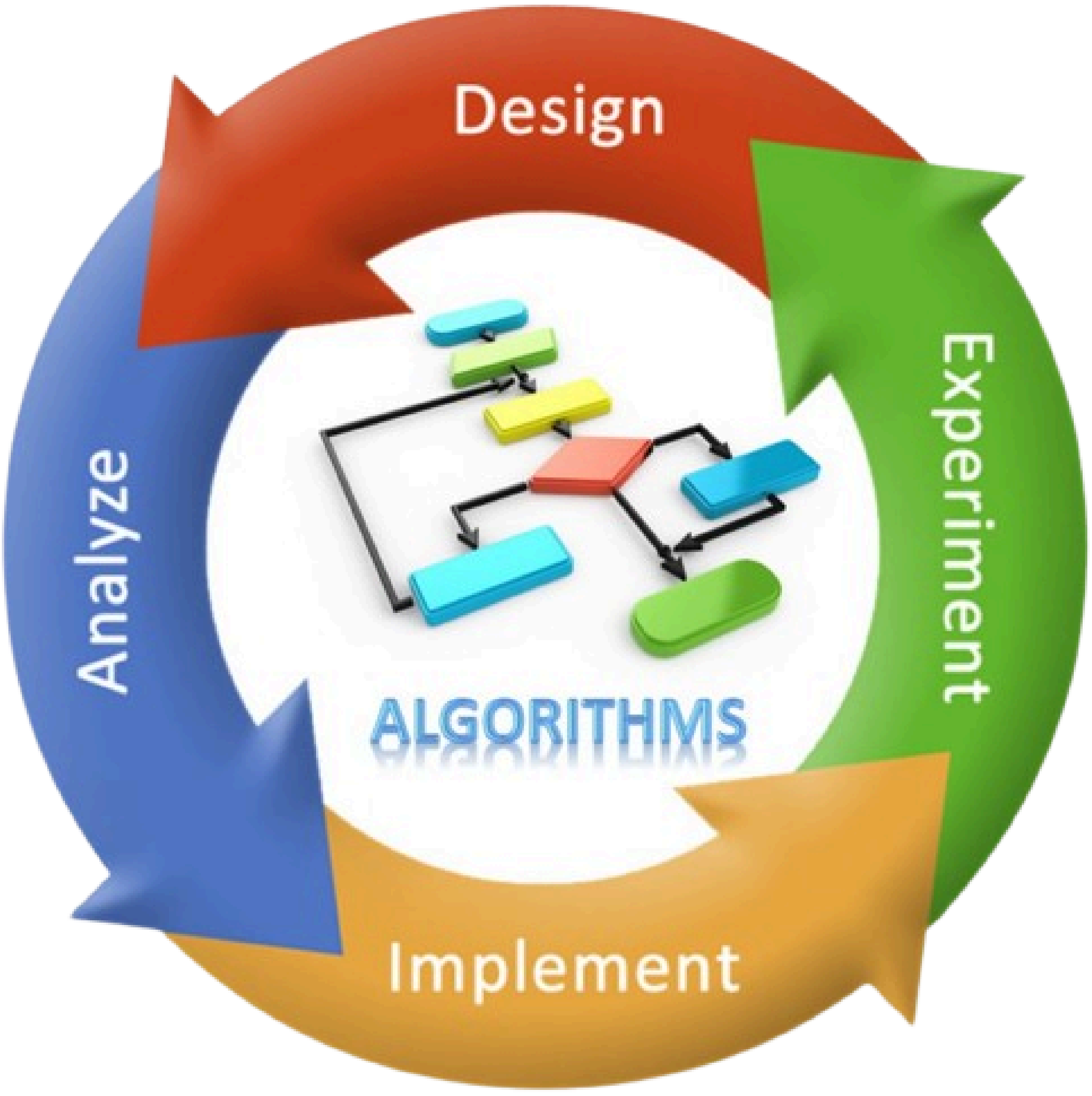
Algorithm 2.2

For 2023 second year second semester



Made By:

MD. ASHIKUZZ ZAMAN

ID: 220119

Course Topics

1. Algorithms, Asymptotic Notations and Complexity Analysis

- Complexity analysis of algorithms
- Worst case, best case, and average case analysis

2. Sorting Algorithms

- Divide and Conquer approach
- Merge Sort and Quicksort: Algorithm and complexity analysis (worst and average case)
- Heap Construction Algorithm and Heap Sort
- Applications of Heap: Priority Queue
- Decision tree model and lower bound on sorting (worst case)
- Sorting in linear time: Radix Sort, Bucket Sort, Counting Sort

3. Graph Algorithms

- Graph representations
- Breadth-First Search (BFS) and Depth-First Search (DFS)
- Minimum Spanning Tree: Kruskal's Algorithm and Prim's Algorithm
- Shortest Path Algorithms: Dijkstra's Algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm

4. Searching Algorithms

- Binary Search Trees (BST)
- Balanced Binary Search Trees: AVL Trees and Red-Black Trees
- B-Trees, Skip Lists, Hashing
- Priority Queues, Heaps, Interval Trees

5. Dynamic Programming

- Longest Common Subsequence (LCS)
- Matrix Chain Multiplication (MCM)

6. Greedy Algorithms

- Greedy Algorithm approach
- Activity Selection Problem
- Huffman Coding and its applications
- Knapsack Problem
- Traveling Salesperson Problem (TSP)

7. Recurrences and Backtracking

- Recurrences
- NP-Hard and NP-Complete Problems
- Backtracking: n-Queens Problem
- Branch and Bound

8. NP-Completeness and Reducibility

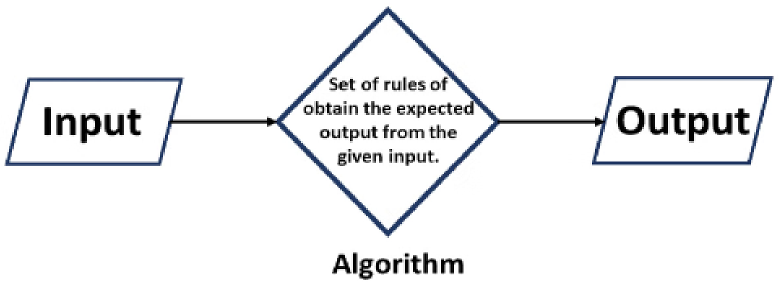
- Lower Bound Theory
- Discussion of NP-Complete problems: Satisfiability, Clique, Vertex Cover, Independent Set, Hamiltonian Cycle, TSP, Knapsack, Set Cover, Bin Packing

9. Computational Geometry

- Line Segment Properties
- Convex Hull
- Graham Scan Algorithm

1.Algorithm, Asymptotic Notations and Complexity Analysis:

Algorithm: An algorithm is a set of step-by-step instructions to accomplish a task or solve a problem.


Input
Set of rules of obtain the expected output from the given input.
Output
Algorithm

Why Do We Need Algorithms :-

- **Solve Problems Efficiently:** Algorithms help solve complex problems step by step in an organized way.
- **Save Time and Effort:** They make tasks faster and easier by automating processes.
- **Reliability:** Algorithms ensure accurate and consistent results.
- **Enable Computers:** They allow computers to handle tasks that humans can't do manually, like processing large amounts of data.
- **Everywhere in Use:** Algorithms are used in math, computer science, engineering, finance, and more to analyze data, optimize processes, and find solutions.

Applications of Algorithms :-

1. **Data Analysis and Processing:**
    - Used in sorting, searching, and organizing large datasets (e.g., search engines, data mining).
2. **Artificial Intelligence and Machine Learning:**
    - Algorithms power AI models, helping in image recognition, natural language processing, and predictions.
3. **Cryptography:**
    - Algorithms secure data by encryption and decryption, ensuring online security (e.g., SSL, blockchain).
4. **Pathfinding and Navigation:**
    - Used in GPS systems and robotics to find the shortest path (e.g., Dijkstra's Algorithm, A*).
5. **Software Development:**
    - Algorithms are essential for building efficient and scalable applications.

6. **Network Optimization:**
    - They help optimize network traffic, routing, and data transfer (e.g., in telecommunications).
7. **Finance and Economics:**
    - Used in stock market predictions, fraud detection, and portfolio management.
8. **Healthcare:**
    - Algorithms assist in diagnosing diseases, drug discovery, and patient monitoring.
9. **Image and Signal Processing:**
    - Found in video compression, noise reduction, and image editing software.
10. **E-commerce and Recommendation Systems:**
    - Algorithms suggest products, optimize pricing, and personalize user experiences (e.g., Amazon, Netflix).
11. **Scheduling and Optimization:**
    - Used in resource allocation, task scheduling, and supply chain management.
12. **Gaming:**
    - Algorithms power game logic, opponent behavior, and graphics rendering.

Algorithm Design Techniques

1. **Divide and Conquer**
    - **Definition**: This technique involves dividing a problem into smaller independent sub-problems, solving each sub-problem, and combining their solutions to solve the original problem.
    - **Examples**:
        - Tower of hanoi
        - Convex Hall
        - Merge Sort
        - Quick Sort
        - Binary Search
        - Strassen's Matrix Multiplication
2. **Greedy Algorithm**
    - **Definition**: Greedy algorithms build a solution step-by-step by selecting the locally optimal choice at each step without considering future consequences.
    - **Examples**:
        - Dijkstra's Shortest Path Algorithm
        - Prim's Minimum Spanning Tree Algorithm
        - Kruskal's Minimum Spanning Tree Algorithm
        - Huffman Encoding

        - Knapsack
        - Traveling Salesman Problem (TSP)
3. **Dynamic Programming**
    - **Definition**: Dynamic programming is a technique used to solve problems by breaking them into smaller, overlapping dependent sub-problems. Each sub-problem is solved only once, and its solution is stored to avoid redundant computations. It typically follows a **bottom-up approach** and works based on the **principle of optimality** (the optimal solution of a problem is built using the optimal solutions of its sub-problems).
    - **Examples**:
        - Fibonacci Sequence
        - Knapsack Problem
        - Longest Common Subsequence (LCS)
        - Matrix Chain Multiplication
4. **Backtracking**
    - **Definition**: This technique systematically explores all possible solutions by building a solution incrementally and backtracking when a solution path fails.
    - **Examples**:
        - N-Queens Problem
        - Hamiltonian Path
        - Subset Sum Problem
        - DFS for Searching
5. **Recursive Algorithm**
    - **Definition**: A recursive algorithm solves a problem by solving smaller instances of the same problem and combining their results.
    - **Examples**:
        - Tower of Hanoi
        - Factorial Calculation
        - Depth-First Search (DFS)
        - Merge Sort
6. **Branch and Bound**
    - **Definition**: This algorithm design technique used to solve optimization problems. It works by dividing the problem into smaller restricted subproblems (branching) and using a bounding function to eliminate subproblems that cannot possibly contain the optimal solution. This reduces the search space and focuses only on promising candidates.
    - **Examples**:

        - Traveling Salesman Problem (TSP)
        - Knapsack Problem (Exact Solution)
        - Integer Programming
        - Maximum Flow Problem
        - N.P Hard problem

7. **Brute Force**
    - **Definition**: Brute force algorithms solve problems by trying every possible solution until the correct one is found.
    - **Examples**:
        - Password Cracking
        - Generating Permutations of a String
        - Finding the Maximum Subarray
        - Linear Search
8. **Randomized Algorithm**
    - **Definition**: These algorithms use random numbers to influence their logic, often providing faster or approximate solutions.
    - **Examples**:
        - Randomized Quick Sort
        - Monte Carlo Algorithm
        - Randomized Primality Test (e.g., Miller-Rabin Test)
        - Reservoir Sampling

There are also many kinds of Algorithm like Heap, Graph Technique, Searching Algorithm, Shorting etc.

Complexity Analysis:-

The **complexity of an algorithm** refers to the performance of an algorithm in terms of time and space as the input size grows. It helps in understanding how efficient an algorithm is.

Types of Complexity

1. **Time Complexity**: Measures how the execution time of an algorithm increases with input size.

2. **Space Complexity**: Measures how much extra memory an algorithm needs as input size increases.

Space Complexity

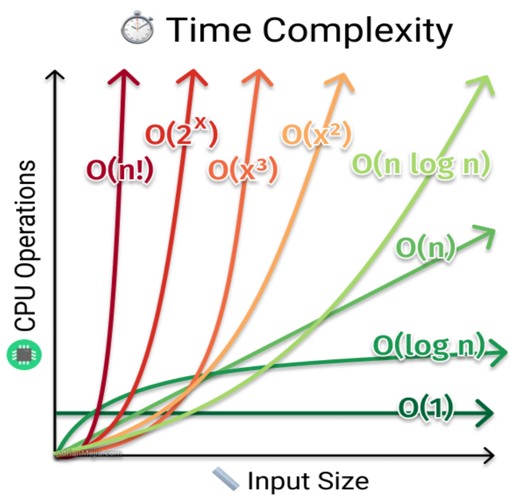
Space complexity depends on:

- Input storage
- Auxiliary storage (temporary variables, recursion stack, etc.)

Space Complexity	Example
O(1)	Using a few variables
O(N)	Storing an array of size N
$O(N^2)$	Storing a matrix of size N × N

Time Complexity (Big-O Notation) :-

Complexity	Notation	Example
Constant	O(1)	Accessing an element in an array
Logarithmic	O(log N)	Binary search
Linear	O(N)	Looping through an array
Linearithmic	O(N log N)	Merge Sort, Quick Sort (average case)
Quadratic	$O(N^2)$	Nested loops (Bubble Sort, Selection Sort)
Cubic	$O(N^3)$	Triple nested loops
Exponential	O(2^N)	Recursive Fibonacci
Factorial	O(N!)	Traveling Salesman Problem



Asymptotic Analysis:-

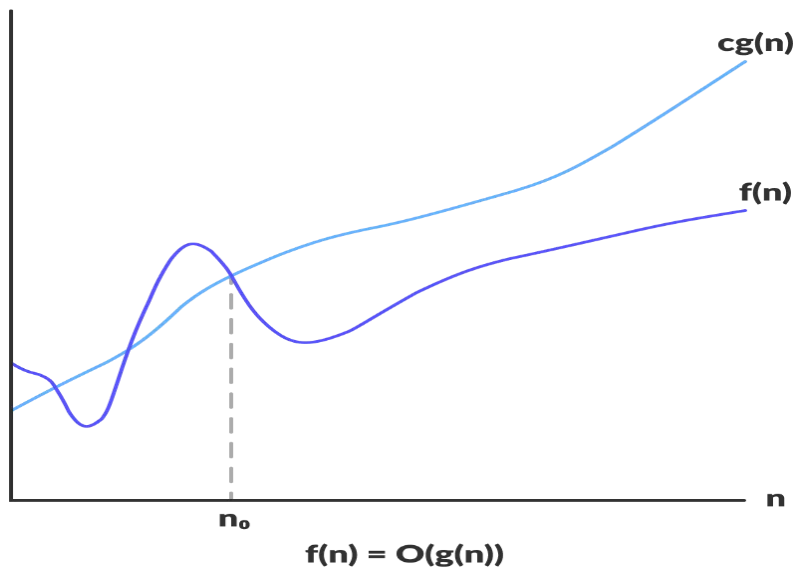
Asymptotic analysis studies the performance of algorithms on large input sizes, ignoring constants and lower-order terms.

Importance

1. Provides a simplified way to analyze algorithm efficiency.
2. Allows comparison of different algorithms' performance.

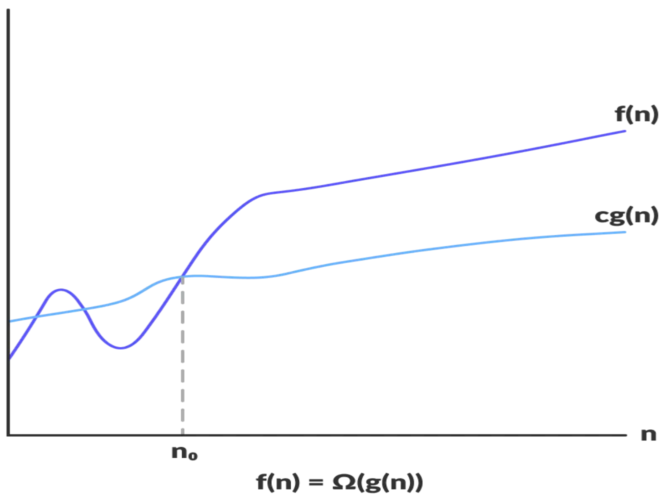
1. Big-O Notation (O) → Worst Case

- **Definition**: Represents the upper bound (maximum time an algorithm can take).
- **Example**: If an algorithm runs in **O(n^2)**, it means that in the worst case, the time complexity grows at most like **n^2**.



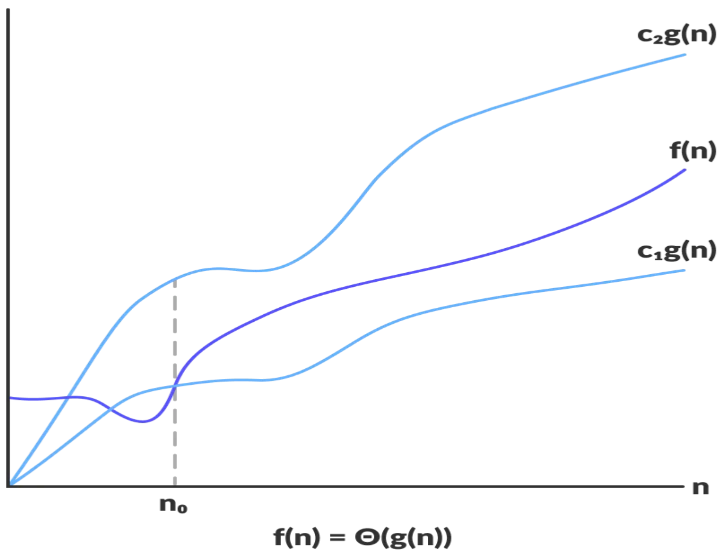
2. Omega Notation (Ω) → Best Case

- **Definition**: Represents the lower bound (minimum time required by an algorithm).
- **Example**: If an algorithm runs in **Ω(n)**, it means that in the best case, the algorithm will take at least **n** time.



