases} k=2 & C(2,2) + C(3,4) + d_1 * d_2 * d_4 \\ & = 0 + 40 + 2 * 4 * 5 \Rightarrow 80 \\ k=3 & C(2,3) + C(4,4) + d_1 * d_3 * d_4 \\ & = 16 + 0 + 2 * 4 * 5 \end{cases}$$

$$= \boxed{36}$$

$\underline{k = 3}$

LONGEST COMMON SUBSEQENCE (LCS)

The **Longest Common Subsequence (LCS)** is a classic dynamic programming problem used to find the longest sequence of characters that appear in the same order in both sequences but **not necessarily consecutively**.

EXAMPLE :

String 1: ACDBE
String 2: ABDE
LCS: ABE

Steps to Perform LCS

1. **Initialize a Matrix:**
   Create a 2D matrix dp[m+1][n+1] where m and n are the lengths of the two sequences.
2. **Fill the Matrix:**
   - If characters match → dp[i][j] = dp[i-1][j-1] + 1
   - If characters don't match → dp[i][j] = max(dp[i-1][j], dp[i][j-1])
3. **Backtrack to Find LCS:**
   - Start from the bottom-right of the matrix.
   - Trace back through matching cells to build the LCS.

Advantages:-

Optimal Substructure: The problem can be broken into smaller subproblems, making it ideal for dynamic programming.
Reusability: Used in bioinformatics, text comparison, and data differencing algorithms.
Efficient Storage: Can be optimized to use O(n) space complexity instead of O(m*n).

Disadvantages

Space Complexity: The matrix requires O(m*n) space, which can be large for long sequences.
Not Suitable for Large Data: For very large sequences, LCS can become inefficient.

Time Complexity:

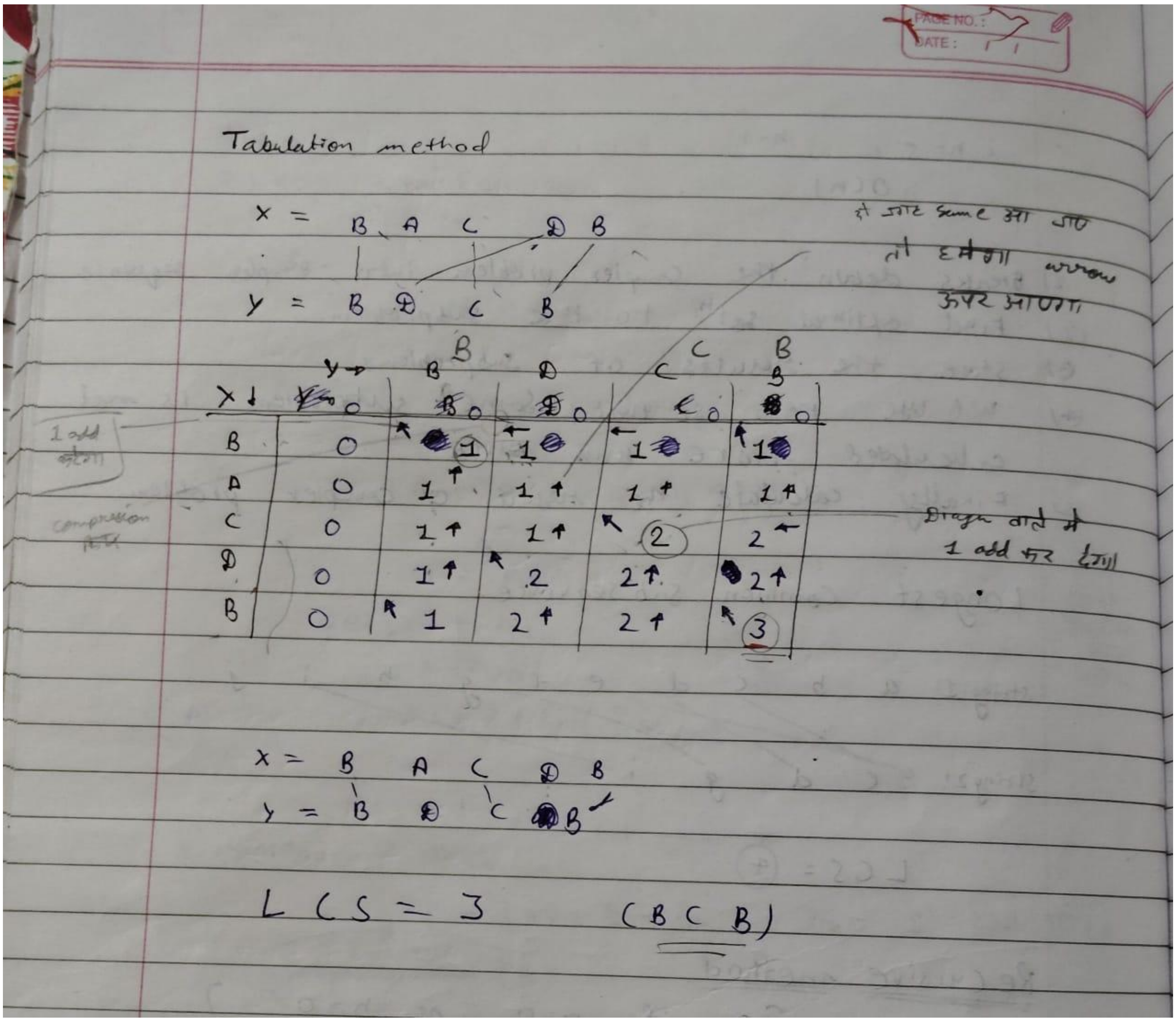
- O(m*n) → where m and n are the lengths of the two sequences.

