
🔍 Searching Algorithms

A **searching algorithm** is a method used to **find the location of a specific element (key)** in a collection of data (like an array or list).
It helps determine **whether** the element exists and, if it does, **where** it is located.

🧩 Types of Searching Algorithms

◇ Linear Search

Linear Search is a simple searching algorithm that checks **each element one by one** in a list or array until the desired element (key) is found or the list ends.
It does **not require the array to be sorted**.

Example:
If you have [10, 20, 30, 40] and you search for 30, it will compare 10 → 20 → 30 (found at 3rd position).

```
int linearSearch(int[] arr, int key) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == key) {
            return i; // return index if found
        }
    }
    return -1; // return -1 if not found
}
```

Case	Time Complexity	Explanation
Best Case	**O(1)**	Element found at the first position
Average Case	**O(n)**	Element found somewhere in the middle
Worst Case	**O(n)**	Element found at the end or not found

Space Complexity: O(1) (no extra memory used)

◇ Binary Search

Binary Search is an efficient searching algorithm that works **only on sorted arrays**.
It repeatedly **divides the search range in half** and compares the middle element with the target value until it is found or the range becomes empty.

Example:
For sorted array [10, 20, 30, 40, 50], searching for 40 →
check mid (30) → key > 30 → search right half → find 40.

```java
int binarySearch(int[] arr, int key) {
    int low = 0, high = arr.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == key) {
            return mid; // found
        } else if (arr[mid] < key) {
            low = mid + 1; // search right half
        } else {
            high = mid - 1; // search left half
        }
    }
    return -1; // not found
}
```

Case	Time Complexity	Explanation
Best Case	**O(1)**	Element found at the middle on first check
Average Case	**O($\log_2 n$)**	Search space reduced by half each time
Worst Case	**O($\log_2 n$)**	Element not found after all divisions

Space Complexity:

- **Iterative Version: O(1)**
- **Recursive Version: O(log n)** (due to recursion stack)

Sorting Algorithms

A sorting algorithm is a method used to arrange data in a specific order — either ascending or descending.
Sorting makes searching and data processing faster and easier.

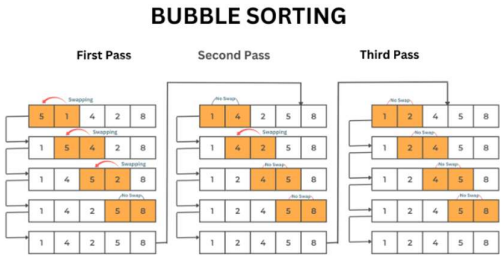
◇ 1. Bubble Sort

Definition:

Bubble Sort repeatedly compares **adjacent elements** and swaps them if they are in the wrong order.
After each pass, the largest element "**bubbles up**" to its correct position.

Algorithm Steps:

```
void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```



Time Complexity:

Case	Complexity	Explanation
Best	**O(n)**	When array already sorted
Average	**O(n^2)**	Typical case
Worst	**O(n^2)**	When array is in reverse order

Space Complexity: O(1)
Stability: ✅ Stable

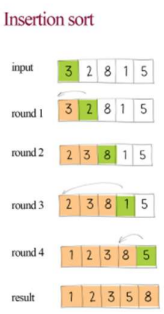
◇ 2. Insertion Sort

Definition:

Insertion Sort builds the final sorted array **one element at a time** by inserting each element into its correct position (like sorting playing cards).

Algorithm Steps:

```java
void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```



Time Complexity:

Case	Complexity	Explanation
Best	**O(n)**	Already sorted
Average	**O(n^2)**	Random order
Worst	**O(n^2)**	Reverse order

Space Complexity: O(1)
Stability: ✅ Stable

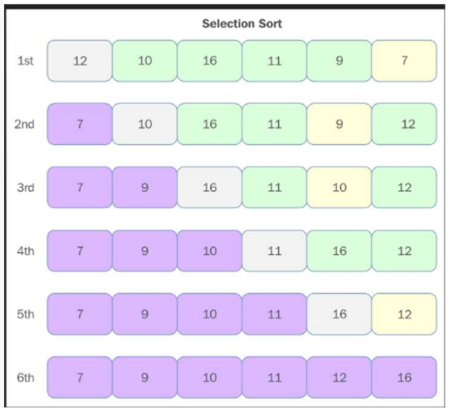
◇ 3. Selection Sort

Definition:

Selection Sort finds the **smallest element** from the unsorted part and **places it at the beginning** in each pass.

Algorithm Steps:

```java
void selectionSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

        // find index of smallest element in remaining array
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // swap arr[i] and arr[minIndex]
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```



Time Complexity:

Case	Complexity	Explanation
Best	**O(n^2)**	Still compares all pairs
Average	**O(n^2)**	Always scans entire array
Worst	**O(n^2)**	Same as average

Space Complexity: O(1)
Stability: ❌ Not stable

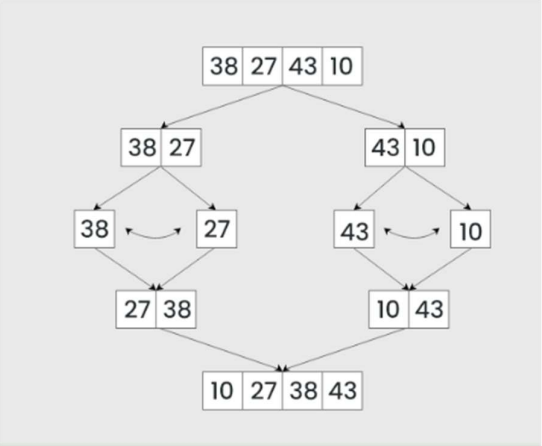
◇ 4. Merge Sort

Definition:

Merge Sort is a **Divide and Conquer** algorithm.
It divides the array into halves, sorts each half, and then merges the sorted halves.

Algorithm Steps:

```java
void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        // sort left half
        mergeSort(arr, left, mid);
        // sort right half
        mergeSort(arr, mid + 1, right);
        // merge both halves
        merge(arr, left, mid, right);
    }
}
```




```java
void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    // copy data to temp arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    // merge the temp arrays
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // copy remaining elements
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Time Complexity:

Case	Complexity	Explanation
Best	**O(n log n)**	Always divides evenly
Average	**O(n log n)**	Consistent

Case	Complexity	Explanation
Worst	**O(n log n)**	Same for all cases

Space Complexity: O(n) (extra space for merging)
Stability: ✅ Stable

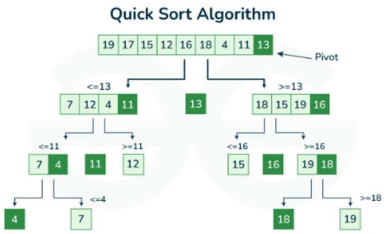
◇ 5. Quick Sort

Definition:

Quick Sort is a **Divide and Conquer** algorithm that selects a **pivot** element, partitions the array so that smaller elements are left of pivot and larger on the right, and then recursively sorts the subarrays.

Algorithm Steps:

```
void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        // recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```




```
int partition(int[] arr, int low, int high) {
    int pivot = arr[high]; // pivot element
    int i = (low - 1);     // smaller element index