3. Theta Notation (Θ) → Average Case

- **Definition**: Represents both upper and lower bounds . It represents actual time an algorithm takes (tight bound).
- **Example**: If an algorithm runs in **Θ(n log n)**, it means that the time complexity is always around **n log n**, no matter the case.



Applications of Stack

A stack is a **LIFO (Last In, First Out)** data structure used in:

1. **Function Call Management:**

    - Used in programming languages for managing function calls, recursion, and local variables (Call Stack).
2. **Expression Evaluation and Conversion:**
    - Used to evaluate postfix expressions or convert infix expressions to postfix or prefix.
3. **Undo Operations in Applications:**
    - Commonly used in text editors, where the last action can be undone (Undo/Redo).
4. **Parsing:**
    - Used in compilers to check for syntax correctness in expressions like parentheses matching.
5. **Backtracking:**
    - Used in algorithms like maze-solving, N-Queens, and Depth-First Search (DFS).
6. **Memory Management:**
    - Helps allocate and deallocate memory during runtime in programs.
7. **Reversing Data:**
    - Used to reverse strings or numbers efficiently.

Applications of Queue

A queue is a **FIFO (First In, First Out)** data structure used in:

1. **Task Scheduling:**
    - Used in operating systems to schedule processes (e.g., Round Robin Scheduling).
2. **Data Buffering:**
    - Used in input/output systems for buffering data (e.g., printers, keyboards).
3. **Breadth-First Search (BFS):**
    - Used in graph traversal algorithms to explore nodes level by level.
4. **Asynchronous Data Transfer:**
    - Used in message queues or communication systems to handle tasks in order.
5. **Customer Service Systems:**
    - Used in ticket booking systems or help desks, where customers are served in the order of their arrival.
6. **Traffic Management:**
    - Used in simulation models to represent queues of vehicles at signals.
7. **Job Scheduling:**
    - Used in batch processing where jobs are executed in the order they arrive.

8. **Cache Implementation:**
    - In applications like the Least Recently Used (LRU) cache, queues help manage data efficiently.

2. Sorting Algorithms

Divide and Conquer Approach: The **Divide and Conquer** strategy is a problem-solving approach that breaks a problem into smaller parts, solves them individually, and then combines the results to form the final solution. It consists of three main steps:

Divide

- Break the main problem into smaller subproblems.
- The subproblems should be similar to the original problem but smaller in size.
- This process continues until the subproblems become simple enough to be solved directly.
- **Example:** In **Merge Sort**, the input array is divided into two halves recursively.

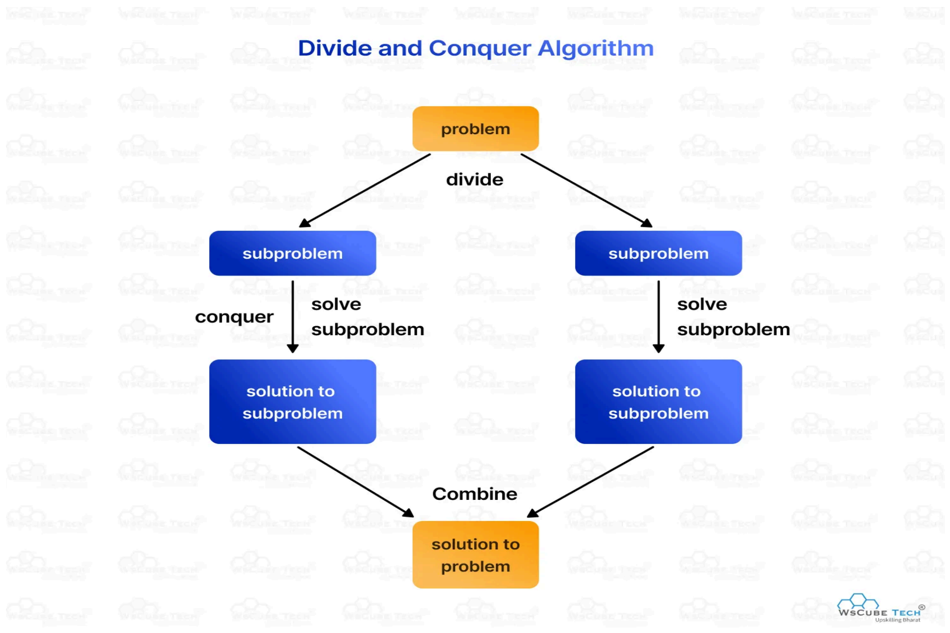
Conquer

- Solve each smaller subproblem independently.
- If a subproblem is small enough (base case), solve it directly.
- Otherwise, keep dividing until a base case is reached.
- **Example:** In **Merge Sort**, after splitting the array into single elements, each element is considered sorted.

Merge (Combine)

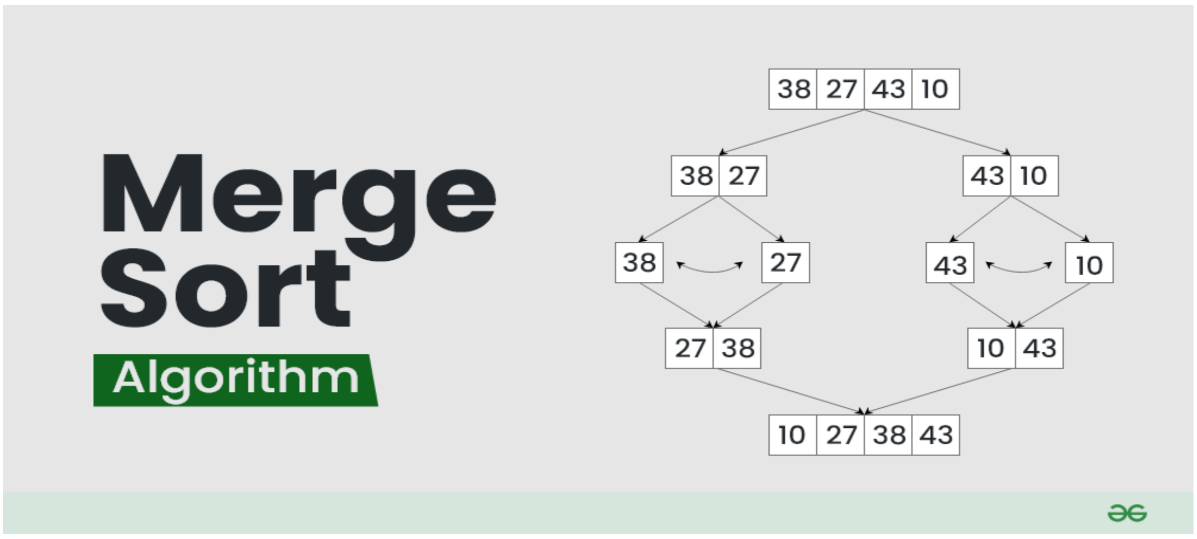
- Combine the solutions of subproblems to form the final solution of the original problem.
- This merging step is where the actual solution is built.
- **Example:** In **Merge Sort**, the sorted halves are merged to get a fully sorted array.

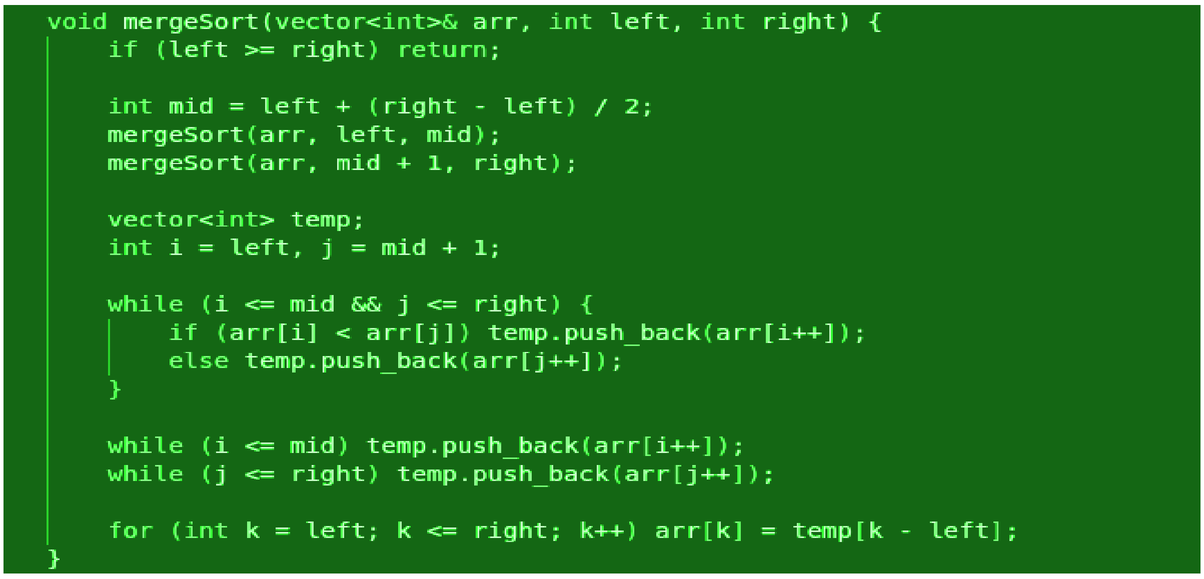
Divide and Conquer is mainly useful when we divide a problem into independent subproblems. If we have overlapping subproblems, then we use Dynamic Programming.



Aspect	Merge Sort	Quick Sort
Time Complexity	Always $O(n \log n)$.	Best: $O(n \log n)$, Worst: $O(n^2)$.
Space Complexity	$O(n)$ (uses extra memory).	$O(\log n)$ (in-place sorting).
Stability	Stable (preserves order of equal elements).	Not stable (unless modified).
Performance on Large Data	Good for external sorting (e.g., large files).	Faster for in-memory operations with random data.
Input Dependency	Performance is independent of input.	Performance depends on pivot selection.

Merge Sort:

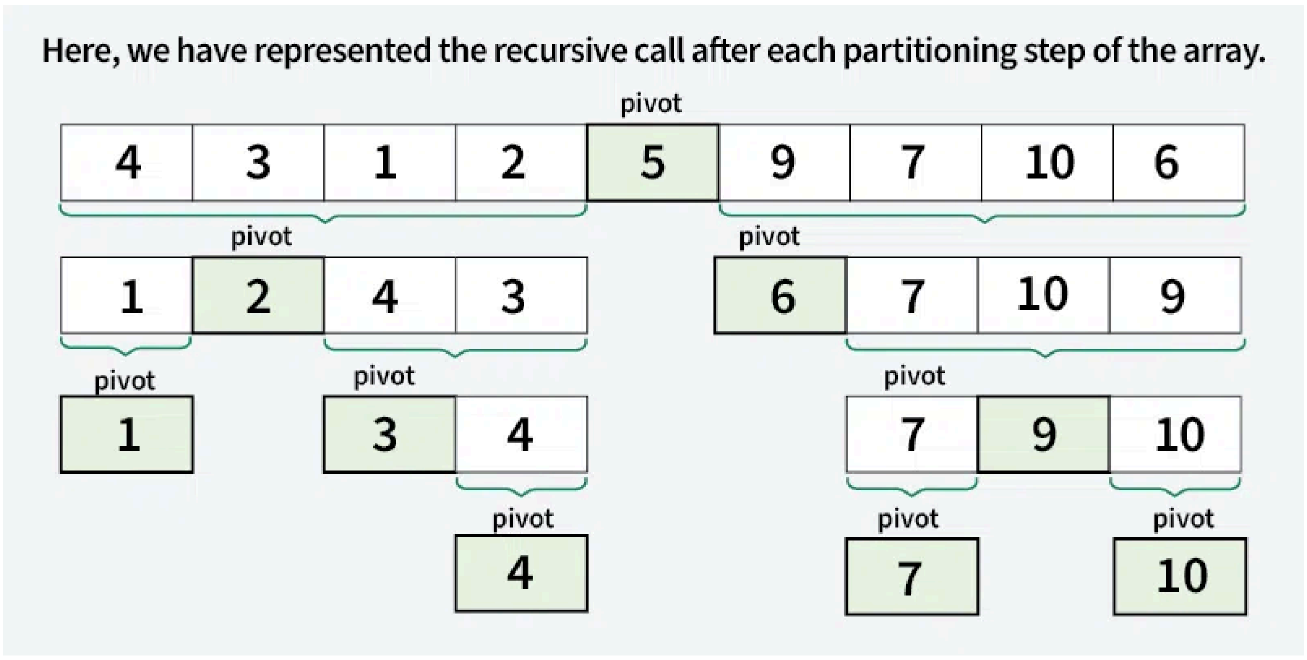


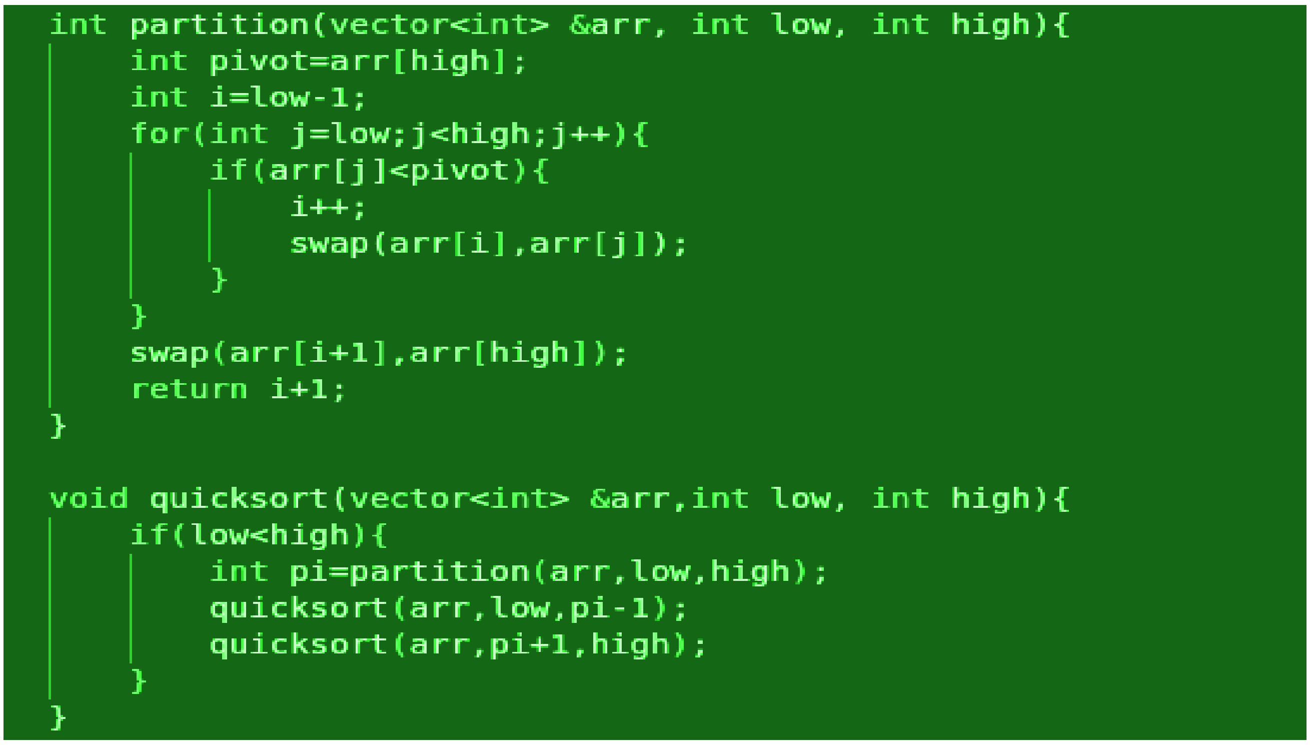


```
void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);

    vector<int> temp;
    int i = left, j = mid + 1;

    while (i <= mid && j <= right) {
        if (arr[i] < arr[j]) temp.push_back(arr[i++]);
        else temp.push_back(arr[j++]);
    }

    while (i <= mid) temp.push_back(arr[i++]);
    while (j <= right) temp.push_back(arr[j++]);

    for (int k = left; k <= right; k++) arr[k] = temp[k - left];
}
```

- **Best Case Time Complexity:** O(nlogn)
- **Worst Case Time Complexity:** O(nlogn)
- **Average Case Time Complexity:** O(nlogn)

Quick Sort


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

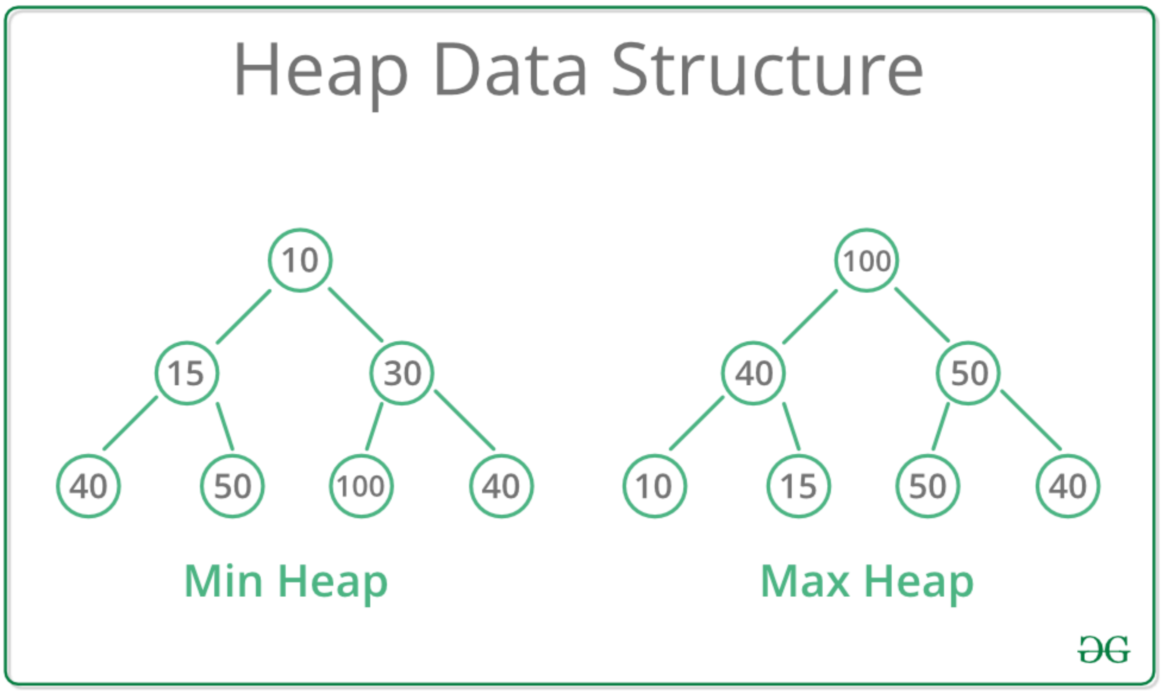

```
int partition(vector<int> &arr, int low, int high){
    int pivot=arr[high];
    int i=low-1;
    for(int j=low;j<high;j++){
        if(arr[j]<pivot){
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[high]);
    return i+1;
}

void quicksort(vector<int> &arr,int low, int high){
    if(low<high){
        int pi=partition(arr,low,high);
        quicksort(arr,low,pi-1);
        quicksort(arr,pi+1,high);
    }
}
```

- **Best Case Time Complexity:** O(nlogn) (Occurs when the pivot divides the array evenly.)
- **Average Case Time Complexity:** O(nlogn)
- **Worst Case Time Complexity:** O(n^2) (Occurs when the pivot is the smallest or largest element repeatedly, leading to unbalanced partitions.)