Space Complexity:

- O(m*n) → for the full matrix.
- O(n) → for optimized space-efficient solution.

Applications

Bioinformatics: DNA and protein sequence alignment.
Text Differencing: Comparing document versions to find differences.
Data Compression: Finding common patterns for compression.
Plagiarism Detection: Identifying common sequences in text.



Tabulation method

x = B A C D B

y = B D C B

दो जगह same आ जाए तो इसमें arrow ऊपर आएगा

X↓	Y→ 0	B 0	D 0	C 0	B 0
B	0	↖ 1	← 1	← 1	↑ 1
A	0	1 ↑	1 ↑	1 ↑	1 ↑
C	0	1 ↑	1 ↑	↖ 2	2 ←
D	0	1 ↑	↖ 2	2 ↑	2 ↑
B	0	↖ 1	2 ↑	2 ↑	↖ 3

1 add करेगा

Diagn वाले मे 1 add कर देगा

x = B A C D B

y = B D C B

LCS = 3 (B C B)

PRIMS ALGO

Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) for a weighted, connected, and undirected graph. The MST connects all vertices with the minimum possible total edge weight without forming any cycles.

Steps to Perform prims algo :

1.Choose a Starting Vertex:

Select any vertex as the starting point.

2. Select the Minimum Edge:
Choose the edge with the smallest weight connecting the current tree to a vertex outside the tree.

3. Add the Vertex and Edge:
Add the vertex and the corresponding edge to the MST.

4. Repeat:
Continue adding the smallest edge that connects a new vertex until all vertices are included.

Advantages:-

Efficient for Dense Graphs: Works well for graphs with many edges.

Simple and Greedy: Selects the smallest edge at every step.

Optimal for MST: Guarantees the minimum spanning tree.

Disadvantages

Inefficient for Sparse Graphs: May perform unnecessary operations on sparse graphs.

Priority Queue Overhead: Requires additional data structures (like min-heap) for efficiency.

Time Complexity:

Using Adjacency Matrix: O(V^2)

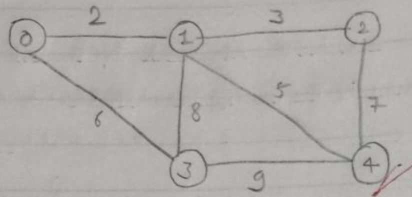
Using Min-Heap and Adjacency List: O(E log V)

Applications

Network Design: Used in laying **cables and roads** efficiently.

Cluster Analysis: In machine learning for **clustering** data.

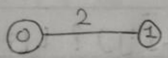
Graph Algorithms: Helps in **approximation algorithms** like Traveling Salesman Problem (TSP).