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            // swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // swap arr[i+1] and pivot
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
```

Time Complexity:

Case	Complexity	Explanation
Best	**O(n log n)**	Pivot divides array equally

Case	Complexity	Explanation
Average	**O(n log n)**	Expected case
Worst	**O(n^2)**	When pivot is smallest/largest each time

Space Complexity: O(log n) (recursion stack)
Stability: ❌ Not stable

◇ Heap Sort

- Uses a **binary heap** (max-heap).
- Repeatedly removes the largest element and heapifies.
- **Technique:** Comparison-based
- **Time:** Best = Avg = Worst = **O(n log n)**
- **Space:** O(1)
- **Stable:** ❌ No

◇ Counting Sort

- Counts occurrences of each element (works only for integers / limited range).
- **Technique:** Non-comparison
- **Time:** O(n + k) (k = range of numbers)
- **Space:** O(n + k)
- **Stable:** ✅ Yes

◇ Radix Sort

- Sorts numbers **digit by digit** using Counting Sort as sub-routine.
- **Technique:** Non-comparison
- **Time:** O(d × (n + k)) (d = number of digits)
- **Space:** O(n + k)
- **Stable:** ✅ Yes

◇ Bucket Sort

- Divides elements into **buckets**, sorts each bucket (often with Insertion Sort), then combines.
- Works best for **uniformly distributed data**.
- **Technique:** Distribution-based
- **Time:** Best = O(n + k), Avg ≈ O(n + k), Worst = $O(n^2)$
- **Space:** O(n + k)
- **Stable:** ✅ Yes (if inner sort stable)

DC VS DP VS GA

Feature	Dynamic Programming (DP)	Divide and Conquer (D&C)	Greedy Algorithm (GA)
Basic Idea	Break problem into **overlapping subproblems** and **store** results to reuse.	Break problem into **independent subproblems** and **combine** results.	Make **best local choice** at each step hoping for global optimum.
Subproblems	**Overlapping** – same subproblem solved multiple times.	**Independent** – each subproblem unique.	**No subproblems**, only sequence of decisions.
Guarantees Optimal Solution?	✅ Always (if designed correctly).	✅ Always (for correct recursion).	⚠️ Not always (depends on problem).
Optimal Substructure	✅ Yes	✅ Yes	✅ Yes *(but may not always give global optimum)*
Approach	**Bottom-Up / Top-Down** (memoization or tabulation).	**Recursive** divide, then merge results.	**Iterative**, choose local optimum at each step.

Feature	Dynamic Programming (DP)	Divide and Conquer (D&C)	Greedy Algorithm (GA)
Reusability	Results are **stored and reused**.	Results **not reused**.	**No storage**, one-time decisions.
Memory Usage	High (stores many sub-results).	Moderate.	Low.
Time Efficiency	Faster than recursion due to reuse.	Slower if overlapping subproblems.	Usually very fast.
Backtracking Used?	⚠️ **Sometimes** — in path reconstruction only (not core logic).	❌ **No**, purely divide and merge.	❌ **No**, choices are final (no undo).
Examples	Fibonacci, Climbing Stairs, 0/1 Knapsack, LCS.	Merge Sort, Quick Sort, Binary Search.	Coin Change (min coins), Activity Selection, Kruskal's, Prim's.

Divide and conquer (Break → Solve → Combine)

Divide and Conquer is a technique that **breaks a problem into smaller parts**, **solves each part**, and then **combines** the results to get the final solution.

It follows three main steps:

1. **Divide** – Split the main problem into smaller subproblems.
2. **Conquer** – Solve each subproblem (recursively if needed).
3. **Combine** – Merge the results of the subproblems to form the overall solution.

Example 1: Merge Sort

- **Divide:** Split the array into two halves.
- **Conquer:** Recursively sort both halves.
- **Combine:** Merge the two sorted halves into a single sorted array.

Example 2: Binary Search

- **Divide:** Check the middle element of the sorted array.

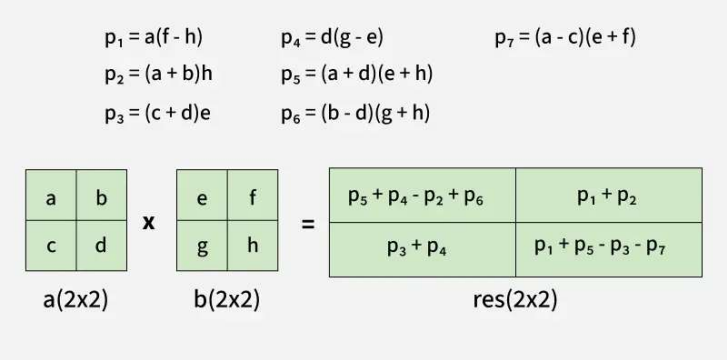
- **Conquer:** If the target is smaller, search the left half; if larger, search the right half.
- **Combine:** The result is found directly when the target equals the middle element.

Example 3: Quick Sort

- **Divide:** Choosing one element as a **pivot** and splitting (partitioning) the array into two parts:
  - Elements **less than** the pivot
  - Elements **greater than** the pivot
- **Conquer:** Recursively sorting both subarrays.
- **Combine:** Combine the sorted subarrays and the pivot into a single sorted array.

Example 3: Strassen's Matrix Multiplication

- tc :$O(n^{2.81})$ (normal matrix multiplication n^3)
- sc :$O(n^2)$ due to creation of intermediate submatrices (normal matrix multiplication same n^2)



- **Divide:** It **divides** the large matrices into 4 smaller submatrices each (n/2 × n/2).
- **Conquer:** It then **recursively** multiplies those smaller submatrices (using 7 multiplications instead of 8).
- **Combine:** it **combines** the results to get the final matrix.

DP

Dynamic Programming is a method for solving problems **efficiently** by **storing solutions** to subproblems and **reusing** them instead of recalculating.

Ingredients of DP:

1. **Optimal Substructure:** Optimal solution of a problem can be built from optimal solutions of its subproblems.
    - *Example:* Shortest path, Knapsack.
2. **Overlapping Subproblems:** Same subproblems are solved multiple times; store results to avoid recomputation.
    - *Example:* Fibonacci, Matrix Chain Multiplication.

Type of DP

⚙️ 1️⃣ **Based on Implementation Method**

a. Top-Down (Memoization)

- You solve the problem **recursively**, starting from the main problem.
- Whenever a subproblem's result is found, you **store (memoize)** it.
- If it's needed again, you **reuse** the stored value.
- Uses **recursion + caching**.

🧩 **Example:** Fibonacci
Instead of recalculating fib(3) again and again, store its value once.

---

b. Bottom-Up (Tabulation)

- You start from the **smallest subproblems** and build your way up.
- Usually implemented **iteratively** using a table (array or matrix).
- No recursion used.

🧩 **Example:** Climbing Stairs
Calculate number of ways to reach step 1, step 2, ... step n, one by one.

Examples

1. Fibonacci Sequence (Simple Recursion Optimization)

Type: *Basic DP / Overlapping Subproblems*
Idea: Each Fibonacci number depends on the previous two.
Example:
To find fib(5) →
fib(5) = fib(4) + fib(3)
But both fib(4) and fib(3) again call smaller values repeatedly.
DP stores previously computed results to avoid repetition.

---

2. Climbing Stairs (Counting Ways)

Type: *DP on Paths / Combinations*
Idea: You can climb either 1 or 2 steps at a time.
Example:
To reach step n, you can come from:

- Step n-1 (then take 1 step), or
- Step n-2 (then take 2 steps)
  So, total ways = ways(n−1) + ways(n−2)
  It's the same relation as Fibonacci — but with a real-world meaning.

---

3. 0/1 Knapsack Problem

Type: *DP on Subsets / Optimization*
Idea: Choose items to maximize value without exceeding capacity.
Example:
You have a bag of capacity 10kg and 4 items with given weights & values.
You must **decide** for each item — take it or leave it — to maximize total value.
DP tries every combination efficiently by storing results for each (item, capacity) pair.

---

4. Binomial Coefficient (nCr)

Type: *DP on Combinatorics*
Idea: Find number of ways to choose r items from n.
Example:
nCr = n-1Cr-1 + n-1Cr
To compute C(5,2), use previous computed smaller values:

C(5,2) = C(4,1) + C(4,2)
DP stores these to avoid recalculating same combinations.

---

5. Matrix Chain Multiplication

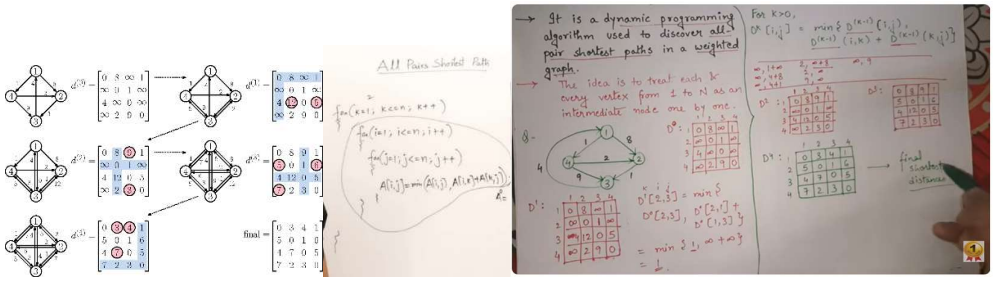
Type: *DP on Intervals / Parenthesization Problems*
Idea: Find the best way to multiply matrices with minimal cost (scalar multiplications).
Example:
You have matrices A, B, C with dimensions that change multiplication cost.
Multiplying ((A×B)×C) may cost more than (A×(B×C)).
DP checks all possible orders, stores sub-results, and finds the minimum cost.

6.Floyd–Warshall Algorithm (Shortest Path Algorithm)

Floyd–Warshall is a **dynamic programming algorithm** used to find the **shortest distance between every pair of vertices** in a graph.

Works on:

- Weighted graph
- Can handle **positive and negative weights**
- **Cannot** handle negative cycles



🧠 Time & Space Complexity

- **Time:** $O(n^3)$
- **Space:** $O(n^2)$

Greedy approach

A problem-solving method where you **choose the best option at each step** (locally optimal) hoping it leads to the **overall best solution** (globally optimal).

1️⃣ Elements of Greedy Strategy

Greedy algorithms build a solution **step by step**, always choosing the **best option at the current step** (locally optimal choice), hoping to reach the global optimum.

Key elements:

1. **Candidate Set:** All possible choices at each step.
2. **Selection Function:** Rule to choose the "best" candidate.
3. **Feasibility Check:** Ensure chosen candidate can be added without violating constraints.
4. **Objective Function:** This defines the **goal of the problem**. What are we trying to: Minimize? (cost, weight, time) Maximize? (profit, value)
5. **Solution Check:** Determines if a complete solution is reached. In Knapsack: when capacity is full.

2️⃣ Local vs Global Optimum

- **Local Optimum:** A **local optimum** is a solution that is **best only within a small nearby region**, but **not necessarily the best overall**.
  - Example: Choosing the shortest edge at each step in Prim's algorithm.
- **Global Optimum:** A **global optimum** is the **best possible solution among all solutions**.
  - Greedy works if **local optimum always leads to global optimum**.

3️⃣ Matroid (Greedy Applicability)

A **matroid** is a structure that guarantees a greedy algorithm finds the **global optimum**.

✅ Greedy algorithm gives optimal solution if the problem is a matroid.

This is called the **Greedy Choice Theorem for Matroids**.

A **matroid** is defined as:

> **M = (S, I)**
> Where:

- **S** = Finite set of elements
- **I** = Collection of independent subsets of S

Matroid Properties:

✅ **1. Non-Empty Property**

The empty set must be independent.

$$\varnothing \in I$$

✅ Meaning: We can always start with nothing.

✅ **2. Hereditary Property (Subset Property)**

Every subset of an independent set is also independent.

$$\text{If } A \in I \text{ and } B \subseteq A, \text{ then } B \in I$$

✅ Meaning: Removing elements never breaks independence.

✅ **3. Exchange Property (Most Important)**

If A and B are two independent sets and A is smaller than B, then:

$$\exists x \in B - A \text{ such that } A \cup \{x\} \in I$$

✅ Meaning: You can always **exchange elements to grow the smaller independent set.**

Examples of Matroid Problems:

- Minimum Spanning Tree (Kruskal's algorithm)
- Activity Selection Problem
- Shortest Path (under certain conditions)

Key Insight:
Greedy works **optimally** on problems that form a **matroid**.

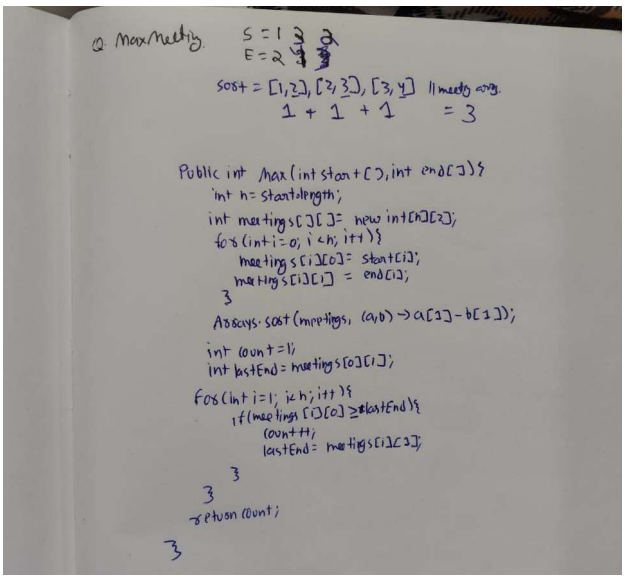
1. Fractional Knapsack (Greedy Algorithm)

- **Type:** Greedy
- **Idea:** You can take **fractions** of items.
- **Approach:**
  - Calculate **value/weight ratio** for each item.
  - Take as much as possible of the item with the **highest ratio** first.
  - Continue until the bag is full.
- **Why Greedy works:**
  - Taking the most valuable weight first always gives the best total because fractions are allowed.
- **Used when:** You can divide the item (like gold, liquids).

🕐 2. Activity Selection (Greedy Algorithm)

- **Type:** Greedy
- **Idea:** Select **maximum number of activities** that don't overlap in time.

- **Approach:**
  - Sort activities by their **finish time**.
  - Always pick the **next activity** that starts after the last selected one finishes.
- **Why Greedy works:**
  - Finishing earliest leaves maximum time for remaining activities.
- **Used when:** Scheduling meetings, CPU tasks, or resource allocation.



🔤 3. Huffman Coding (Greedy Algorithm)

Huffman Coding is a **data compression algorithm** that assigns **shorter codes to more frequent characters** and **longer codes to less frequent characters** — minimizing the total number of bits used

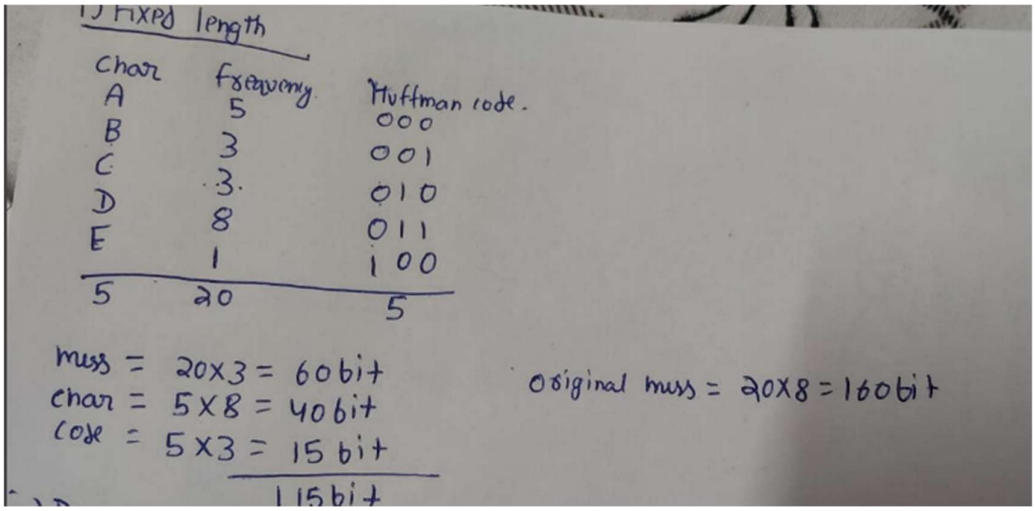
- **Type:** Greedy
- **Idea:** Create the **shortest binary codes** for symbols based on their frequency.
- **Approach:**
  - Always combine the **two least frequent symbols** into a new node.
  - Repeat until one tree remains.
  - Assign shorter codes to frequent symbols.
- **Why Greedy works:**
  - Combining smallest frequencies first ensures minimal total weighted path length.
- **Used in:** Data compression (ZIP, JPEG, MP3).

🧠 Steps (Algorithm)

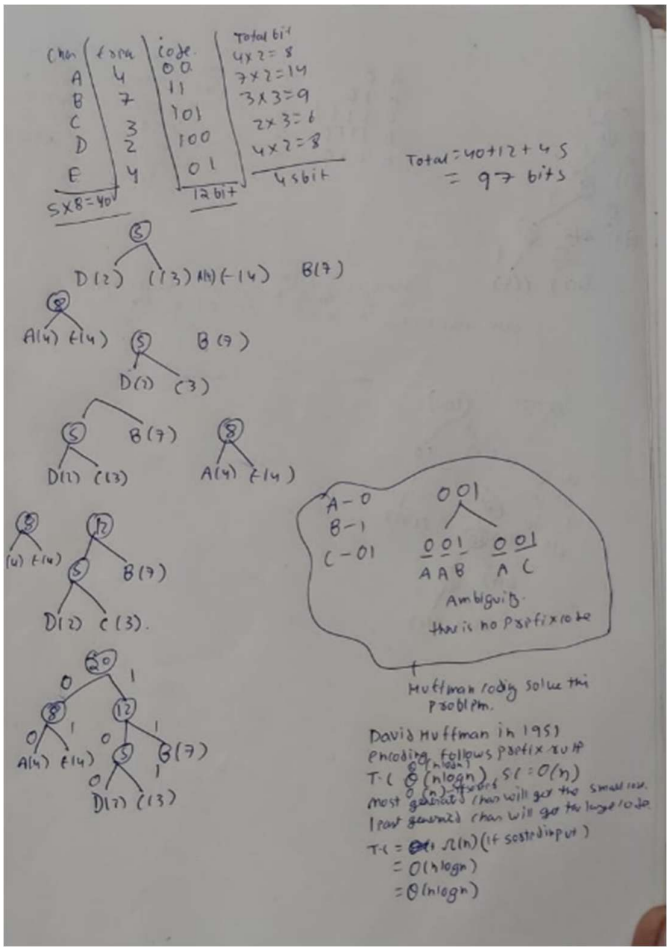
1. **Count frequency** of each character in the data.

2. **Create leaf nodes** for each character and insert them into a **min-heap** (priority queue) based on frequency.
3. **Repeat until one node remains:**
    - Remove **two smallest frequency nodes** from heap.
    - Create a **new internal node** with frequency = sum of both nodes.
    - Set left = first node, right = second node.
    - Insert this new node back into the heap.
4. The remaining node is the **root of the Huffman Tree**.
5. **Assign codes:**
    - Traverse tree:
        - Left edge → add 0
        - Right edge → add 1
    - Codes assigned to leaf nodes form the **Huffman codes**.

Fixed length coding



v**ariable-Length Prefix Coding** (no code is a prefix of another to remove ambiguity)



Tree vs Graph 👇

Feature	Tree 🌳	Graph 🌐
Definition	A special type of graph that is **connected** and **acyclic**	A general data structure of **vertices and edges** that may have cycles
Structure type	**Hierarchical** (parent–child relationship)	**Network-like** (connections between any nodes)
Cycles	❌ No cycles	✅ Can have cycles
Connectivity	Always **connected**	May be **connected** or **disconnected**
Edges count	Exactly **(n – 1)** for *n* nodes	Up to **n(n−1)/2** (undirected) or **n(n−1)** (directed)

Feature	Tree 🌳	Graph 🌐
Root node	Has a single **root** (in directed tree)	No root — any node can connect to any other
Path between nodes	**Exactly one unique path** between any two nodes	**One or more paths** may exist between nodes
Direction	Usually **directed** (like binary tree)	Can be **directed** or **undirected**
Traversal methods	Inorder, Preorder, Postorder, Level order	BFS (Breadth-First Search), DFS (Depth-First Search)
Applications	Hierarchical data (e.g., file systems, XML)	Networks, maps, social graphs, web links
Example	Binary Tree, Family Tree	Road Map, Social Network, Internet Graph

---

✅ **In short:**

A **Tree** is a **special kind of Graph** that is **connected and cycle-free**.