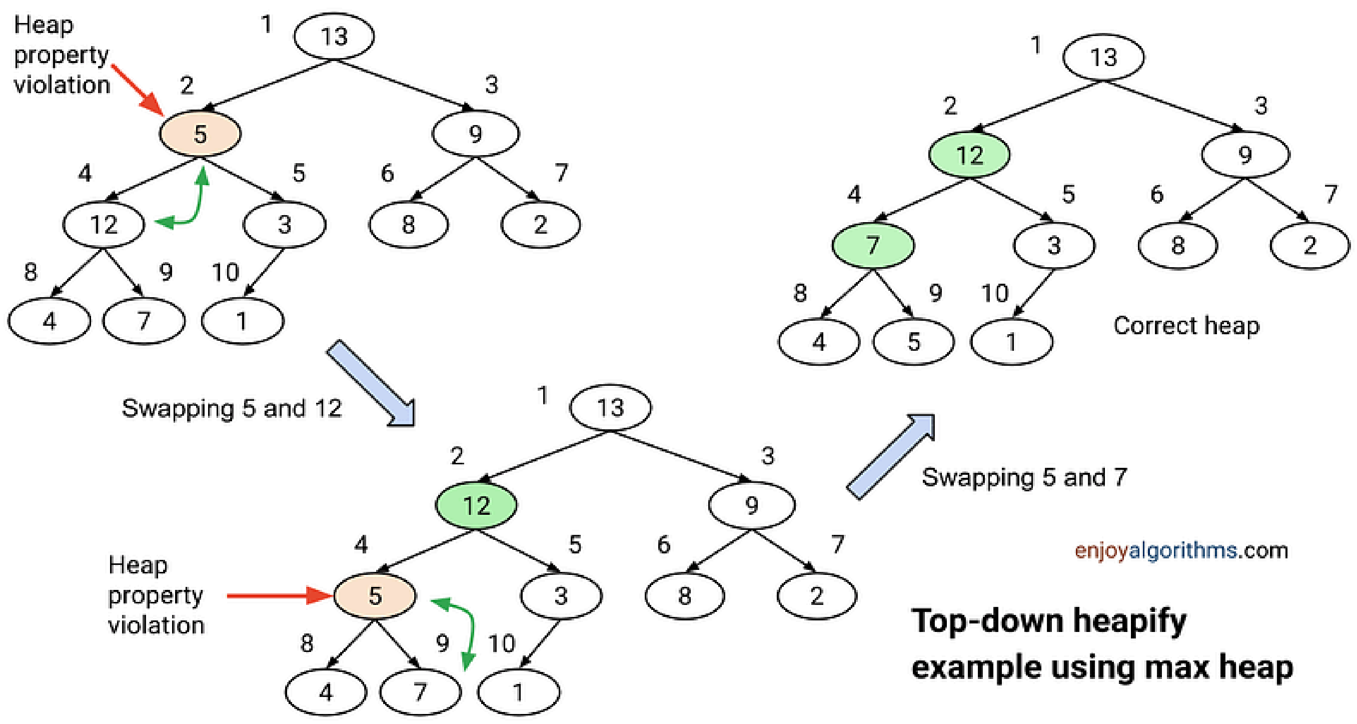
In term of Time complexity, Merge Sort is more appropriate because it guarantees O(nlogn) in all cases.

Heap Construction Algorithm and Heap Sort:

- **Heap** is a complete binary tree each of whose node's value is greater (or smaller) than the node value of its children. If the node value is greater than the node value of its children, then the heap is called max heap. Otherwise, the heap is called min heap.
- **Types**:
  - **Min Heap**: Parent node ≤ child nodes (smallest element at the root).
  - **Max Heap**: Parent node ≥ child nodes (largest element at the root).
- **Operations**:
  - **Insertion**: `O(log N)`
  - **Deletion (Extract-Min/Max)**: `O(log N)`
  - **Heapify**: `O(log N)`
- **Applications**:
  - Dijkstra's Algorithm (Shortest Path)
  - Prim's Algorithm (Minimum Spanning Tree)
  - Job Scheduling

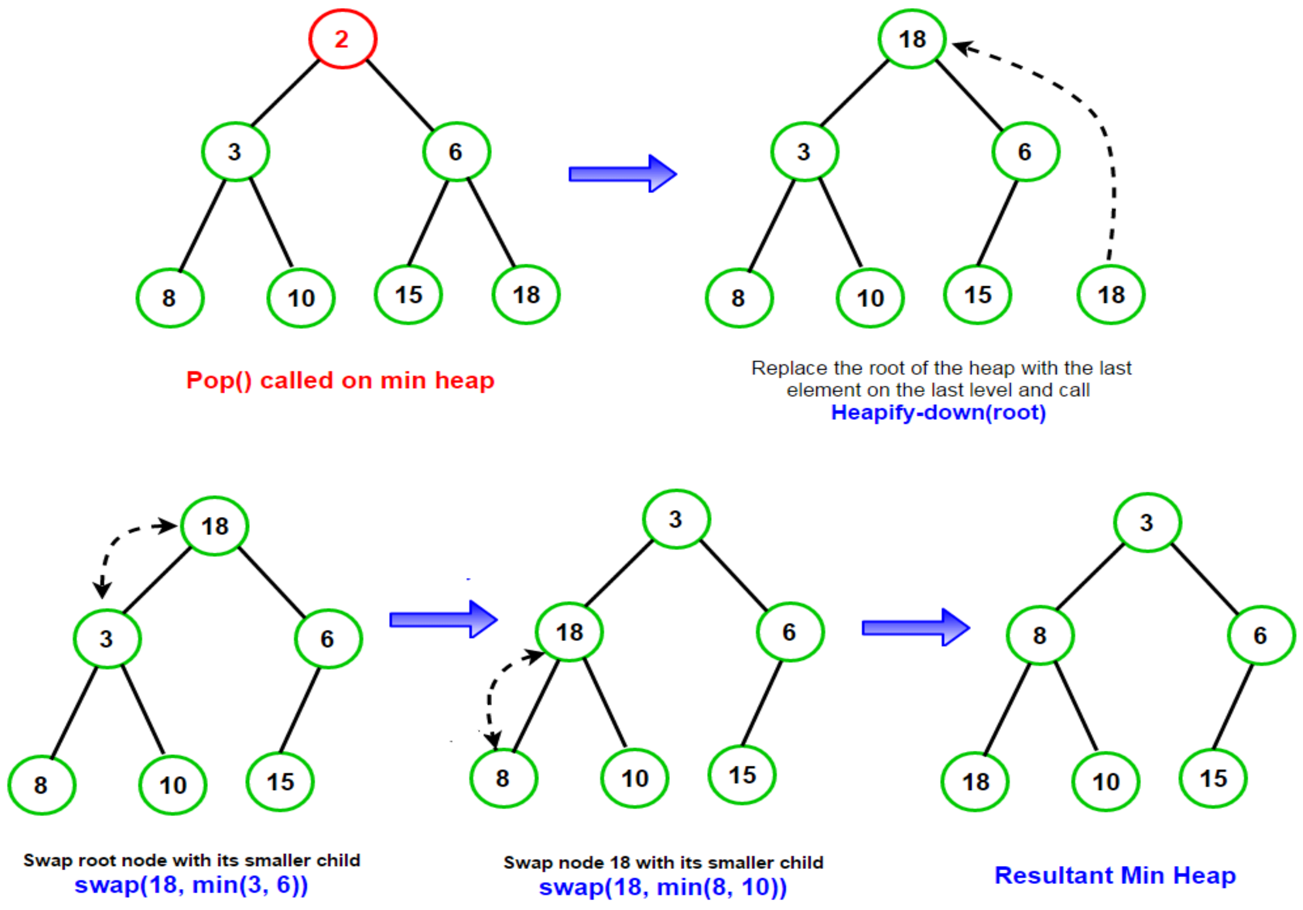


Max Heapify:



Heap property violation
Swapping 5 and 12
Heap property violation
Swapping 5 and 7
Correct heap
enjoyalgorithms.com
Top-down heapify example using max heap

Min Heapify:

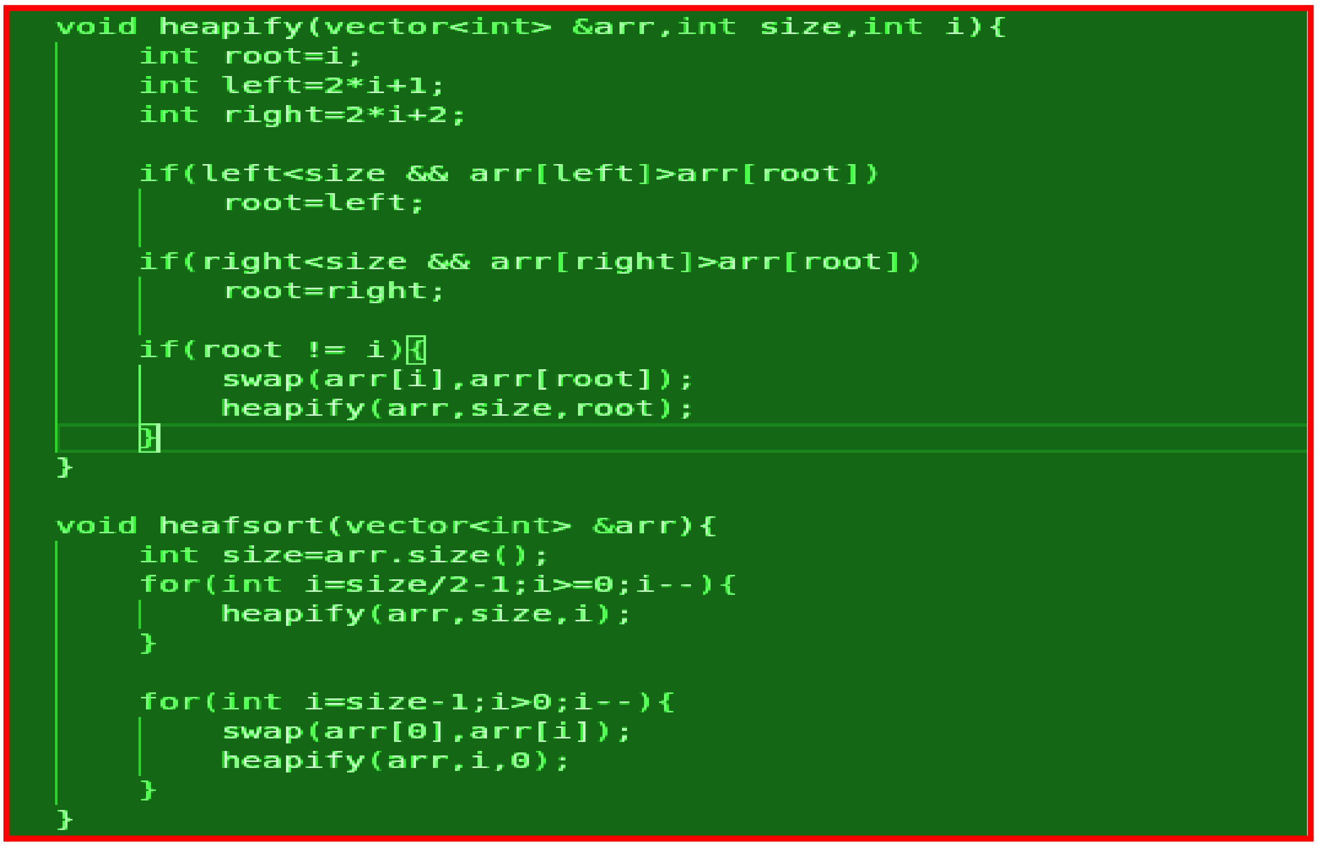


Pop() called on min heap
Replace the root of the heap with the last element on the last level and call
Heapify-down(root)
Swap root node with its smaller child
swap(18, min(3, 6))
Swap node 18 with its smaller child
swap(18, min(8, 10))
Resultant Min Heap

Steps to Construct a Max Heap:

1. **Start with an Empty Heap**
    - Add the first element as the root of the heap.
2. **Insert Each Element One by One**
    - Add the next element as a left child first.
    - If the left child is occupied, add it as a right child.
3. **Heapify Up (Maintain Max Heap Property)**
    - Compare the newly added child with its parent.
    - If the parent's value is smaller than the child's, swap them.
    - Repeat this process until the child is smaller than or equal to its parent or reaches the root.
4. **Repeat for All Elements**
    - Continue adding elements and applying the heapify-up process until all elements are inserted.

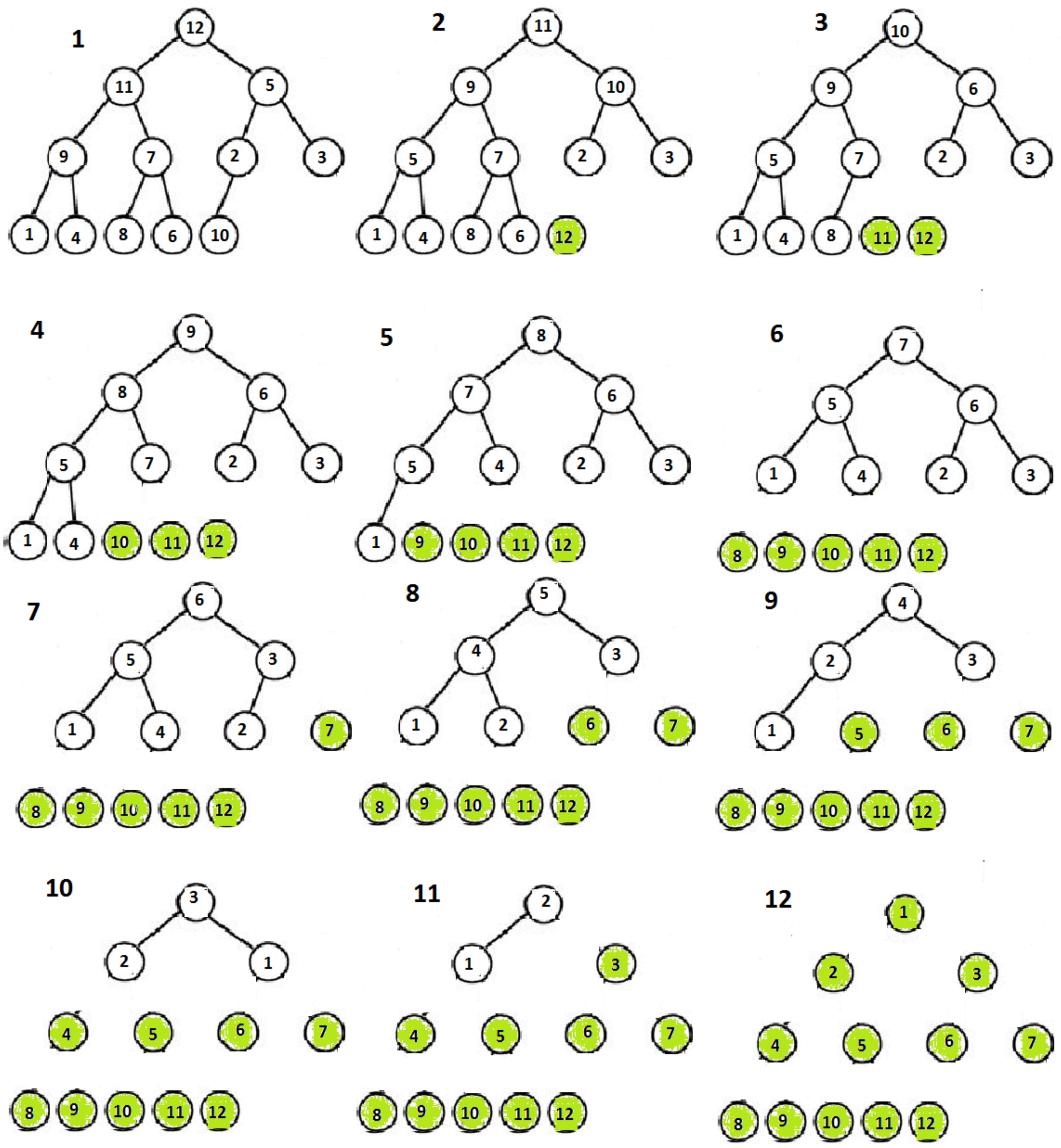
CPP code for Heap Sort (if you want, write this):



```cpp
void heapify(vector<int> &arr,int size,int i){
    int root=i;
    int left=2*i+1;
    int right=2*i+2;

    if(left<size && arr[left]>arr[root])
        root=left;

    if(right<size && arr[right]>arr[root])
        root=right;

    if(root != i){
        swap(arr[i],arr[root]);
        heapify(arr,size,root);
    }
}

void heafsort(vector<int> &arr){
    int size=arr.size();
    for(int i=size/2-1;i>=0;i--){
        heapify(arr,size,i);
    }

    for(int i=size-1;i>0;i--){
        swap(arr[0],arr[i]);
        heapify(arr,i,0);
    }
}
```

Heap-sort Algorithm:

- Convert the unsorted array into a Max Heap, where the largest element is at the root. This is done using the heapify operation.
- Swap the root (largest element) with the last element of the heap.
- Reduce the heap size (ignore the last sorted element).
- Heapify the root to maintain the heap property. Repeat this process until all elements are sorted.