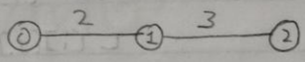
$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 2 & 0 & 6 & 0 \\
1 & 2 & 0 & 3 & 8 & 5 \\
2 & 0 & 3 & 0 & 0 & 7 \\
3 & 6 & 8 & 0 & 0 & 9 \\
4 & 0 & 5 & 7 & 9 & 0
\end{array}$$

Poims Algo to Solve :-

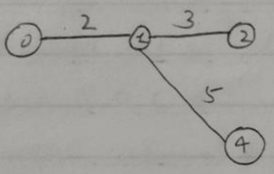
Step-1



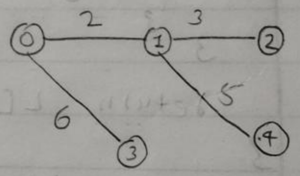
Step-2



Step-3



Step-4



MST = 2+3+6+5 ⇒ 16

KRUSKAL ALGO

Kruskal's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) for a weighted, connected, and undirected graph. It finds the MST by selecting the edges with the smallest weights while avoiding cycles.

Steps to Perform prims algo :

1. Sort All Edges:
Sort the edges of the graph in **ascending order** of their weights.

2. **Select the Smallest Edge:**
Pick the edge with the **smallest weight**.

3. **Check for Cycles:**
Use **Union-Find** (Disjoint Set Union) to check if adding the edge creates a **cycle**.

- **If no cycle:** Add the edge to the MST.
- **If cycle:** Discard the edge.

4.Repeat:
Continue adding edges until the MST contains **(V - 1)** edges, where V is the number of vertices.

Advantages:-

Efficient for Sparse Graphs: Works well for graphs with fewer edges.

Greedy and Optimal: Guarantees the minimum spanning tree.

Simple to Implement: Uses straightforward edge selection and cycle checking.

Disadvantages

Cycle Checking Overhead: Requires Union-Find operations.

Less Efficient for Dense Graphs: Sorting all edges takes more time.

Time Complexity:

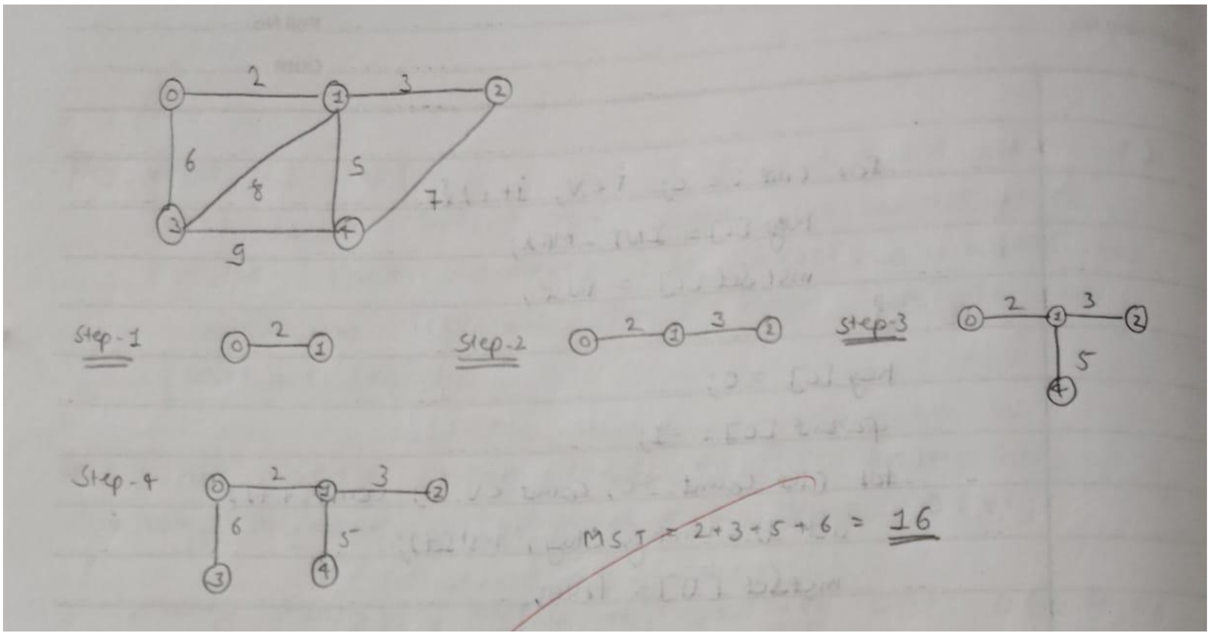
Using Union-Find with Path Compression: O(E log E), where E is the number of edges.

Applications

Network Design: Used in designing cable networks with minimal cost.

Clustering: Helps in data clustering problems.

Graph Analysis: Detecting connected components in a graph.