A **Graph** is a **broader structure** that can have **cycles, loops, or disconnections**.

Types of trees

🌳 **1. General Tree**

- The most basic form of tree.
- Each node can have **any number of children**.
- No specific rule on how many children or how they're ordered.
  Example: Company hierarchy, file system.

🌿 **2. Binary Tree**

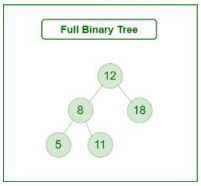
- Each node has **at most two children** — called **left** and **right**.
  Example: Expression trees, binary search trees.

🌱 **3. Binary Search Tree (BST)**

- A **binary tree** with a specific order:
    - Left child < Parent
    - Right child > Parent
- Makes **searching, insertion, and deletion** efficient (O(log n) average).

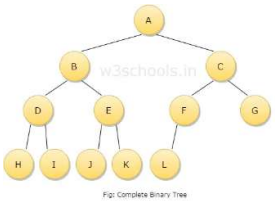
🌾 4. Full Binary Tree

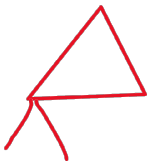
- Every node has **either 0 or 2 children**.
- No node has only one child.



🌴 5. Complete Binary Tree

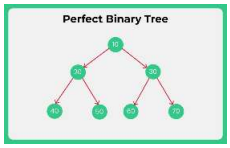
- All levels are **completely filled** except possibly the last one, which is filled **from left to right**.
- **Used in:** Heaps.

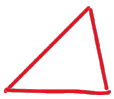




🌻 6. Perfect Binary Tree

- **All internal nodes** have two children and **all leaf nodes** are at the same level.
- Total nodes = 2^h − 1 (where h = height).





🌾 7. Balanced Binary Tree

- The **height difference** between left and right subtrees of every node is **at most 1**.
- Ensures better performance for operations.
- **Examples:** AVL Tree, Red-Black Tree.

🎄 8. AVL Tree

- A **self-balancing binary search tree**.
- Automatically rotates nodes to keep height difference ≤ 1.

🌳 9. Red-Black Tree

- Another **self-balancing BST** with **color properties** (red/black nodes).
- Ensures roughly balanced height for efficient operations.

🌸 10. B-Tree

- A **multi-level balanced search tree** used in **databases and file systems**.
- Each node can have **multiple keys and children**.
- Efficient for disk-based storage.

🌼 11. B+ Tree

- A **variation of B-Tree** where all **data is stored in leaf nodes**.
- Leaf nodes are **linked**, allowing **faster sequential access**.

🎄 12. Trie (Prefix Tree)

- Used to **store strings** by characters.
- Each path from root to leaf represents a **word**.
- **Used in:** Auto-complete, spell checkers.

🌐 Types of Graphs

🔹 **1. Undirected Graph**

- Edges **don't have direction**.
- If A–B is an edge, you can travel both ways (A → B and B → A).
- **Example:** Friendship network (if A is friend of B, B is friend of A).

🔹 **2. Directed Graph (Digraph)**

- Edges have a **specific direction** (arrows).
- If A → B exists, it doesn't mean B → A.
- **Example:** Instagram following (A follows B ≠ B follows A).

🔹 **3. Weighted Graph**

- Each edge has a **weight (cost, distance, or time)**.
- **Example:** Road map where edges represent distance or travel time.

🔹 **4. Unweighted Graph**

- All edges are considered to have **equal weight**.
- **Example:** Simple social network (each connection counts equally).

🔹 **5. Cyclic Graph**

- Contains **at least one cycle** (a path that starts and ends at the same vertex).
- **Example:** Round-trip routes in transportation networks.

🔹 **6. Acyclic Graph**

- **No cycles** present.
- **Example:** Tree or dependency graph.

🔹 **7. Connected Graph**

- There's a **path between every pair of vertices**.
- **Example:** All computers in a local network are connected.

🔹 **8. Disconnected Graph**

- Some vertices **cannot be reached** from others.
- **Example:** Two separate social groups with no mutual friends.

🔹 **9. Complete Graph**

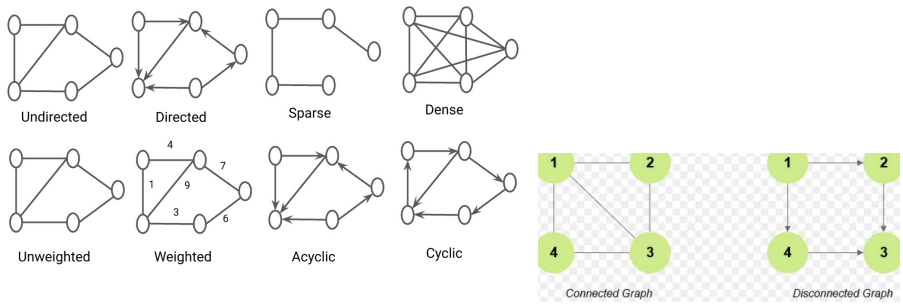
- Every vertex is **connected to every other vertex**.
- For *n* vertices → edges = n(n−1)/2.
- **Example:** Every person knows everyone else in the group.

🔹 **10. Sparse Graph**

- Has **few edges** compared to the number of possible edges.
- **Example:** Road network between cities (not every city connects directly).

🔹 **11. Dense Graph**

- Has **many edges** (close to complete).
- **Example:** Small team where everyone collaborates with everyone.



Optimal Binary Search Tree

1. Normal BST problem

- In a normal BST, keys are just inserted in some order.
- Example: Insert keys 10, 20, 30 → Tree becomes skewed (like a linked list).
- Searching takes longer if frequently used keys are deep in the tree.
- **Average search cost is not optimized.**

2. Optimal BST solution

- OBST considers **frequency of access** for each key.
- Frequently accessed keys are placed **closer to the root**.
- Rarely accessed keys are placed **lower**.
- This **minimizes the total search cost**.

An **Optimal Binary Search Tree (OBST)** is a **binary search tree (BST)** that is structured to **minimize the expected search cost** based on the probabilities of searching for each key.

- In a normal BST, all keys have equal probability, but in OBST, each key may have **different search frequencies**.
- The **goal** is to build a BST such that **frequently accessed keys are near the root** to minimize the average search cost.
- **TC:** $O(n^3)$(or $O(n^2)$ With Knuth optimization)
- **SC:** $O(n^2)$

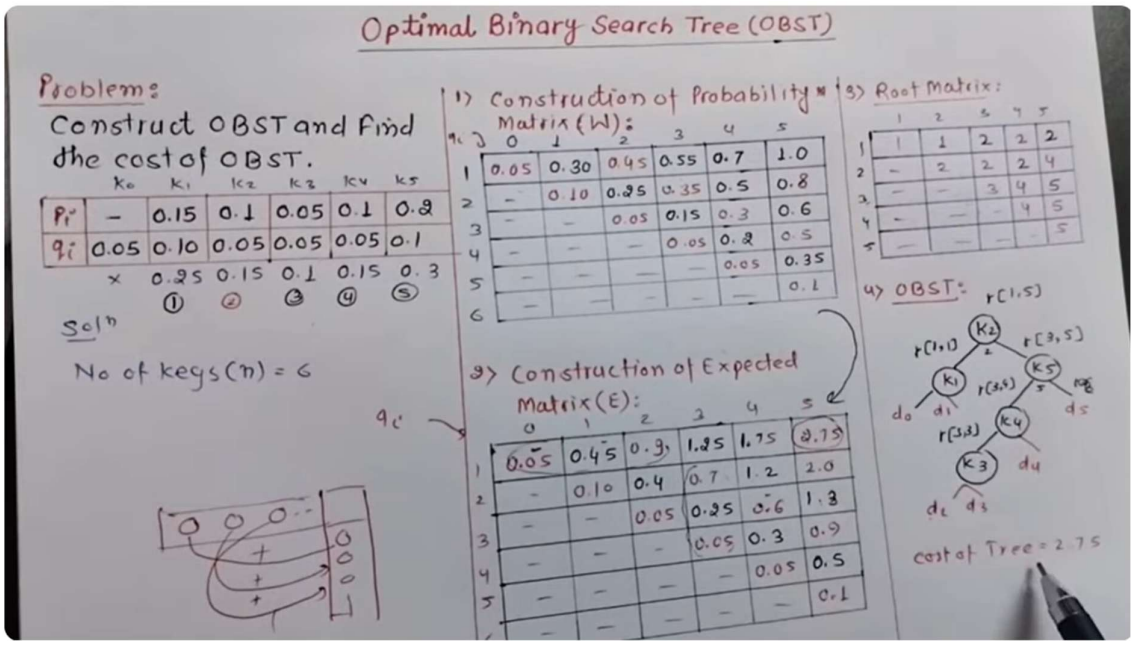
Keys: $K = \{10,20,30\}$
Probabilities: $p = \{0.2,0.5,0.3\}$

$Probabilities\ of\ unsuccessful\ search$: $q = \{0.1,0.05,0.05,0.1\}$.

Using DP approach below

$$c[i,j] = c[i,k-1] + c[k+1,j] + \sum_{s=i}^{j} f_s$$

"Cost of keys i to j = minimum of (left subtree cost + right subtree cost + sum of all frequencies in i..j) for all roots r from i to j."



Algorithm of OBST

Pick a root:

- You have many keys. Try making **each key as the root** one by one.

Make left and right parts:

- Keys smaller than root → left subtree
- Keys bigger than root → right subtree

Calculate cost:

- The cost = (cost of left subtree) + (cost of right subtree) + (sum of frequencies of all keys in this part)

Choose the best root:

- The root that gives **the smallest total cost** becomes the actual root.

Repeat for each subtree:

- Do the same steps for the left part and right part **recursively** until all keys are placed.

🌳 Minimum Spanning Tree (MST)

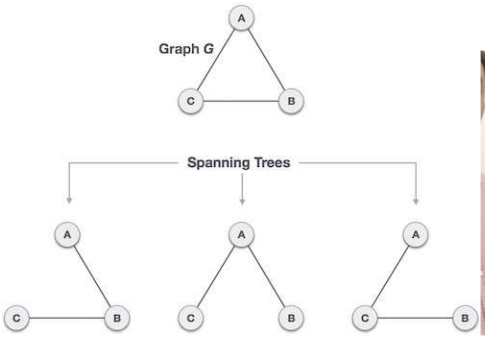
Definition:

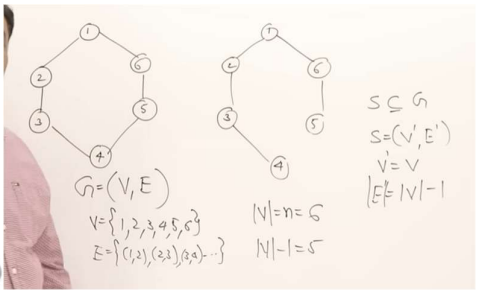
A **Minimum Spanning Tree** is a subset of edges of a **connected, weighted, undirected graph** that:

1. Connects **all vertices**
2. Contains **no cycles**
3. Has **minimum possible total weight**

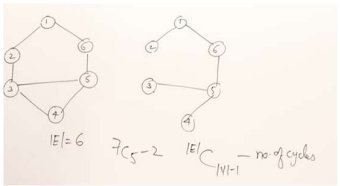
📌 Key Points

- MST has exactly **(V – 1) edges**
- Only works for **undirected graphs**
- If multiple MSTs are possible, any one is fine
- Used in networks, routing, clustering, etc.





Formula for total spanning tree (generally no of cycle is 1 but in below 2)



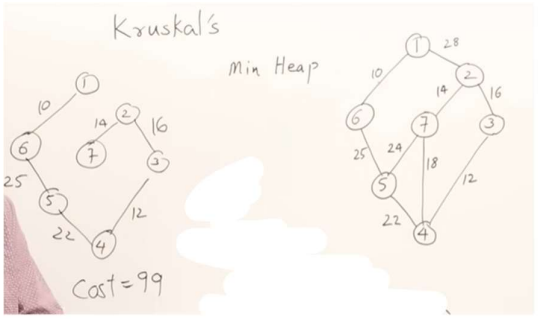
Formula for Complete graph K_n: Total spanning trees =n^{n-2}

⭐ Two Main Algorithms for MST

1️⃣ Kruskal's Algorithm (Greedy)

- Sort edges by weight (ascending)
- Pick smallest edge that **does not form a cycle**
- Use **Disjoint Set (Union–Find)** to check cycles
- Continue until you pick **V – 1 edges**
- ✓ It will generate **one spanning tree per connected component**.
  ✗ It cannot produce a single MST for the whole graph.

Time:
O(E log E)

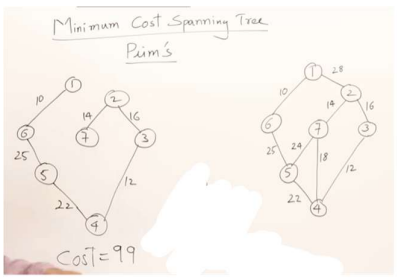