Applications of Heap (Priority Queues):

1. **Task Scheduling in Operating Systems**
    - The CPU scheduler picks the highest-priority task from a queue.
    - Example: Process scheduling in Linux uses a priority-based approach.

2. **Dijkstra's Algorithm (Shortest Path in Graphs)**
    - A priority queue helps find the shortest path efficiently using a **Min Heap**.

3. **Prim's Algorithm (Minimum Spanning Tree)**
    - Used to find the Minimum Spanning Tree (MST) efficiently.

4. **Huffman Coding (Data Compression)**
    ○ Builds an optimal prefix code tree using a **Min Heap**.

5. **Event-Driven Simulations**
    ○ Used in simulations to process events in chronological order.

6. **Network Packet Scheduling**
    ○ Routers use priority queues to handle packets efficiently based on importance.

7. **Job Scheduling in Printers**
    ○ High-priority print jobs are processed before lower-priority ones.

Decision tree model and lower bound on sorting (worst case):

A decision tree is a graphical representation of different options for solving a problem and shows how different factors are related.

Click Here

3.Graph Algorithms:

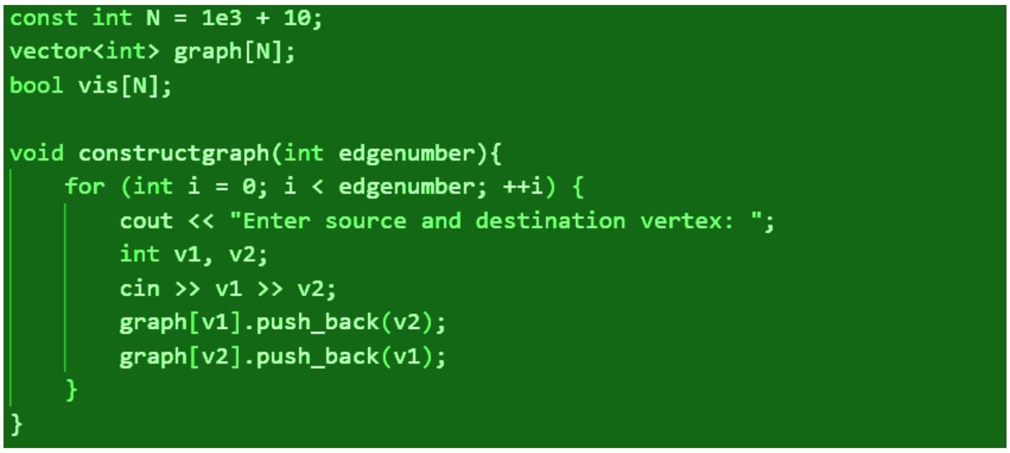
Graph:

A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

There are the two most common ways to represent a graph : For simplicity, we are going to consider only unweighted graphs in this post.

1. Adjacency Matrix
2. Adjacency List

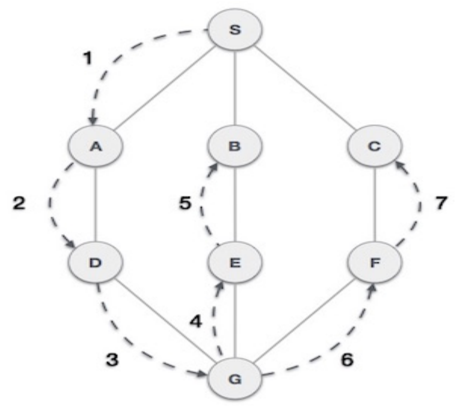
Construct graph:



```
const int N = 1e3 + 10;
vector<int> graph[N];
bool vis[N];

void constructgraph(int edgenumber){
    for (int i = 0; i < edgenumber; ++i) {
        cout << "Enter source and destination vertex: ";
        int v1, v2;
        cin >> v1 >> v2;
        graph[v1].push_back(v2);
        graph[v2].push_back(v1);
    }
}
```

Depth First Search (DFS) Algorithm:

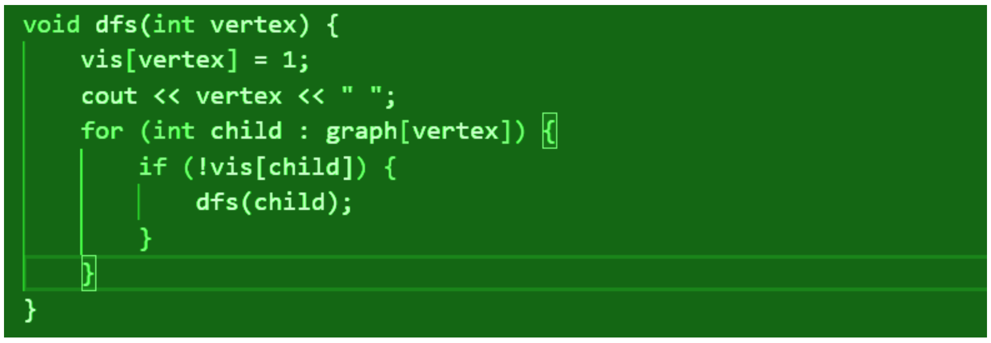
Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



Step for DFS Recursive Implementation:

- **Start DFS** from a vertex.
- Visit the current vertex, mark it, and explore its unvisited neighbors.
- The process continues recursively for each neighbor, following a depth-first path.
- **DFS** goes deep along a path, while **BFS** explores level by level using a queue.

DFS:



```
void dfs(int vertex) {
    vis[vertex] = 1;
    cout << vertex << " ";
    for (int child : graph[vertex]) {
        if (!vis[child]) {
            dfs(child);
        }
    }
}
```

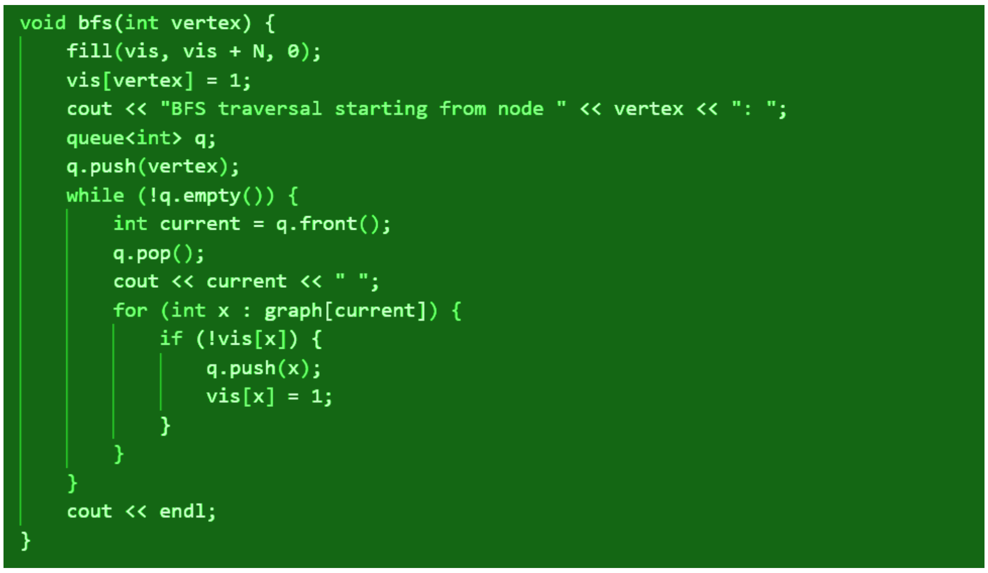
Breadth-First Search (BFS):

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent.

Step for DFS Recursive Implementation:

- **Start BFS** from a vertex.
- Visit the current vertex and explore all its immediate neighbors.
- Enqueue all unvisited neighbors and repeat the process level by level, moving outward from the starting point.

BFS:



```
void bfs(int vertex) {
    fill(vis, vis + N, 0);
    vis[vertex] = 1;
    cout << "BFS traversal starting from node " << vertex << ": ";
    queue<int> q;
    q.push(vertex);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";
        for (int x : graph[current]) {
            if (!vis[x]) {
                q.push(x);
                vis[x] = 1;
            }
        }
    }
    cout << endl;
}
```

<u>Minimum Spanning Tree (MST):</u>

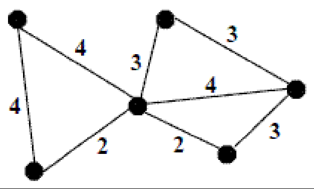
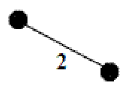
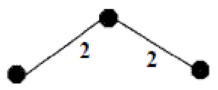
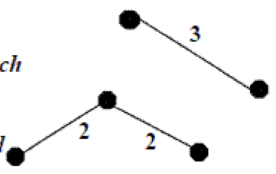
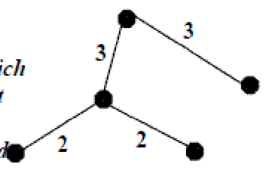
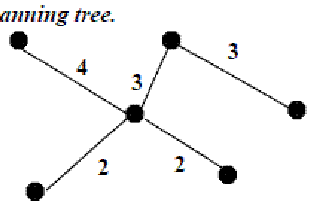
The **minimum spanning tree** is a spanning tree whose sum of the edges is minimum. Consider the below graph that contains the edge weight.

Kruskal's algorithm:

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's Algorithm

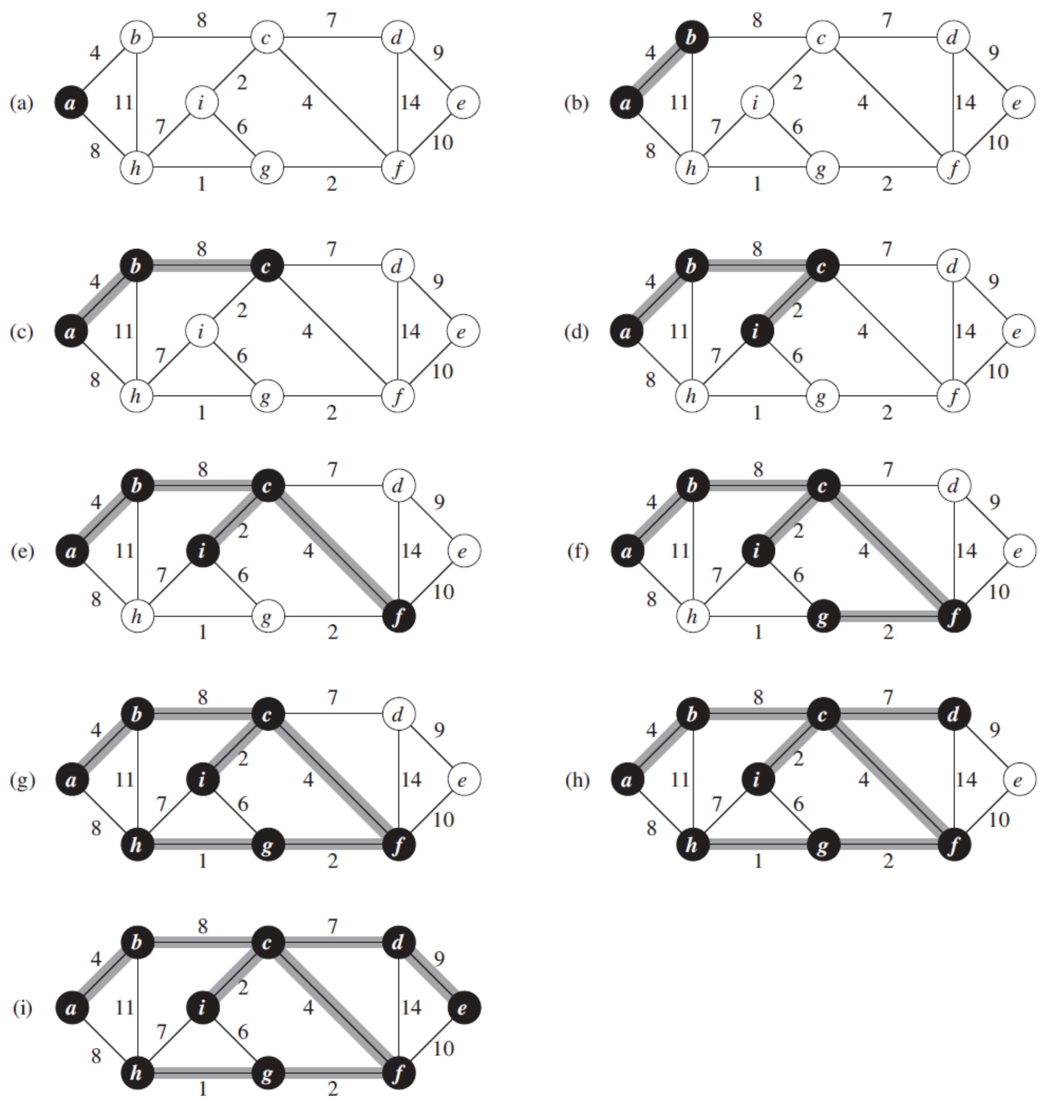
1 *Given a network...........* 	2 *Choose the shortest edge (if there is more than one, choose any of the shortest)........* 	3 *Choose the next shortest edge and add it........* 
4 *Choose the next shortest edge which wouldn't create a cycle and add it.* 	5 *Choose the next shortest edge which wouldn't create a cycle and add it.* 	6 *Repeat until you have a minimal spanning tree.* 

Prim's Algorithm:

Minimum Spanning Tree (MST) is to build the MST incrementally by starting with a single vertex and gradually extending the tree by adding the closest neighboring vertex at each step.