Step-1

Step-2

Step-3

Step-4

MST = 2+3+5+6 = 16

N QUEEN PROBLEM

The **N-Queen problem** is a **backtracking algorithm** used to place **N queens** on an **N×N chessboard** such that **no two queens attack each other**.

- **Objective:** Place all queens such that:
  - No two queens share the **same row**.
  - No two queens share the **same column**.

Steps to Solve N-Queen Problem

1. **Start with the First Row:**
   Place a queen in the first row and move to the next row.
2. **Check for Safety:**
   For each column in the current row, check if it is safe to place a queen:
   - No queen in the **same column**.
   - No queen on the **left diagonal**.
   - No queen on the **right diagonal**.
3. **Place the Queen:**
   If safe, place the queen and move to the next row.
4. **Backtrack if Necessary:**
   If no valid column is found in a row:
   - **Backtrack** to the previous row.
   - Move the queen to the next possible column.
5. **Repeat Until All Queens are Placed:**
   - Continue placing queens row by row until all queens are placed.
   - Print or store the valid board configuration.

Advantages:-

Demonstrates backtracking effectively.

Useful in AI and CSP problems.

Enhances problem-solving skills.

Disadvantages

Memory-intensive due to recursion.

Limited practical use.

Time Complexity: Worst-case Time Complexity: O(N!)

Space Complexity: O(N^2)) (for board storage)

Applications

Artificial Intelligence: Used in constraint satisfaction problems (CSP).

Combinatorial Optimization: Helps in solving backtracking problems.

Parallel Computing: Used in multi-threaded task scheduling problems.

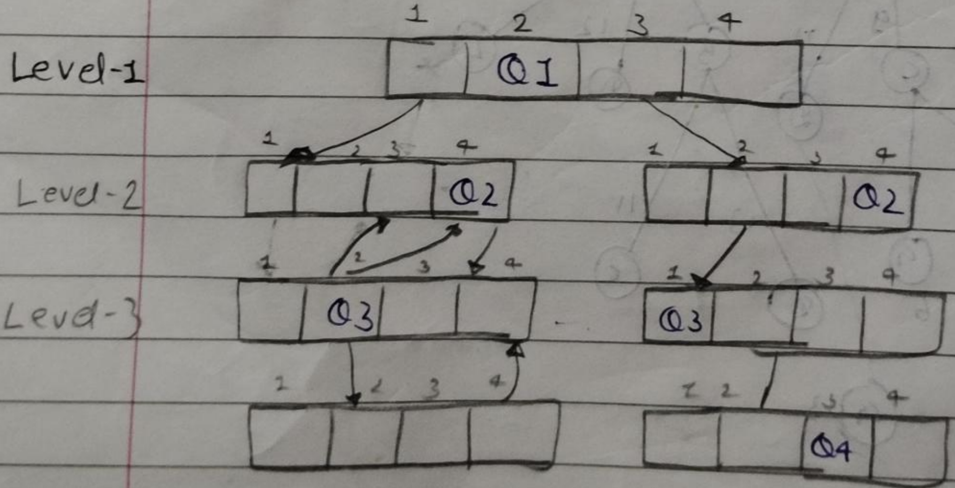
N - Queen Problem.

Q. Q1, Q2, Q3, Q4.

Step.1

Q1			
		Q2	
	Q3		

	Q1		
			Q2
Q3			
		Q4	



RABIN KRAP STRING MATCHING ALGO

The **Rabin-Karp algorithm** is a string matching algorithm used to find a **pattern** in a **text** using **hashing**.

- It compares the **hash value** of the pattern with the hash values of substrings in the text.
- If the hash values match, it performs a **character-by-character** comparison to confirm the match.

Steps to Perform Rabin-Karp Algorithm:

Calculate Hash Values:

- **Compute the hash value of the pattern.**
- **Compute the hash value of the first substring of the same length in the text.**

Compare Hashes:

- **If the hash values match, compare the strings.**
- **If they don't match, slide the window by one character.**
- **Recalculate the hash value for the next substring.**

Repeat:

- **Continue this process until the end of the text.**

Advantages:-

Efficient for multiple pattern matching.

Faster hashing makes it quicker for large datasets.

Disadvantages

Hash collisions can cause false matches.

Inefficient in the worst case.

Time Complexity:

▯ **Best Case: O(n + m)**

▯ **Worst Case: O(n × m) (due to hash collisions)**
Where: n → length of the text , m → length of the pattern

Applications :

Plagiarism detection.

Spam filtering.

Searching large texts.