```
Input: Graph G(V, E) with weights

1. Sort all edges in ascending order of weight
2. Initialize MST = empty set
3. For each edge (u, v) in sorted edges:
       if adding (u, v) does NOT form a cycle in MST:
           add (u, v) to MST
       if MST has (V-1) edges:
           break
4. Output MST
```

2️⃣ Prim's Algorithm (Greedy)

- Start from any vertex
- Add the **cheapest edge** that connects tree to a new vertex
- Continue until all vertices are included
- Works like Dijkstra's but with edge weights only

Time:
O(E log V) using Min-Heap




```
1. Pick any starting vertex S
2. Initialize MST = {S}, total cost = 0
3. While MST does not contain all vertices:
       Find the minimum weight edge (u, v) such that
           u in MST, v not in MST
       Add v to MST and the edge (u, v)
4. Output MST
```

Aspect	Kruskal	Prim
Greedy Choice	Pick **edges** by weight	Pick **vertex + edge** with min cost
Works better	Sparse graphs	Dense graphs
Can handle disconnected graphs?	✅ Minimum Spanning Forest	❌ Only one connected component

Algorithm	TC	SC	Notes
Kruskal	O(E log E)	O(E + V)	Better for **sparse graphs**
Prim (list + heap)	O(E log V)	O(V + E)	Better for **dense graphs**

Single Source Shortest Path (SSSP) Algorithms

SSSP algorithms find the **shortest paths from a single source vertex** to **all other vertices** in a weighted graph.

- **Dijkstra = Greedy approach, cannot handle negative edges**
- **Bellman-Ford = Dynamic programming approach, can handle negative edges and detect negative cycles**

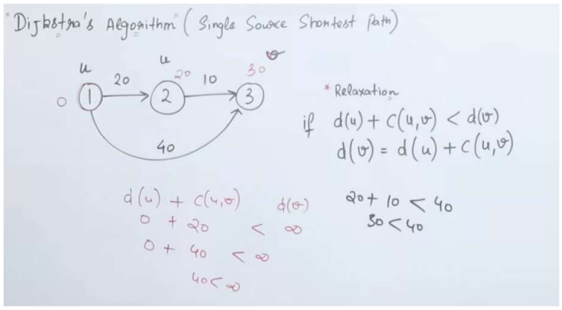
Dijkstra's Algorithm

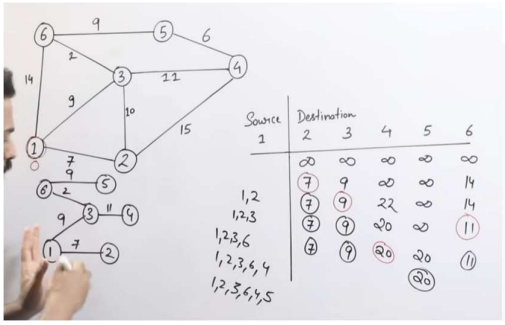
Dijkstra's algorithm finds the **shortest path from a source vertex to all other vertices** in a **weighted graph with non-negative edge weights**.

Key Points:

- Works for **connected or disconnected graphs**
- **Cannot handle negative weight edges**
- Uses **greedy approach**: always pick the vertex with the **minimum distance** so far

- Can be implemented with **arrays** (simple) or **priority queue** (efficient)



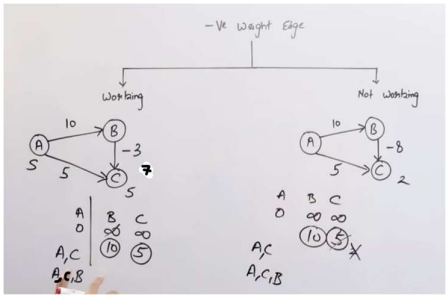