Below are the steps for finding **MST** using **Prim's algorithm**:

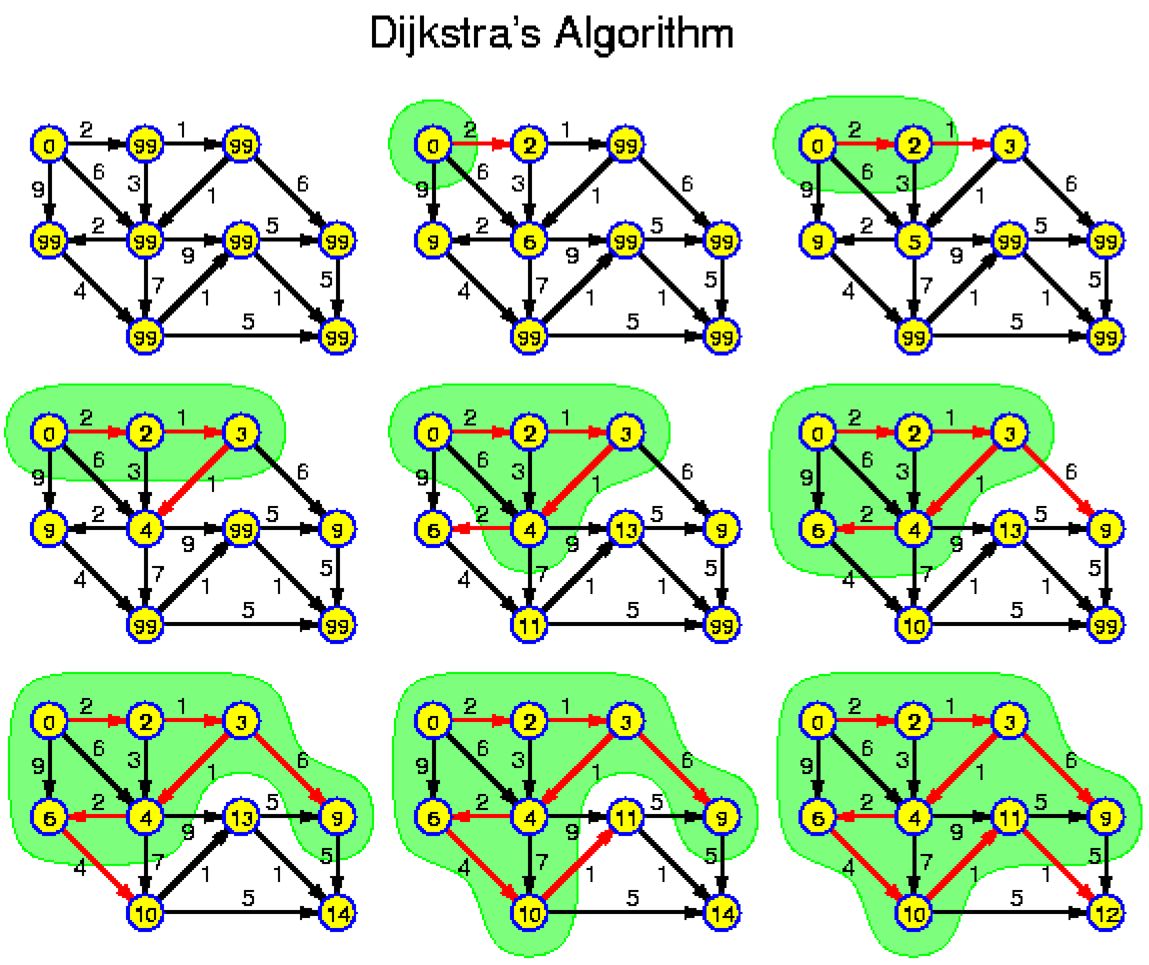
1. Start by picking any vertex and add it to the MST.
2. Look for the smallest edge that connects a vertex already in the MST to a vertex not yet in the MST.
3. Add that edge and vertex to the MST.
4. Repeat this process until all vertices are included in the MST.



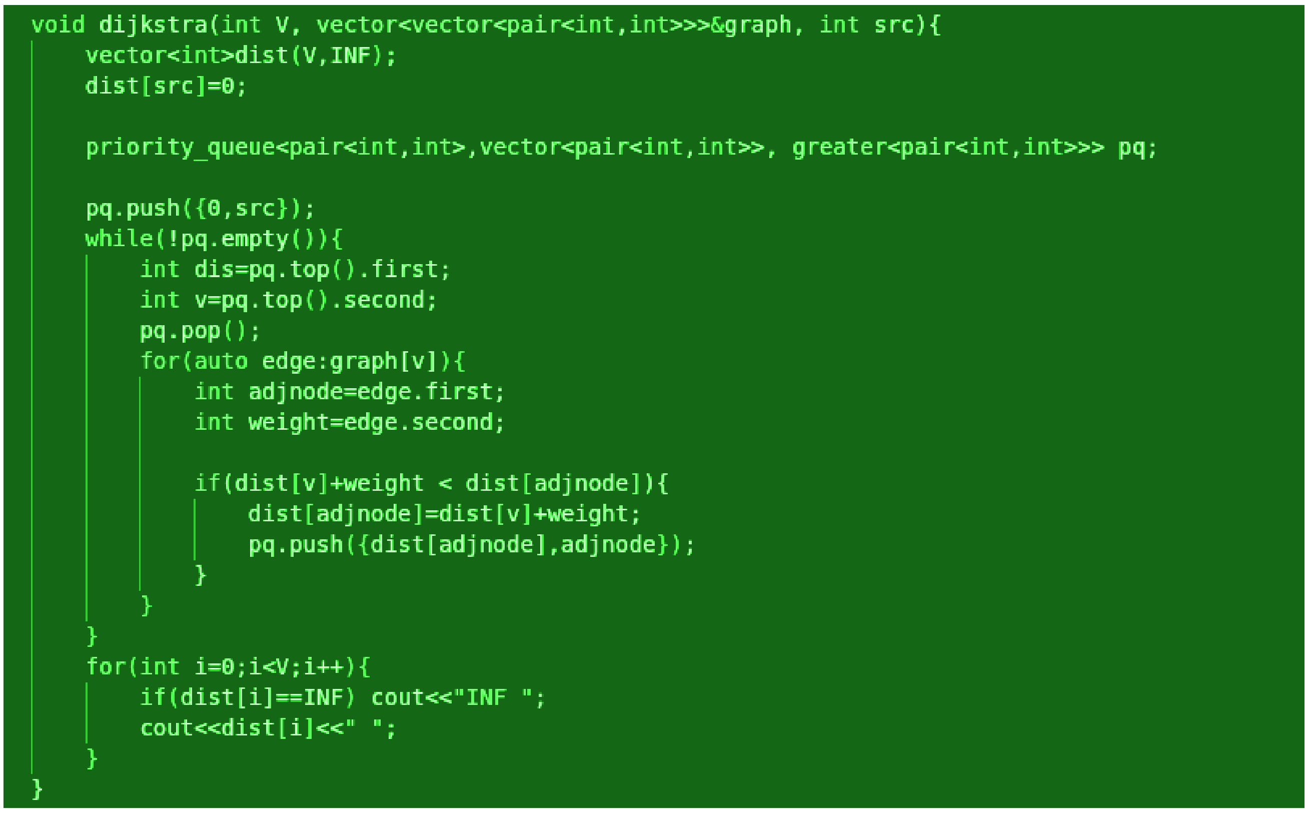
Shortest Path Algorithms: Dijkstra’s Algorithm Algorithm:

How it works:

1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.
5. Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.
6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.



CPP Code:



```cpp
void dijkstra(int V, vector<vector<pair<int,int>>>&graph, int src){
    vector<int>dist(V,INF);
    dist[src]=0;

    priority_queue<pair<int,int>,vector<pair<int,int>>, greater<pair<int,int>>> pq;

    pq.push({0,src});
    while(!pq.empty()){
        int dis=pq.top().first;
        int v=pq.top().second;
        pq.pop();
        for(auto edge:graph[v]){
            int adjnode=edge.first;
            int weight=edge.second;

            if(dist[v]+weight < dist[adjnode]){
                dist[adjnode]=dist[v]+weight;
                pq.push({dist[adjnode],adjnode});
            }
        }
    }
    for(int i=0;i<V;i++){
        if(dist[i]==INF) cout<<"INF ";
        cout<<dist[i]<<" ";
    }
}
```

Applications of Dijkstra’s Algorithm:

- Google maps uses Dijkstra algorithm to show shortest distance between source and destination.
- In computer networking , Dijkstra’s algorithm forms the basis for various routing protocols, such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System).
- Transportation and traffic management systems use Dijkstra’s algorithm to optimize traffic flow, minimize congestion, and plan the most efficient routes for vehicles.
- Airlines use Dijkstra’s algorithm to plan flight paths that minimize fuel consumption, reduce travel time.
- Dijkstra’s algorithm is applied in electronic design automation for routing connections on integrated circuits and very-large-scale integration (VLSI) chips.

4. Searching Algorithms:

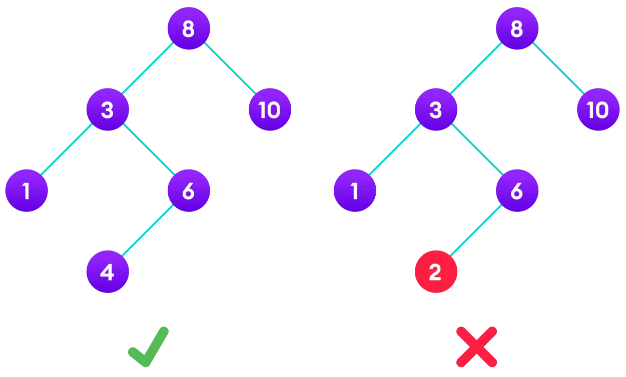
Binary Search Trees (BST):

A Binary Search Tree is a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value.

A clear advantage with Binary Search Trees is that operations like search, delete, and insert are fast and done without having to shift values in memory.

The properties that separate a binary search tree from a regular binary tree is

- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties



Algorithm: (Searching)

```
If root == NULL
        return NULL;
If number == root->data
        return root->data;
If number < root->data
        return search(root->left)
If number > root->data
        return search(root->right)
```

Time Complexity

- Best/Average Case (Balanced BST): O(log n) for insertion, deletion, and search.
- Worst Case (Skewed BST): O(n) if the tree is unbalanced.

Balanced Binary Search Trees: AVL Trees and Red-Black Trees:

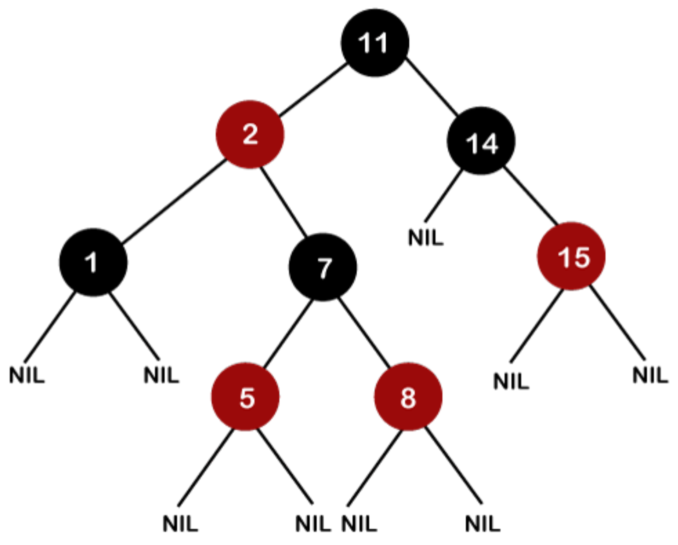
1. Red-Black Tree 🔴⚫

Properties:

- Each node is **Red** or **Black**.
- **Root is always Black**.
- No two consecutive **Red** nodes.
- Every path from root to leaves has the **same number of black nodes**.
- Height is **O(log n)** (at most **twice** the height of AVL).

Creation:

- Insert like a normal BST.
- Fix violations using **color flips** & **rotations**.



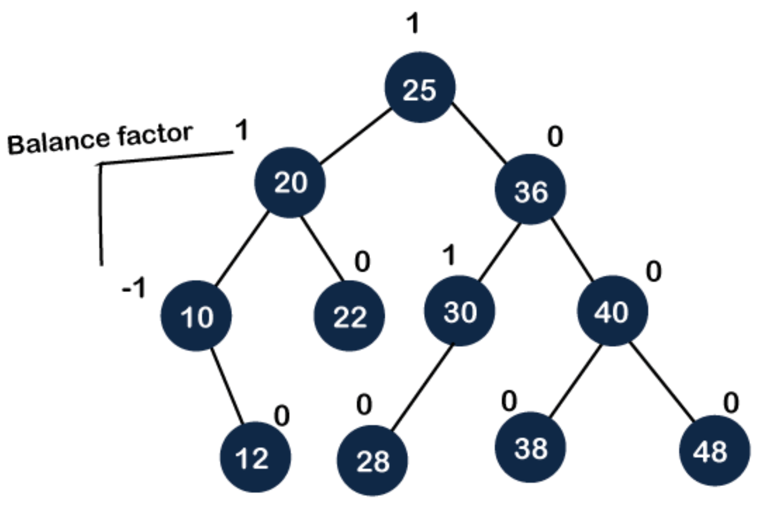
2. AVL Tree 🌲

Properties:

- Each node stores a **Balance Factor** (-1, 0, 1).
- If balance factor **exceeds 1 or -1**, rotations are applied.
- **Strictly balanced** → Ensures O(log n) height.

Creation:

- Insert like a BST.
- If imbalance occurs, apply **LL, RR, LR, or RL rotations**.



Click Here

B-Trees, Skip Lists, Hashing:

1. B-Trees 🌳

- **Self-balancing** m-ary search tree (generalized BST).
- Each node holds multiple keys; used in databases & filesystems.
- **Operations:** O(log n) (search, insert, delete).

2. Skip Lists 📑

- **Layered linked list** with random-level pointers for fast lookup.
- Probabilistic alternative to BSTs with O(log n) avg operations.
- Used in databases and concurrent applications.

3. Hashing#️⃣

- Maps keys to indices using a **hash function**.
- Supports **constant-time** (O(1)) average lookup in **hash tables**.
- **Collision handling:** Chaining (linked lists) & Open addressing (probing).

Priority Queues, Heaps, Interval Trees

1. Priority Queue ⏫

- Abstract data structure where elements have **priorities**.

- **Highest priority dequeued first** (min/max).
- Implemented using **Heaps** for O(log n) insert/delete.

2. Heaps 🏔️

- **Binary Heap**: Complete binary tree with heap property.
  - **Min-Heap**: Parent ≤ Children.
  - **Max-Heap**: Parent ≥ Children.
- Used in **Priority Queues, Heap Sort, Dijkstra's Algorithm**.

3. Interval Tree 📏

- **Augmented BST** storing intervals.
- Fast **overlapping interval queries** in O(log n).
- Used in **computational geometry, scheduling, genome analysis**.

5. Dynamic Programming

Click Here

Dynamic Programming is an optimization technique that improves recursive solutions by storing results of subproblems to reduce time.

Longest Common Subsequence (LCS):

The **Longest Common Subsequence (LCS)** problem finds the longest sequence that is common in two strings, maintaining the order but not necessarily consecutiveness.

Approach:

1. Create a 2D DP table `dp[i][j]` where `dp[i][j]` stores the LCS length of the first `i` characters of string X and the first `j` characters of string Y.
2. If `X[i-1] == Y[j-1]`, `dp[i][j] = dp[i-1][j-1] + 1`.
3. Else, `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`.

Example:

For `X = "ASHIK"` and `Y = "MISKATUL"`, the LCS is `"AIK"` with length `3`.

Code: cpp

```cpp
int LCS(string X, string Y) {
    int m = X.length(), n = Y.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            dp[i][j] = (X[i-1] == Y[j-1]) ? dp[i-1][j-1] + 1 :
max(dp[i-1][j], dp[i][j-1]);
    return dp[m][n];
}
```

Time Complexity: `O(m * n)`

Space Complexity: `O(m * n)`

Matrix Chain Multiplication (MCM):

Matrix Chain Multiplication (MCM) aims to find the optimal order of matrix multiplication to minimize scalar multiplications.

Steps:

1. **Define the DP Table**:

   Let `dp[i][j]` represent the minimum cost to multiply matrices from `Ai` to `Aj`.
2. **Recurrence**:

   $dp[i][j]=min(dp[i][k]+dp[k+1][j]+p[i-1]*p[k]*p[j])$

   Where `p[]` is the array of matrix dimensions.
3. **Base Case**:

   `dp[i][i] = 0` for all `i`.

Code:cpp

```cpp
int matrixChainOrder(vector<int>& p) {
    int n = p.size() - 1;
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int l = 2; l <= n; l++) {
        for (int i = 0; i < n - l + 1; i++) {
            int j = i + l - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + p[i] *
p[k+1] * p[j+1]);
            }
        }
    }
    return dp[0][n-1];
}
```

```
int main() {
    vector<int> p = {10, 20, 30, 40, 30};
    cout << matrixChainOrder(p) << endl;
    return 0;
}
```

Output: `30000`

Explanation:

The minimum scalar multiplications needed for matrices `[10, 20, 30, 40, 30]` is `30000`.

Time Complexity: `O(n^3)`
Space Complexity: `O(n^2)`

6. Greedy Algorithms

Greedy Algorithm approach:

A **Greedy Algorithm** is a problem-solving technique used to solve optimization problems by making the **locally optimal choice** at each step with the hope of finding the **global optimal solution**.

- It works by choosing the best option **available at the moment** without considering future consequences.
- The algorithm assumes that making **greedy choices** will lead to the **best overall solution**.

This technique is used to find a feasible solution that may or may not be optimal. A feasible solution is a subset that meets all the given criteria. The optimal solution is the best and most favorable one within that subset. If multiple solutions meet the criteria, they are considered feasible, but the optimal solution is the one that provides the best outcome.

Feasible Solution:

- A solution that satisfies all the constraints and conditions of the problem.
- There may be more than one feasible solution.

Optimal Solution:

- The best solution among all feasible solutions that gives the most favorable result.
- Usually, there is only one optimal solution.

Components of Greedy Algorithm:

1. **Candidate Set** – The possible options used to build the solution.
2. **Selection Function** – Chooses the best option to add to the solution.
3. **Feasibility Function** – Checks if the chosen option can be part of the solution.
4. **Objective Function** – Assigns a value to the current solution or partial solution.
5. **Solution Function** – Checks if the complete solution is achieved.

Pseudocode of Greedy Algorithm:

```
Algorithm Greedy(a, n)
{
    solution := ∅;    // Initialize solution as an empty set
    for i = 1 to n do
    {
        x := select(a);    // Select the next best candidate
        if feasible(solution, x) then
        {
            solution := solution ∪ {x};    // Add x to the solution if
feasible
        }
    }
    return solution;
}
```

Applications / Examples of Greedy Algorithm:

1. **Fractional Knapsack Problem**
2. **Job Scheduling Problem**
3. **Traveling Salesman Problem (TSP)**
4. **Minimum Spanning Tree (MST)** (e.g., using Prim's or Kruskal's algorithm)
5. **Graph Coloring**
6. **Huffman Coding** (used in data compression)

Advantages:

1. **Simple and easy to implement**
2. **Efficient for certain problems**
3. **Faster compared to other techniques like Dynamic Programming**
4. **Provides an optimal solution for some problems** (e.g., MST, Huffman Coding)

Disadvantages:

1. **Does not always guarantee the optimal solution**
    - For some problems, a **greedy approach may fail** (e.g., 0/1 Knapsack Problem).
2. **Short-sighted decisions**
    - The algorithm only considers the **current best choice**, which might not lead to the **global best solution**.