```
1. Initialize distance[] for all vertices as INF, except source = 0
2. Create a set of visited vertices
3. While there are unvisited vertices:
       a) Pick vertex u with minimum distance
       b) Mark u as visited
       c) For each neighbor v of u:
              if distance[u] + weight(u,v) < distance[v]:
                  distance[v] = distance[u] + weight(u,v)
4. Repeat until all vertices are visited
5. distance[] now contains shortest distances from source
```

Implementation	Time Complexity	Space Complexity
Using Min-Heap (Priority Queue)	O(E log V)	O(V + E)

Note: **When Dijkstra's Algorithm Fails: Negative Edge Weights**

- **Dijkstra assumes all edge weights ≥ 0**.
- If the graph has a **negative edge**, Dijkstra may **choose a wrong path** because it greedily assumes "current shortest distance" is final.



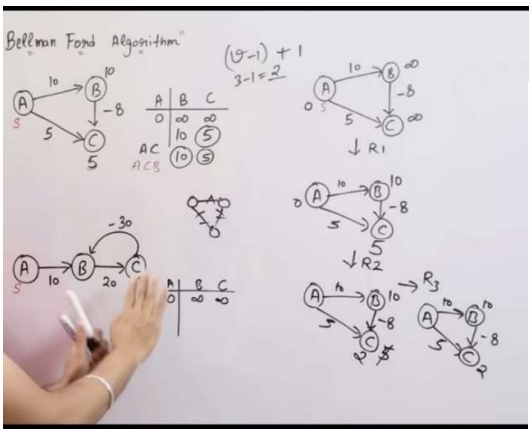
Bellman-Ford Algorithm

The Bellman-Ford algorithm finds the **shortest path from a source vertex to all other vertices** in a **weighted graph**, **even if edges have negative weights**.

It can also **detect negative weight cycles**.

Key Points:

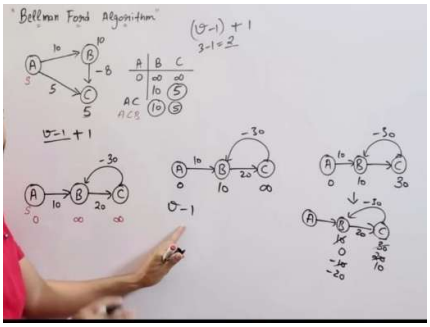
- Slower than Dijkstra: **O(V × E)**
- Used in routing, networks, and graphs with negative edges

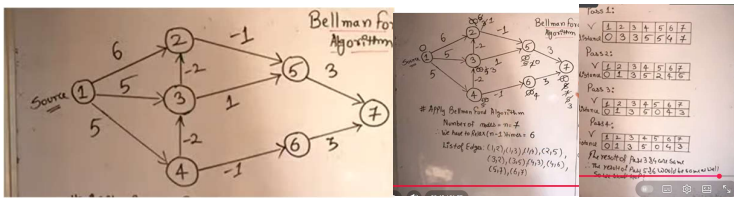


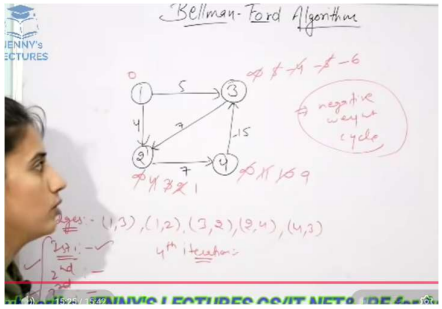

```
1. Initialize distance[] for all vertices as INF
   distance[S] = 0

2. Repeat (V-1) times:
       For each edge (u, v) with weight w:
           if distance[u] + w < distance[v]:
               distance[v] = distance[u] + w

3. Check for negative cycles:
       For each edge (u, v) with weight w:
           if distance[u] + w < distance[v]:
               "Graph contains negative cycle"
```

Advantage over Dijkstra

- Works with **negative weights**
- Can detect **negative cycles**



Question : 

Question on negative cycle : 

Time & Space Complexity Value

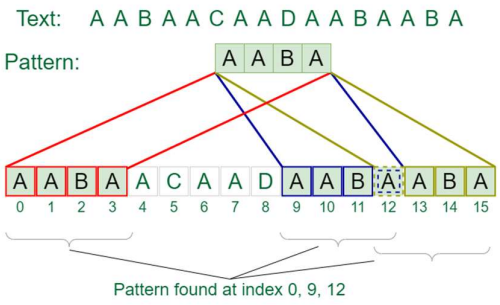
Time	O(V × E)
Space	O(V)

String matching algorithm

Shift-1=index (if question want to know after how many shift pattern found)

Naive Pattern Searching algorithm:

- The Naive String-Matching Algorithm is the simplest method to search for a pattern P in a text T.
- It's called "naive" because it doesn't use any preprocessing or clever tricks—just checks every possible position
- Importantly, it doesn't stop at the first occurrence; it can find all occurrences of the pattern in the text. (For example, in the diagram below, we can see that the pattern occurs **3 times** in the text.)




```java
public static void search(String pat, String txt) {
    int M = pat.length();
    int N = txt.length();

    // A loop to slide pat[] one by one
    for (int i = 0; i <= N - M; i++) {
        int j;

        // For current index i, check for pattern match
        for (j = 0; j < M; j++) {
            if (txt.charAt(i + j) != pat.charAt(j)) {
                break;
            }
        }

        // If pattern matches at index i
        if (j == M) {
            System.out.println("Pattern found at index " + i)
        }
    }
}
```

Aspect	Complexity
Worst-case time	**O((n-m+1)*m) =>** O(n · m)
Best-case time	O(**n-m+1**) => O(n)
Average-case time	O(n · m)
Space	O(1)

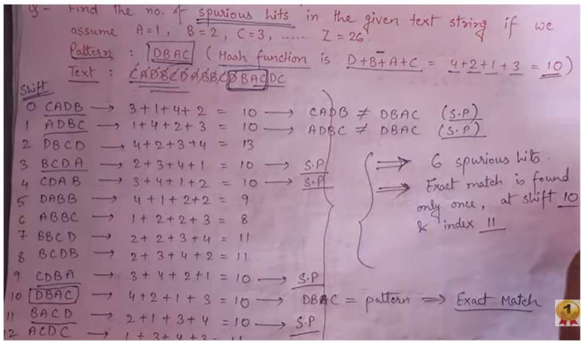
Rabin-Karp Algorithm for Pattern Searching

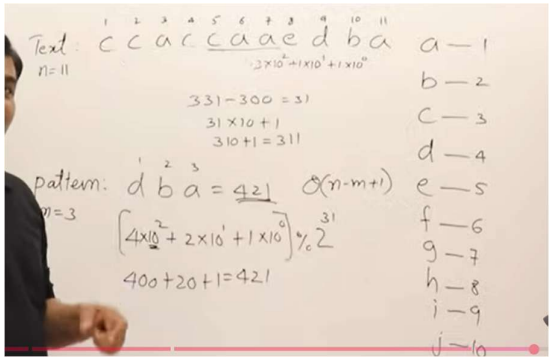
The Rabin-Karp algorithm is a string-searching algorithm that uses hashing to efficiently find a pattern within a text by comparing hash values of the pattern and substrings of the text, rather than comparing characters one by one.

- **True Hit:** Hash matches **and** characters match → valid pattern occurrence.
- **Spurious Hit:** Hash matches **but** characters do not match → false positive.
- No Hit: Hash does not match → skip substring.

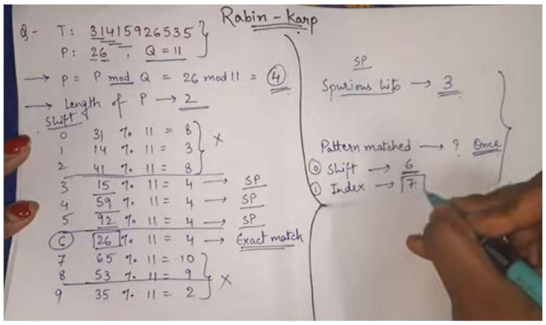
Type of question on rabin-karp algorithm

When alphabet is given alphabet but using hashing to avoid spurious hit





when number is given with mod q



Case	Time Complexity
Best	O(n + m) **(no hash collisions)**
Average	O(n + m) **(few hash collisions)**
Worst	O(n × m) **(many hash collisions)**
Space Complexity	O(1) **(O(m) if precomputing powers for rolling hash)**

- **Without precomputing powers:** Each time you slide the window, you recompute powers like p^(m-1), which is slower.
- **With precomputing powers:** Store p^0...p^(m-1) in an array once → next hashes are calculated in O(1), faster but uses O(m) extra space.

Advantages:

- Fast average-case using **hashing**.
- Good for **multiple pattern search**.
- Uses **rolling hash**, avoids recomputation.

Disadvantages:

- **Worst-case O(n×m)** if many hash collisions.
- Needs **hash computation & handling collisions**.
- Less efficient for **short texts**.

---

total operations in naïve and rabin karp

- **Text:** aaaaaab (length 7)
- **Pattern:** aab (length 3)

1️⃣ **Naive Algorithm**

- Compare pattern at every possible starting index i = 0 to n-m = 4.
- Character comparisons per index:

i	Substring	Comparisons
0	"aaa"	3
1	"aaa"	3
2	"aaa"	3
3	"aaa"	3
4	"aab"	3

Total character comparisons = 3 + 3 + 3 + 3 + 3 = 15 ✅

2️⃣ **Rabin-Karp Algorithm**

- Compute **pattern hash** (1 operation)
- Compute **rolling hash** for every substring (4 rolling hashes)
- Only **compare characters if hash matches** (last substring "aab") → 3 comparisons

Total operations (roughly):

- Hash calculations: 1 initial + 4 rolling = 5
- Character comparisons = 3

✅ **Rabin-Karp: 5 hash + 3 char comparisons = 8 "operations"**

KMP(Knuth-Morris-Pratt) STRING MATCHING ALGORITHM

- KMP (Knuth-Morris-Pratt) is a fast-string-matching algorithm that finds a **pattern P** inside a **text T** by avoiding rechecking of characters using a helper array called **LPS (Longest Prefix Suffix)**.
- KMP is always linear — O(n + m) in best, average, and worst cases — because LPS prevents any backtracking in the text.

Case	Complexity	Condition (When It Happens)
Best Case	**O(n + m)**	Characters match straight without many LPS jumps (pattern quickly aligns).
Average Case	**O(n + m)**	Normal text where occasional mismatches occur and LPS helps skip redundant checks.
Worst Case	**O(n + m)**	Highly repetitive text & pattern causing maximum LPS usage (e.g., "AAAAAA...AA" & "AAA...AB"). Still linear because KMP never rechecks characters.
Space Complexity	**O(m)**	LPS array stores values only for the pattern of length *m*.

Advantages of KMP

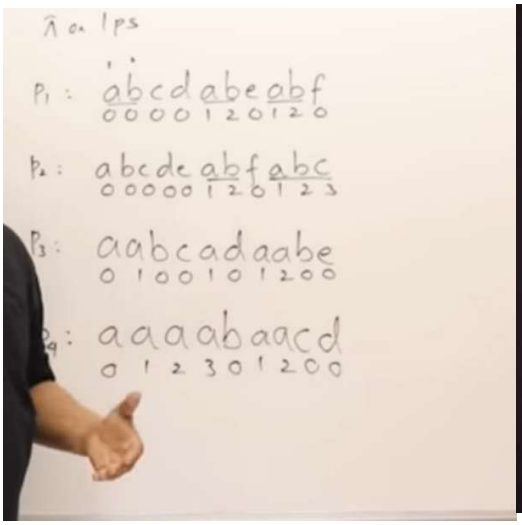
1. **Linear Time Complexity – O(n + m)**
   No backtracking in the text; much faster than naive in worst case.
2. **Efficient for long texts with repeated patterns**
   LPS table prevents rechecking characters.
3. **Deterministic performance**
   Best, average, worst cases are all linear.
4. **Good for real-time searching**
   Useful in text editors, searching logs, DNA sequences, etc.
5. **Works well with repetitive patterns**
   Examples: "AAAAABAAAA".

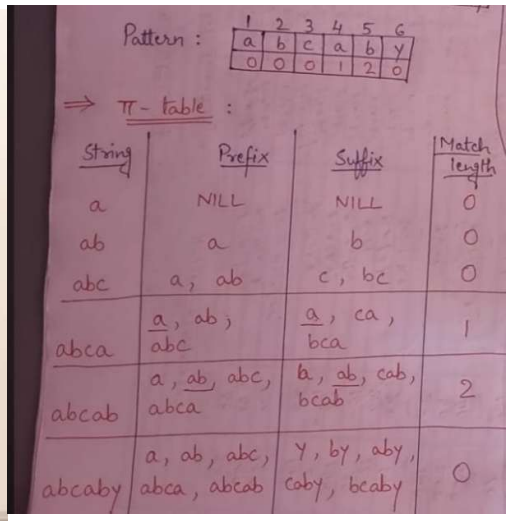
Disadvantages of KMP

1. **Building LPS table adds extra work**
   Preprocessing overhead O(m).
2. **More complex to understand and implement**
   Harder than naive or simple sliding-window methods.
3. **Uses extra space O(m)**
   Needs the LPS array for the pattern.
4. **Slower in simple / random data compared to naive**
   Naive sometimes works faster when mismatches happen early.

what is the PI / Prefix / LPS Table?

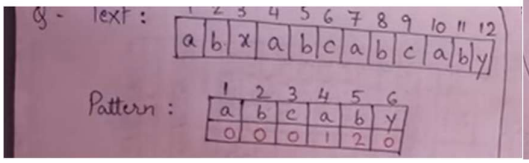
- It is an array that stores the length of the **longest proper prefix** which is also a **suffix** for each prefix of the pattern.
- It helps KMP know **how many characters can be reused** after mismatch.

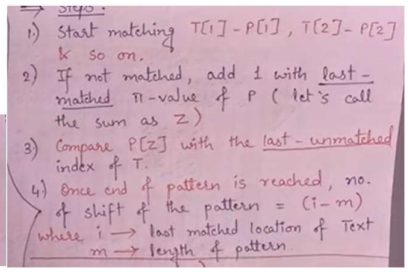


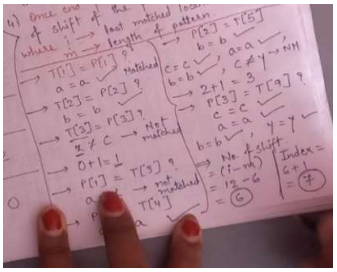


Match length in “abcab” is 2 because ab is common in both has length 2

Algo







STRING MATCHING USING FINITE AUTOMATA (FA METHOD)

Finite Automata method builds a **deterministic finite automaton (DFA)** for the pattern and uses it to scan the text in **O(n)** time.

- **Preprocessing Phase (Building DFA Transition Table) : Time Complexity: O(m · |Σ|),**
- Searching phase write below

Case	Time Complexity	Condition
Best Case	**O(n)**	Each char processed once, no special jumps.
Average Case	**O(n)**	All transitions done in constant time.
Worst Case	**O(n)**	DFA always moves in O(1) time per character.
Space Complexity:	**O(m · |Σ|)**	m = pattern length |Σ| = number of possible characters

---

⭐ **Key Idea**

1. Preprocess the pattern **P** and build a DFA transition table.
2. Run the text **T** through the DFA.
3. When DFA reaches the **final state = m**, pattern is found.

Advantages

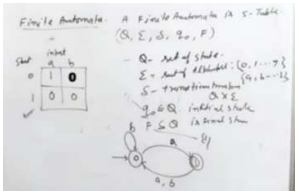
- **Searching is extremely fast → O(n)**
- **No backtracking** (always moves forward)
- **Perfect for multiple searches with same pattern** (only preprocess once)
- **Deterministic** — one transition per character

Disadvantages

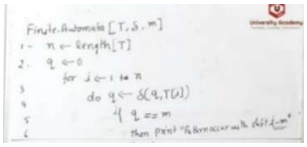
- **Preprocessing is slow → $O(m^2 \cdot |\Sigma|)$**
- **Uses large memory → O(m · |Σ|)**

- Harder to implement than Naive or KMP
- Not practical when alphabet is large (like Unicode)

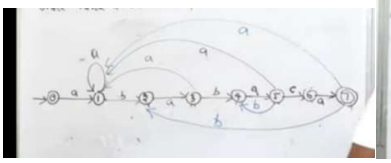
Finite Automata

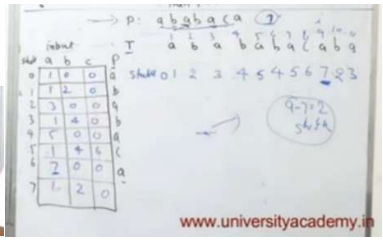


Algo



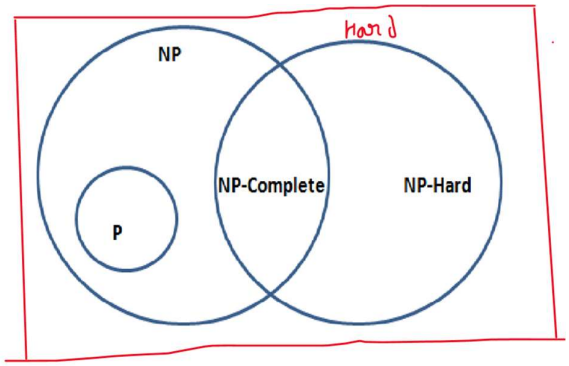
Example





Unit 5

complexity classes:



1. P (Polynomial Time)

A problem is in **P** if:

- It can be **solved** in polynomial time
  (Examples: **O(n), O(log n), O(n log n), O(n^2)**, etc.) using a **deterministic polynomial** algorithm

Examples:

- Linear Search
- Binary Search
- Quick Sort
- Insertion Sort

---

2. NP (Nondeterministic Polynomial Time)

A problem is in **NP** if:

- It may require **exponential time** to *solve*.
- But a given solution can be **verified in polynomial time** Using a **nondeterministic polynomial** algorithm.

Examples:

- Satisfiability (SAT)
- Knapsack
- Longest Common Subsequence (LCS)
- Travelling Salesman (decision version)

(**Note:** Factorial is not an NP problem; computing factorial is easy → O(n).)

---

3. NP-Hard

A problem is **NP-Hard** if:

- **Every NP problem can be reduced to it** in polynomial time.
- It is **at least as hard as NP** problems.
- It **does NOT need to be in NP**, so verification may not be polynomial.

Examples:

- Travelling Salesman Problem (optimization version)
- Halting Problem (NP-hard but not in NP)

---

4. NP-Complete

A problem is **NP-Complete** if it satisfies **both**:

✓ Condition 1: It is in NP

(solution can be verified in polynomial time)

✓ Condition 2: It is NP-hard

(all NP problems can be reduced to it in polynomial time)

Examples:

- SAT (Cook's theorem)
- Vertex Cover
- Clique
- Subset Sum

5 Other Related Classes

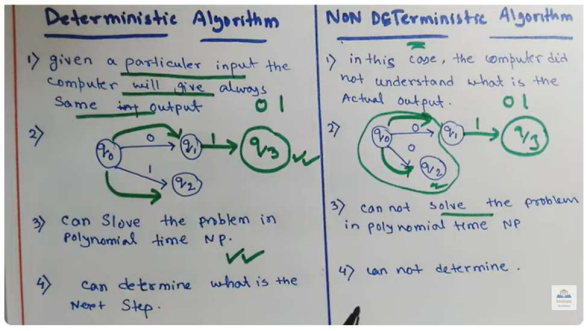
Class	Description	Example
co-NP	Complements of NP problems	UNSAT (formula is unsatisfiable)
PSPACE	Problems solvable with polynomial memory	Generalized Chess, TQBF

Class	Description	Example
EXP	Solvable in exponential time	Many combinatorial search problems

☑ Difference between Decision and Optimization Problem

Decision Problem	Optimization Problem
Output is **Yes/No**	Output is a **best (optimal) solution**
Asks: **Does a solution exist?**	Asks: **What is the best solution?**
Easy to verify (fits NP definition)	Harder; gives actual optimum value
Example → "Is there a path of length ≤ k?"	Example → "Find the shortest path."
Used in NP, NP-Complete, NP-hard theory	Used in real-world problem-solving (cost, time, weight)
Solutions checked quickly	Solutions computed, not just checked
Reduced from optimization by adding a limit k	Reduced to decision form for complexity analysis

Deterministic vs non deterministic algorithm



1 Deterministic Linear Search

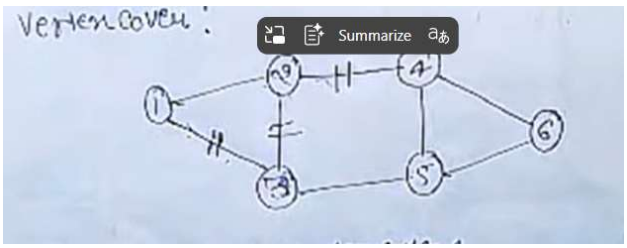
```java
int linearSearch(int[] A, int x) {
    for (int i = 0; i < A.length; i++) {
        if (A[i] == x) return i; // found
    }
    return -1; // not found
}
```

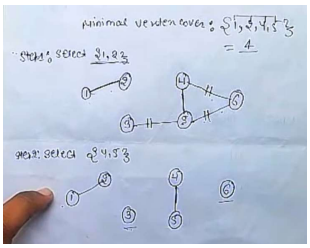
2 Non-deterministic Linear Search