📝 How to Identify Greedy Algorithm Problems:

1. **Optimization Problem:**
    - If the problem asks for **maximum** or **minimum** of something, consider a greedy approach.
2. **Greedy Choice Property:**
    - A **locally optimal solution** should lead to a **globally optimal solution**.
3. **Optimal Substructure:**
    - A problem has an **optimal substructure** if an optimal solution to the entire problem can be constructed by combining optimal solutions of its subproblems.

Difference Between Greedy and Dynamic Programming:

1. **Greedy Algorithm** works if the problem has **Greedy Choice Property** (making the best local choice gives the global best) and **Optimal Substructure**.
   Dynamic Programming also needs **Optimal Substructure**, but it additionally requires **Overlapping Subproblems** (solving the same subproblems repeatedly).
2. In **Greedy Algorithm**, each **local decision** directly leads to the best **overall solution**.
   In **Dynamic Programming**, the solution to the **main problem** is built by solving and combining smaller, **overlapping subproblems**.

Activity Selection Problem:

The **Activity Selection Problem** is a classic example where a greedy approach is used. Given a set of activities with start and finish times, we need to select the maximum number of activities that don't overlap.

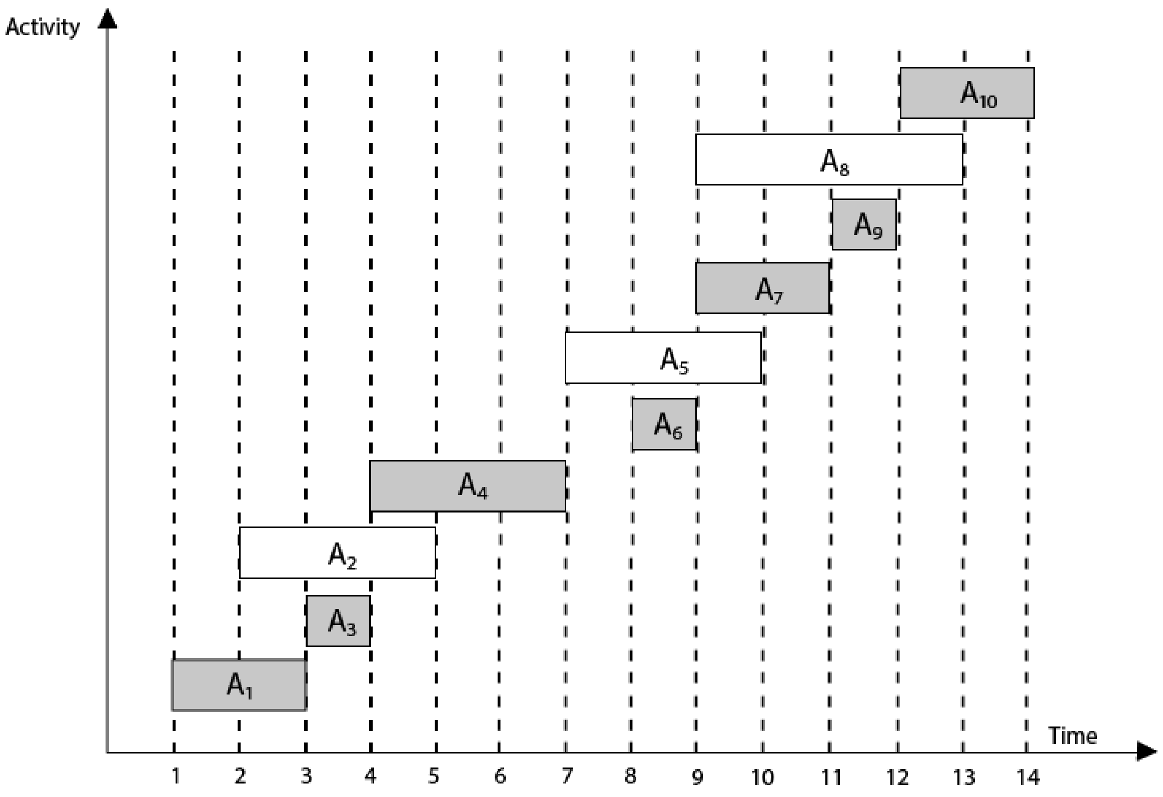
Greedy Approach:

- Sort activities by their finish times.
- Select the first activity, then select the next activity whose start time is after the finish time of the previously selected activity.

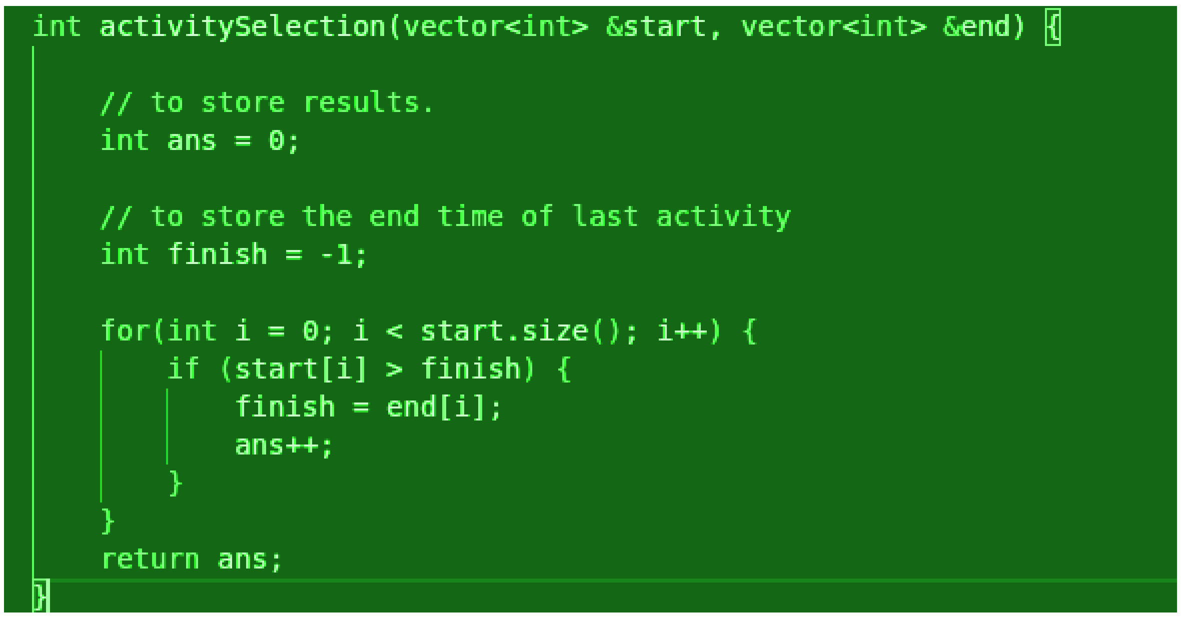
Example:
Start = (1,2,3,4,7,8,9,9,11,12)
finish = (3,5,4,7,10,9,11,13,12,14)

Arranging the activities in increasing order of end time:

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Code:CPP


```cpp
int activitySelection(vector<int> &start, vector<int> &end) {

    // to store results.
    int ans = 0;

    // to store the end time of last activity
    int finish = -1;

    for(int i = 0; i < start.size(); i++) {
        if (start[i] > finish) {
            finish = end[i];
            ans++;
        }
    }
    return ans;
}
```

Huffman Coding and its applications:

Huffman Coding is a lossless data compression algorithm. It assigns shorter binary codes to more frequent characters.

Huffman Coding is **greedy** because it **always picks the two lowest frequency characters first** to build the tree, ensuring the most optimal compression.

How It Works (Step-by-Step)

Suppose a string:

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C

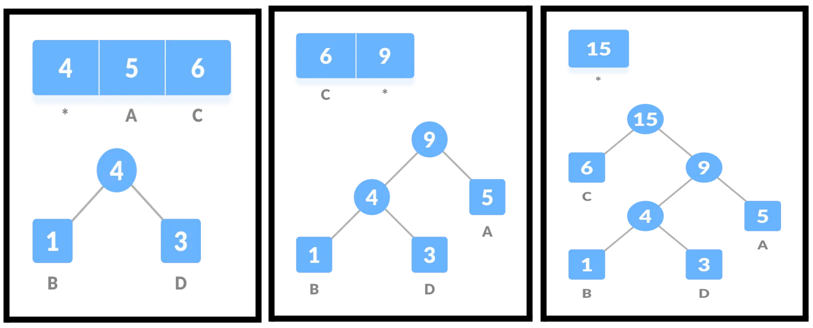
1. **Count Frequency** → Find how often each character appears.

1	6	5	3
B	C	A	D

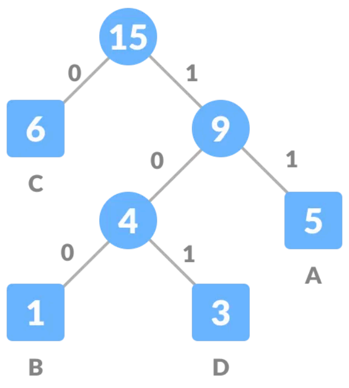
2. **Sort by Frequency** → Arrange characters in increasing order.

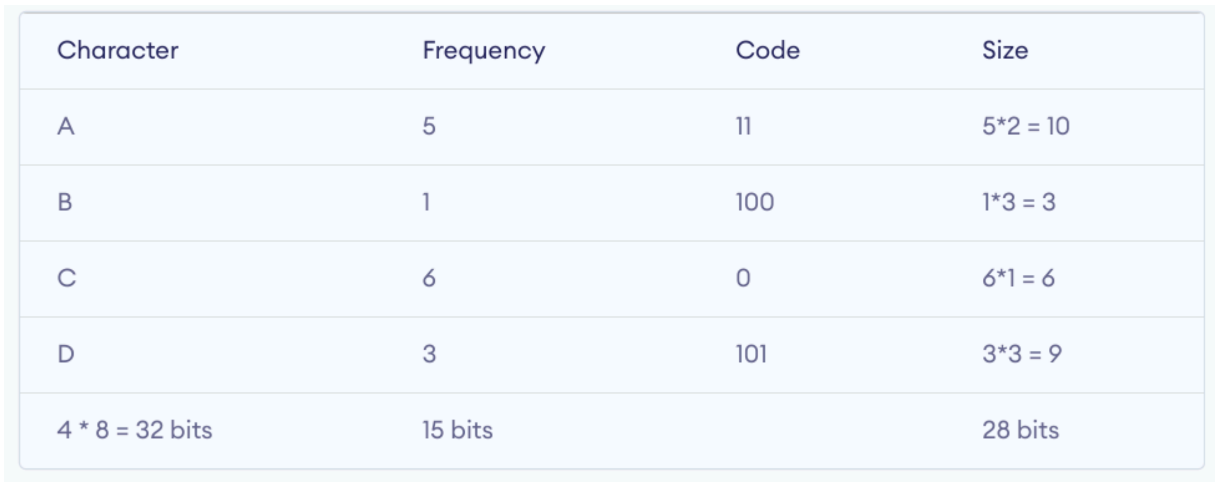
1	3	5	6
B	D	A	C

3. **Build Tree Greedily** →
    - Pick the two **smallest** frequencies.
    - Merge them into a new node (sum of both).
    - Repeat until one node remains (Huffman Tree).



4. **Assign Binary Codes** →
    - Left edge = `0`, Right edge = `1`.
    - More frequent characters get **shorter codes**, less frequent get **longer codes**.





Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Applications of Huffman Coding:

1. **File Compression** – Used in ZIP, GZIP, 7z, etc.
2. **Multimedia Encoding** – Used in JPEG, MP3, MPEG, etc.
3. **Efficient Data Transmission** – Reduces bandwidth usage.
4. **Text Encoding** – Used in character encoding schemes like ASCII optimization.

Traveling Salesperson Problem (TSP)

The **Travelling Salesperson Problem (TSP)** aims to find the shortest possible route that visits every city exactly once and returns to the starting point. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

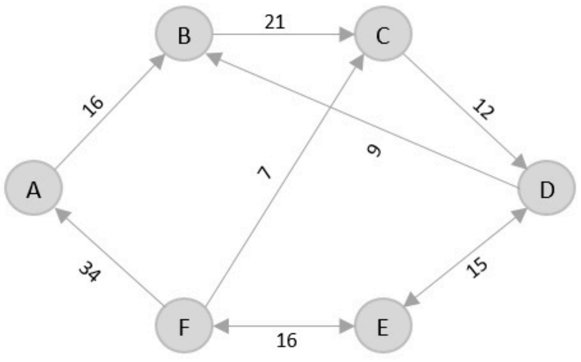
Algorithm:

1. **Sort all edges** in ascending order based on distance.

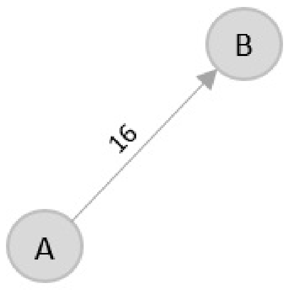
2. **Pick the shortest edge** that connects two cities, ensuring one of them is the starting point.
3. **Move to the next city** and choose the shortest available edge that doesn't form a cycle (unless it's the final return to the start).
4. **Continue** until all cities are visited exactly once and the path returns to the starting city.

Example:

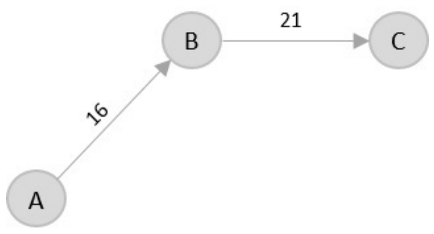
Consider the following graph with six cities and the distances between them −



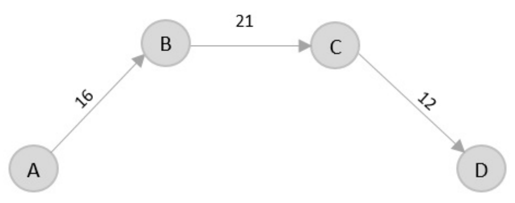
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



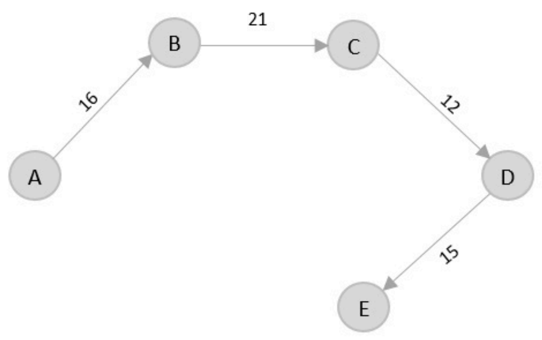
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



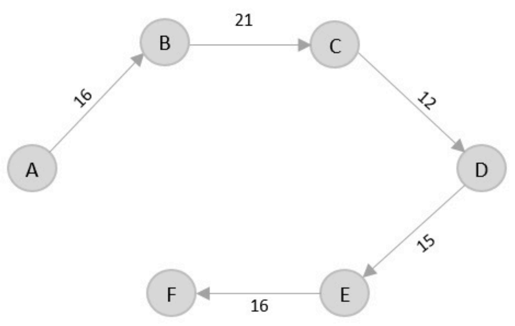
Theres only one edge between C → D, therefore it is added to the output graph.



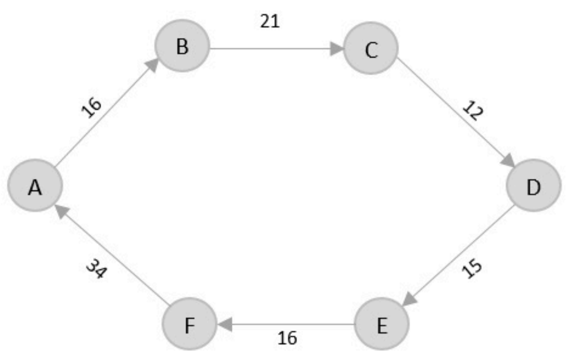
Theres two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.



Theres only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

<u>Knapsack Problem</u>

The **Knapsack Problem** is a classic optimization problem where we aim to maximize the total value of items placed in a knapsack without exceeding its weight capacity.