```java
int linearSearchNonDeterministic(int[] A, int x) {
    int i = nonDeterministicChoice(0, A.length-1); // "guess" index
    if (A[i] == x) return i; // found
    else return -1; // not found
}
```

Vertex Cover problem

- A **graph** consists of **vertices (nodes)** and **edges (connections)**.
- A **vertex cover** is a set of vertices such that **every edge in the graph touches at least one vertex from this set**.
- In other words, every edge is “covered” by the chosen vertices.





Clique Decision Problem

Definition:

- Given a graph **G = (V, E)** and an integer **k**, the **Clique Decision Problem** asks:

“Does the graph contain a **clique of size k**?”

A **clique** in a graph is a **complete subgraph**, meaning:

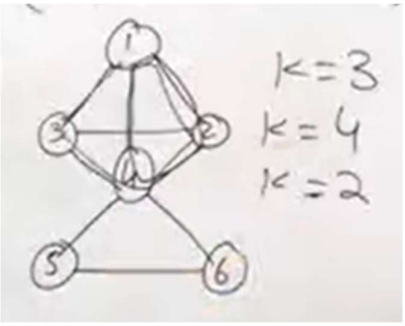
- Every pair of vertices in the clique is connected by an edge.
- It can be of any size (2 vertices, 3 vertices, ..., up to the size of the graph).

Example Graph:

Vertices: {A, B, C, D}

Edges: {(A,B), (A,C), (B,C), (C,D)}

- k = 3 → Is there a clique of size 3?
- **Answer:** Yes → {A, B, C} is a clique because all pairs are connected:
    - (A,B), (A,C), (B,C)
- k = 4 → Is there a clique of size 4?
- **Answer:** No → There is no subset of 4 vertices where every pair is connected.



k=2 (5,6) k=3 (1,3,4) k=4(1 3 2 4)

What is Reduction?

Definition:

Reduction is the process of converting one problem into another problem in **polynomial time** such that a solution to the new problem gives a solution to the original problem.

Why it's useful:

- Shows how "hard" a problem is by comparing it to a known hard problem.
- Used to prove **NP-hardness** and **NP-completeness**.

2️⃣ How Reduction Works (Step by Step)

1. Take a known problem **P** (usually NP-Complete).
2. Transform any instance of **P** into an instance of the new problem **Q**.
3. Ensure the transformation is **polynomial-time**.
4. The answer should be equivalent:
    - YES instance of P → YES instance of Q
    - NO instance of P → NO instance of Q

If you can do this, **Q is at least as hard as P** (NP-hard).

Example: Divisibility by 6 → Divisibility by 2 and 3

Problem A: Check if a number $n$is divisible by 6.
Problem B: Check if a number $n$is divisible by 2 **and** divisible by 3.

Step 1: Transform Problem A into Problem B

- Take the number $n$from Problem A.
- Instead of directly checking divisibility by 6, check **Problem B**: is $n$divisible by 2 **and** 3?

Step 2: Solve Problem B

- If B says **Yes** (divisible by 2 and 3), then **A is also Yes** (divisible by 6).
- If B says **No**, then **A is No**.

Step 3: Conclusion

- We successfully **reduced Problem A to Problem B**.
- Solving B solves A.
- The transformation (checking divisibility by 2 and 3) is **polynomial time** (very fast).

Example: 3-SAT → Vertex Cover

Step 1: Known NP-Complete Problem: 3-SAT

- Given a boolean formula in 3-CNF (Conjunctive Normal Form), check if it is satisfiable.

Step 2: New Problem: Vertex Cover

- Given a graph G and number k, is there a vertex cover of size ≤ k?

Step 3: Reduction Idea:

- Convert each variable and clause in 3-SAT into vertices and edges in a graph.
- Construct the graph such that:
    - If the formula is satisfiable → there exists a vertex cover of size k in the graph
    - If formula is not satisfiable → no vertex cover of size k exists

Step 4: Result:

- By solving Vertex Cover, you can solve 3-SAT.
- Therefore, **Vertex Cover is NP-hard**.