Types of Knapsack Problems:

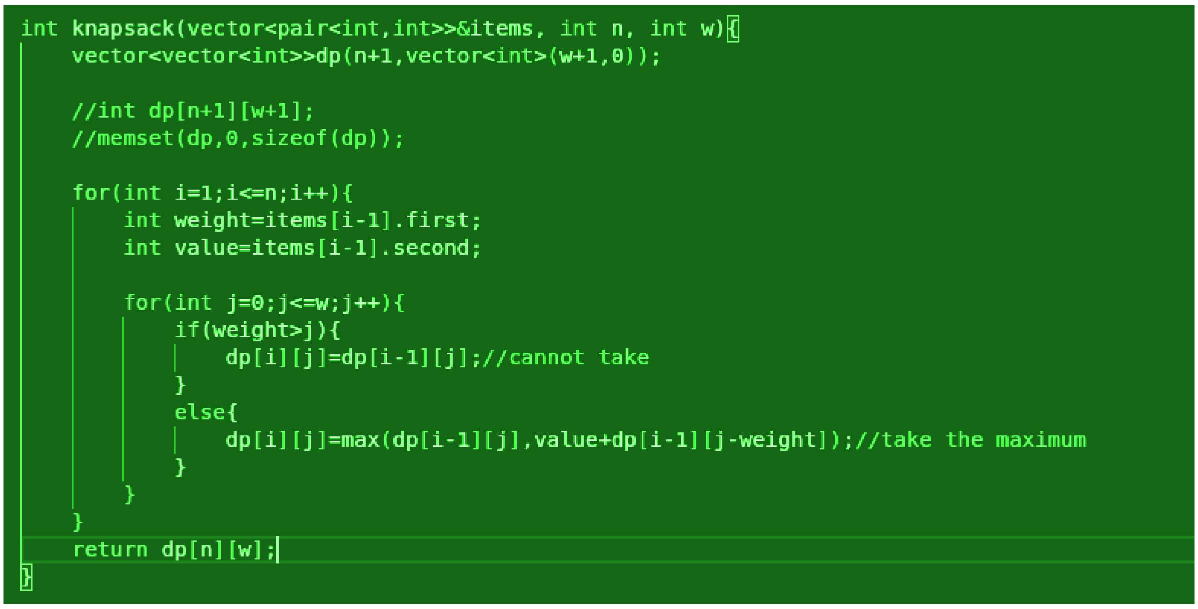
1. **0/1 Knapsack:** You either take an item or leave it (no splitting).
2. **Fractional Knapsack:** Items can be divided, meaning you can take a fraction of an item.

2. 0/1 Knapsack Problem (Dynamic Programming Approach):

<u>Algorithm:</u>

1. **Initialize** `dp[n+1][w+1]` with `0`.
2. **Loop through items (`i = 1` to `n`)**:
3. Get `weight` and `value` of the item.
4. **Loop through capacities (`j = 0` to `w`)**:
   a. If `weight > j`: **Skip item** → `dp[i][j] = dp[i-1][j]`.
   b. Else: **Choose max** of including or excluding the item:
      dp[i][j]=max(dp[i−1][j],value+dp[i−1][j−weight])
5. **Return `dp[n][w]`** → Maximum value possible.

Code : CPP



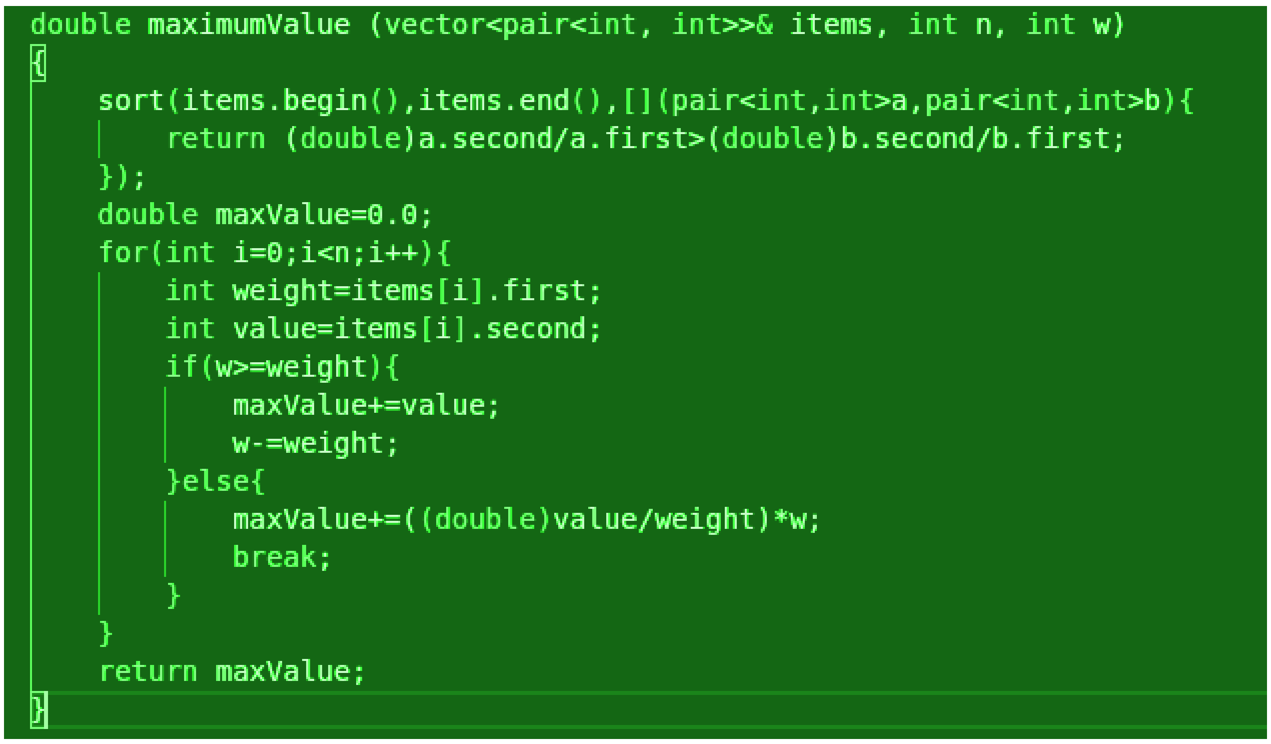
```cpp
int knapsack(vector<pair<int,int>>&items, int n, int w){
    vector<vector<int>>dp(n+1,vector<int>(w+1,0));

    //int dp[n+1][w+1];
    //memset(dp,0,sizeof(dp));

    for(int i=1;i<=n;i++){
        int weight=items[i-1].first;
        int value=items[i-1].second;

        for(int j=0;j<=w;j++){
            if(weight>j){
                dp[i][j]=dp[i-1][j];//cannot take
            }
            else{
                dp[i][j]=max(dp[i-1][j],value+dp[i-1][j-weight]);//take the maximum
            }
        }
    }
    return dp[n][w];
}
```

2.Fractional Knapsack Problem (Greedy Approach):

Algorithm:

1. **Sort Items by Value-to-Weight Ratio:**
   - Sort the `items` in **descending order** based on **value/weight** ratio.
   - This ensures that the most valuable items per unit weight are chosen first.
2. **Initialize `maxValue` to 0.0.**
3. **Iterate Through the Sorted Items:**
   - For each item:
     - Extract **weight (`weight`)** and **value (`value`)**.
     - If the **remaining capacity (`w`) is greater than or equal to `weight`**, take the **full item**:
       - Add `value` to `maxValue`.
       - Decrease `w` by `weight`.
     - Else, take a **fraction of the item**:
       - Add `((double)value / weight) * w` to `maxValue`.
       - Break the loop (knapsack is full).
4. **Return `maxValue` as the final maximum total value of the knapsack.**

Code : CPP

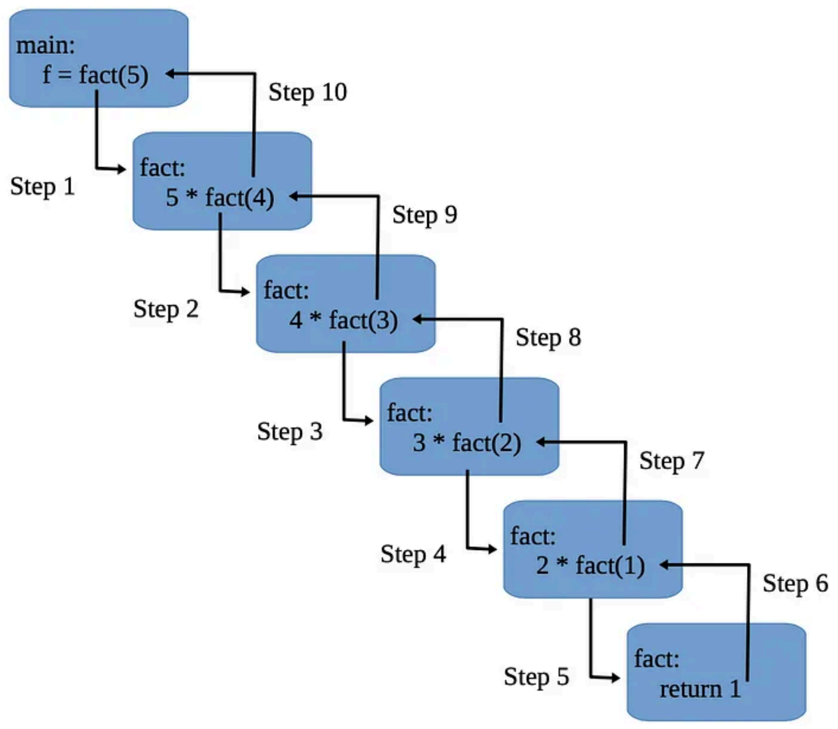


```cpp
double maximumValue (vector<pair<int, int>>& items, int n, int w)
{
    sort(items.begin(),items.end(),[](pair<int,int>a,pair<int,int>b){
        return (double)a.second/a.first>(double)b.second/b.first;
    });
    double maxValue=0.0;
    for(int i=0;i<n;i++){
        int weight=items[i].first;
        int value=items[i].second;
        if(w>=weight){
            maxValue+=value;
            w-=weight;
        }else{
            maxValue+=((double)value/weight)*w;
            break;
        }
    }
    return maxValue;
}
```

7. Recurrences and Backtracking

Recurrences:

Recursion is when a function calls itself to solve a problem.



Backtracking is a way to solve problems by trying different solutions and "backing up" when you hit a dead-end. It helps in finding the right solution by exploring all possible choices, but if one path doesn't work, it "backtracks" to try another path.

How Backtracking Works:

1. **Try a Solution**: You make a choice and continue solving the problem.
2. **Check if the Solution is Valid**: If it's valid, continue; if not, backtrack.
3. **Backtrack if Needed**: If you can't solve with the current choice, undo the choice and try a different one.
4. **Repeat** until the problem is solved.

Example Problems:

- N-Queen Problem
- Sudoku Solver
- Hamiltonian Path
- Subset Sum

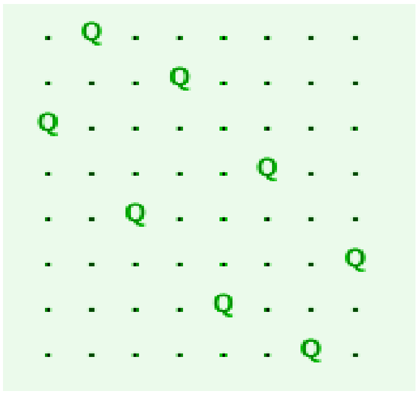
<u>Backtracking: n-Queens Problem:</u>

The **n-Queens Problem** is a classic combinatorial problem that involves placing n queens on an n×n chessboard so that no two queens attack each other. This means:

1. No two queens can be in the same row.
2. No two queens can be in the same column.
3. No two queens can be in the same diagonal.

Solution Approach:

- **Backtracking** is commonly used to solve this problem.
- Place queens row by row:
  - Start with the first row and place a queen in a valid column.
  - Move to the next row and place another queen in a column that doesn't conflict with previously placed queens.
  - If no valid position exists in the current row, backtrack to the previous row and move the queen to a different column.
- Repeat this process until all queens are placed.



```
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . Q . .
. . Q . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
```

<u>NP-Hard and NP-Complete Problems:</u>

Completeness of an algorithmic problem refers to whether the algorithm provides a solution for every instance of the problem within its domain of definition.

1. P (Polynomial-Time)

- **Definition**: Problems that can be solved in polynomial time by a deterministic algorithm.
- **Completeness**: For problems in P, a complete algorithm exists that can always solve the problem efficiently.
- **Example**:
  - **Shortest Path Problem**: Dijkstra's algorithm solves it in O(V^2) or O(ElogV) for graphs with non-negative weights.

2. NP (Nondeterministic Polynomial-Time)

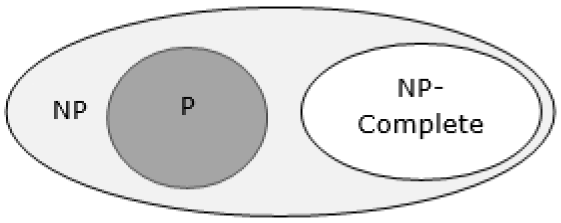
- **Definition**: Problems whose solutions can be verified in polynomial time, even if finding the solution might take longer.
- **Completeness**: An algorithm may exist to solve such problems, but it might not run efficiently for large inputs.
- **Example**:
  - **Traveling Salesman Problem (TSP)**: Given a list of cities and distances, find the shortest route visiting all cities.
  - While verifying a given route is polynomial, finding the optimal route is computationally expensive.

3. NP-Hard Problems

- **Definition**: Problems as hard as the hardest problems in NP. They do not have to belong to NP (e.g., optimization problems).
- **Completeness**: No polynomial-time algorithm is known, and solving them efficiently would solve all NP problems.
- **Example**:
  - **Halting Problem**: Deciding if a program halts on a given input.
  - Not verifiable in polynomial time, so not in NP, but it's at least as hard as any NP-problem.

4. NP-Complete Problems

- **Definition**: Problems that are both in NP and NP−Hard.
- **Completeness**: If a polynomial-time algorithm is found for any NP-Complete problem, all NP problems can be solved in polynomial time.
- **Example**:
  - **SAT Problem (Boolean Satisfiability)**: Determine if there exists an assignment of variables such that a given Boolean formula evaluates to true.
  - It was the first problem proved to be NP-Complete (Cook-Levin theorem).



Key Relationships

- P⊆N: All problems in P are also in PN.
- NP Complete problems are the "hardest" in NP.
- Proving P=NP or P != NP is one of the biggest unsolved problems in computer science.

Complexity Class	Characteristic Feature
P	Easily solvable in polynomial time.
NP	Answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and take a long time to check.
NP-complete	A problem that is both NP and NP-hard.

Branch and Bound:

The Branch and Bound Algorithm is a method used in combinatorial optimization problems to systematically search for the best solution. It works by dividing the problem into smaller subproblems, or branches, and then eliminating certain branches based on bounds on the optimal solution. This process continues until the best solution is found or all branches have been explored. Branch and Bound is commonly used in problems like the traveling salesman and job scheduling.

A branch and bound algorithm provide an optimal solution to an NP-Hard problem by exploring the entire search space. Through the exploration of the entire search space, a branch and bound algorithm identify possible candidates for solutions step-by-step.

Key Concepts:

- Branching: The process of dividing the problem into smaller subproblems (branches). Each branch represents a different choice or decision in the solution space.
- Bounding: For each subproblem, an upper or lower bound (depending on whether we are maximizing or minimizing) is calculated. This bound helps determine if the current subproblem is promising or if it should be discarded.
- Pruning: If the bound of a subproblem indicates that it cannot lead to a better solution than what has already been found, it is pruned (i.e., discarded) from further exploration.
- Solution Space Tree: The problem is represented as a tree structure where the root node represents the initial problem, and the child nodes represent subproblems created by branching.

Steps of the Branch and Bound Algorithm:

1. Start at the root node, representing the initial problem.
2. Branch: Divide the problem into smaller subproblems.
3. Bound: Calculate an upper or lower bound for each subproblem. If the bound suggests the subproblem cannot lead to a better solution, prune it.
4. Explore the best solution: Choose the subproblem with the best bound and continue solving it.
5. Stop when all subproblems are either solved or pruned.

8. NP-Completeness and Reducibility

Lower Bound Theory:

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Lower Bound Theory uses a number of methods/techniques to find out the lower bound.

Aim: The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Techniques:

The techniques which are used by lower Bound Theory are:

1. **Comparison Trees** → Used for sorting and searching problems.
2. **Oracle and Adversary Argument** → Used for proving lower bounds dynamically.
3. **State Space Method** → Used in problems with complex state transitions.

Method 1. Comparison trees:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence (a1; a2……an).

Given ai,aj from (a1, a2…..an) We Perform One of the Comparisons

- ai < aj less than
- ai ≤ aj less than or equal to
- ai > aj greater than
- ai ≥ aj greater than or equal to
- ai = aj equal to

To determine their relative order, if we assume all elements are distinct, then we just need to consider ai ≤ aj '=' is excluded &, ≥,≤,>,< are equivalent.

Consider sorting three numbers a1, a2, and a3. There are 3! = 6 possible combinations: 1. (a1, a2, a3), (a1, a3, a2), 2. (a2, a1, a3), (a2, a3, a1) 3. (a3, a1, a2), (a3, a2, a1) The Comparison based algorithm defines a decision tree.

Decision Tree: A decision tree is a full binary tree that shows the comparisons between elements that are executed by an appropriate sorting algorithm operating on an input of a given size. Control, data movement, and all other conditions of the algorithm are ignored.

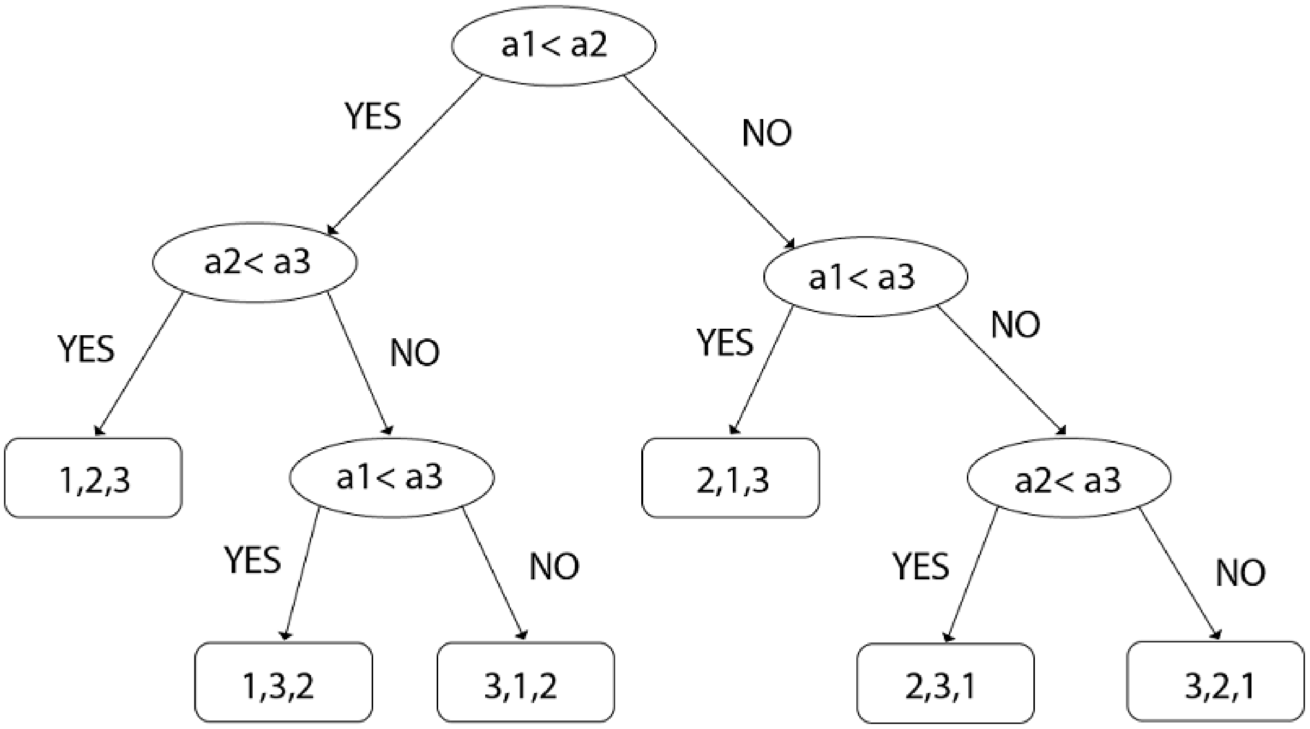
In a decision tree, there will be an array of length n.

So, total leaves will be n! (I.e. total number of comparisons)

If tree height is h, then surely

n! ≤2n (tree will be binary)

Taking an Example of comparing a1, a2, and a3.
Left subtree will be true condition i.e. ai ≤ aj
Right subtree will be false condition i.e. ai >aj



So from above, we got N! ≤2n

Taking Log both sides

Log n! ≤ h log 2

h log 2≥log n!

h ≥ log2 n!

h≥log 2 [1, 2, 3……n]

h ≥$\log_2 1 + \log_2 2 + \log_2 3 + \cdots \log_2 n$

h ≥ $\sum_{i=1}^{n} \log_2 i$

h ≥$\int_i^n \log_2 i - 1 \, d^i$

h ≥$\log_2 i . x^o \int_1^n - \int_1^n \frac{1}{i} xi \, di$

h ≥ $n\log_2 n - \int_1^n 1 \, di$

h ≥ $n\log_2 n - i \int_1^n$

h≥ $n \log_2 n - n + 1$

Ignoring the Constant terms

h≥ $n\log_2 n$

h=π n (log n)

1. Comparison Trees

A **comparison tree** is a decision tree representation of an algorithm where each internal node represents a comparison between two elements, and each leaf represents a possible sorted order.

- It helps determine the minimum number of comparisons needed for sorting or searching problems.
- The depth of the decision tree provides a lower bound on the worst-case number of comparisons.
- Example: In **comparison-based sorting**, the lower bound is **Ω(n log n)**, as derived using decision trees.

2. Oracle and Adversary Argument

This method involves an **adversary** (or oracle) that tries to make the algorithm perform as many operations as possible by controlling the input dynamically.

- The adversary provides answers to comparisons in such a way that forces the algorithm to make the maximum number of steps.
- It is commonly used in **searching** and **decision problems** to prove lower bounds.
- Example: In the **element uniqueness problem**, an adversary can provide inputs that require **Ω(n log n)** comparisons.

3. State Space Method

This method analyzes the number of **distinct states** a problem can be in and determines the number of operations required to transition between them.

- It involves modeling the problem as a state transition graph and computing the minimum number of steps to reach the final state.
- It is particularly useful in **graph algorithms**, **dynamic programming**, and **backtracking problems**.
- Example: In **sorting**, the number of possible input permutations (states) is **n!**, leading to a lower bound of **Ω(n log n)** using information theory.

<u>Discussion of NP-Complete problems: Satisfiability, Clique, Vertex Cover, Independent Set, Hamiltonian Cycle, TSP, Knapsack, Set Cover, Bin Packing:</u>

NP-Complete Problems

NP-Complete problems are **hard problems** that don't have a quick solution, but if we somehow find an answer, we can check it **fast**. If we solve one NP-Complete problem quickly, we can solve all of them quickly!

1. Satisfiability (SAT)

- **Problem:** Given a Boolean formula in conjunctive normal form (CNF), determine if there exists an assignment of variables that satisfies all clauses.
- **Example:** $(A \lor B) \land (\neg A \lor C) \land (B \lor \neg C)$
- **Significance:** The first problem proven to be NP-Complete (**Cook's Theorem**).
- **Variants:**
  - **3-SAT:** Each clause has at most 3 literals (still NP-Complete).
  - **2-SAT:** Each clause has at most 2 literals (solvable in polynomial time).

2. Clique Problem

- **Problem:** Given an undirected graph G=(V,E)G = (V, E)G=(V,E) and an integer kkk, determine if there exists a **clique** (a complete subgraph) of size **at least** kkk.
- **Example:** Finding a fully connected subgraph of size kkk in a social network.
- **Application:** Used in bioinformatics, network analysis, and social media clustering.

3. Vertex Cover

- **Problem:** Given a graph G=(V,E)G = (V, E)G=(V,E) and an integer kkk, determine if there exists a set of at most kkk vertices such that every edge in EEE has at least one endpoint in this set.
- **Example:** Finding the minimum number of guards required to cover all entrances of a museum.
- **Relation:** Closely related to **Clique** (complement of each other).

4. Independent Set

- **Problem:** Given a graph G=(V,E)G = (V, E)G=(V,E) and an integer kkk, determine if there exists an **independent set** of size at least kkk (a set of vertices such that no two are connected by an edge).

- **Relation: Independent Set** is the complement of **Vertex Cover** (if one is small, the other is large).

5. Hamiltonian Cycle

- **Problem:** Given a graph G=(V,E) determine if there exists a cycle that visits every vertex exactly once.
- **Example:** Can a delivery truck visit every city exactly once before returning to the starting point?
- **Relation:** Closely related to **Traveling Salesman Problem (TSP)**.

6. Traveling Salesman Problem (TSP)

- **Problem:** Given a set of cities and distances between them, find the shortest possible tour that visits each city exactly once and returns to the starting point.
- **Variants:**
  - **Decision Version (NP-Complete):** Is there a tour with cost ≤ k?
  - **Optimization Version (NP-Hard):** Find the shortest tour.
- **Applications:** Logistics, route planning, circuit design.

7. Knapsack Problem

- **Problem:** Given nnn items, each with weight wiw_iwi and value viv_ivi, and a knapsack with weight capacity WWW, determine the maximum value that can be carried.
- **Variants:**
  - **0/1 Knapsack:** Each item can be included **at most once** (NP-Complete).
  - **Fractional Knapsack:** Items can be **divided** (solvable in polynomial time using Greedy approach).

- **Applications:** Budget allocation, resource management.

---

8. Set Cover Problem

- **Problem:** Given a universe UUU and a collection of subsets S1,S2,...SmS_1, S_2, ... S_mS1,S2,...Sm, find the smallest number of subsets whose union covers all of UUU.
- **Example:** Finding the minimal number of security cameras to cover an entire building.

- **Approximation:** It has a **logarithmic approximation** algorithm but no polynomial exact solution.

9. Bin Packing Problem

- **Problem:** Given a set of items with different sizes and a fixed number of bins, determine the minimum number of bins needed to fit all items.
- **Example:** Packing boxes efficiently in a truck.
- **Variants:**
    - **1D Bin Packing:** Items have only **one dimension** (e.g., weight).
    - **2D/3D Bin Packing:** Items have **multiple dimensions** (e.g., width, height).
- **Approximation Algorithms:** Best Fit, First Fit, Next Fit.

9. Computational Geometry

Computational geometry is a branch of computer science that deals with solving problems involving geometric objects (points, lines, polygons) using algorithms. Most problems are solved in two dimensions (2D), like finding intersections or calculating the convex hull.

Line Segment Properties

1. Is a segment clockwise or counterclockwise from another that shares an endpoint? To determine the orientation of one segment with respect to another that shares an endpoint, we use the cross product of the two vectors formed by these segments. The sign of the cross product indicates the direction (clockwise or counterclockwise).

2. Which way do we turn when traversing two adjoining line segments? When traversing two adjoining line segments, you can determine the direction of the turn (left or right) by calculating the cross product of the vectors formed by these segments.

3. Do two line segments intersect? To check if two line segments intersect, you can compute the line equation for each segment. If the intersection point of these lines lies within both segments, they intersect.

Cross Product of Two Vectors (P1 x P2)

The cross product is a fundamental operation to determine the orientation of two vectors. For vectors P1 = (X1, Y1) and P2 = (X2, Y2), the cross product is:

$$P1 \times P2 = \det \begin{pmatrix} X_1 & X_2 \\ Y_1 & Y_2 \end{pmatrix} = X1 \cdot Y2 - X2 \cdot Y1$$

The cross product tells us the relative orientation of two vectors:

- If P1×P2>0, P1 is clockwise from P2 relative to the origin.
- If P1×P2>0, P1 is counterclockwise from P2 relative to the origin.
- If P1×P2=0, the vectors are collinear, pointing in the same or opposite directions.

Determining the Turn Direction Between Two Consecutive Line Segments

To determine whether two consecutive line segments, P0P1 and P1P2, turn left or right at P1, we use the cross product. Specifically, compute:

(P2−P0)×(P1−P0)

- If the cross product is positive, the turn is clockwise (right turn).
- If the cross product is negative, the turn is counterclockwise (left turn).
- If the cross product is zero, the points P0, P1, and P2 are collinear.

Determining if Two Line Segments Intersect

A common method to check if two line segments intersect involves:

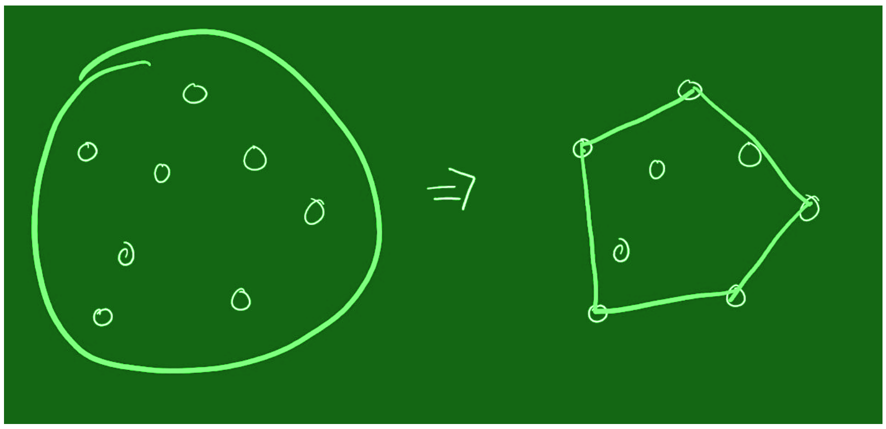
1. **Finding the line equations** for both segments, typically in the form y=mx+b, where mmm is the slope and b is the y-intercept.
2. **Finding the point of intersection (POI)** between these two lines. You can solve for the point by setting the two line equations equal.
3. **Checking if the intersection point lies within the bounds of both segments**. This ensures that the intersection is on both segments, not just the infinite lines extending from them.

Convex Hull

The Convex Hull is the subset of points that forms the smallest convex polygon which encloses all points in the set.

The **Convex Hull Algorithm** is used to find the convex hull of a set of points in computational geometry. The convex hull is the smallest convex set that encloses all the points, forming a convex polygon. This algorithm has many applications, including image processing, route planning, and object modeling.

To visualize this, imagine that each point is a pole. Then, imagine what happens if you were to wrap a rope around the outside of all the poles, and then pull infinitely hard, such that the connections between any two points that lie on the edge of the rope are lines. The set of points that touch the rope is the convex hull.



Graham Scan Algorithm

The algorithm finds all vertices of the convex hull ordered along its boundary. It uses a stack to detect and remove concavities in the boundary efficiently.

The Graham Scan algorithm for finding the convex hull of a set of points follows a three-step process:

Step 1: Sorting

The first step is to sort the points by their counterclockwise angle around a pivot point, denoted as P0P_0P0. The pivot is the leftmost point, and in case of a tie, the bottommost point is chosen. The sorting also breaks ties by the distance to the pivot.

Step 2: Maintaining a Stack

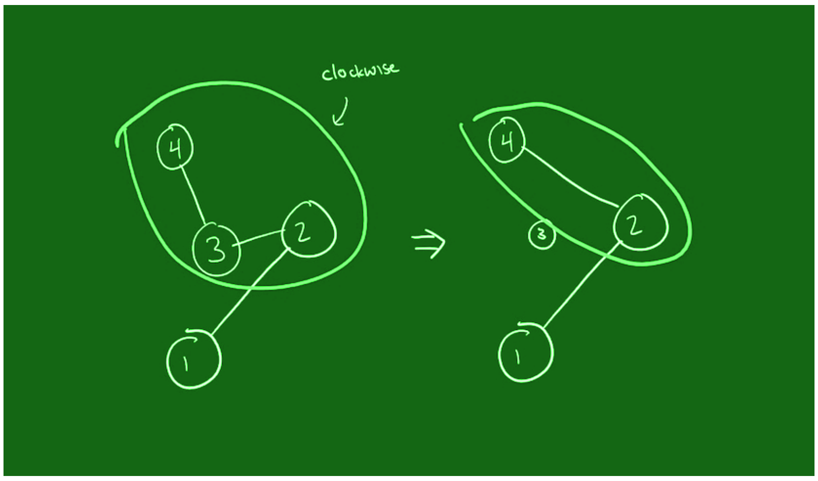
Next, we maintain a stack, referred to as `hull`, that holds the points of the convex hull. The key invariant here is that for any three consecutive points aaa, bbb, and ccc in the stack, the turn formed by the triplet a→b→ca \rightarrow b \rightarrow ca→b→c should be counterclockwise. This means that the point ccc lies to the left of the line from aaa to bbb, ensuring that the points in the stack form the vertices of a convex polygon.

Step 3: Constructing the Convex Hull

We begin by selecting the pivot point P0P_0P0 as the first point and the next point based on the sorting step. Then we iterate through the remaining points in the sorted list (denoted as `cand`), attempting to add each candidate point to the stack.

- Denote the current stack as `hull`, where the top element is hull[i], and the current candidate point as cand[j].
- Before adding cand[j] to the stack, we check whether the triplet hull[i−1]→hull[i]→cand[j] forms a counterclockwise turn.
    - If it does, we add cand[j] to the stack and continue with the next candidate point cand[j+1].
    - If not, the point hull[i] lies within the convex hull of the other points in the stack with cand[j], so we pop hull[i] from the stack and continue with cand[j].

The process continues until all the candidate points have been processed, and the points in the stack form the convex hull of the set of points.



Example:

To find the convex hull of the points (1,2),(2,3),(5,3),(3,2),(2,0),(6,2) using the Graham Scan algorithm, we proceed as follows:

Step 1: Sorting the Points

The first step is to sort the points by their counterclockwise angle around the leftmost point, which is (1,2) In case of ties in the angle, we break them by distance from the pivot point.

- The sorted points, based on their counterclockwise angle, are: (2,0),(6,2),(3,2),(5,3),(2,3) with (1,2) as the initial pivot. Note that the points (6,2)(and (3,2) are collinear, but since (6,2) s farther away, it comes before (3,2)(3,2)(3,2) in the sorted list.

Step 2: Maintaining the Stack

We start with an initial stack containing the pivot point (1,2)(1,2)(1,2) and the first sorted point (2,0)(2,0)(2,0). The stack is now:

stack=[(1,2),(2,0)]

We then process each remaining point from the sorted list:

- **Processing point (3,2):** We check if the turn formed by the points (1,2) ,(2,0), and (3,2) is counterclockwise. It is, so we add (3,2) to the stack. The stack now is:
  stack=[(1,2),(2,0),(3,2)]
- **Processing point (6,2):** We check if the turn formed by the points (2,0), (3,2), and (6,2) is counterclockwise. It is not (the turn is clockwise). To restore the counterclockwise property, we pop the last point (3,2) from the stack. After popping, we check the turn formed by (1,2), (2,0) and (6,2), which is counterclockwise. Thus, we add (6,2) to the stack. The stack now is:
  stack=[(1,2),(2,0),(6,2)]
- **Processing point (5,3):** We check if the turn formed by the points (2,0), (6,2), and (5,3) s counterclockwise. It is, so we add (5,3) to the stack. The stack now is:
  stack=[(1,2),(2,0),(6,2),(5,3)]
- **Processing point (2,3):** We check if the turn formed by the points (6,2), (5,3), and (2,3) is counterclockwise. It is, so we add (2,3)(2,3)(2,3) to the stack. The stack now is:
  stack=[(1,2),(2,0),(6,2),(5,3),(2,3)]

Step 3: Completing the Convex Hull

We have now processed all the points, and the stack contains the vertices of the convex hull in counterclockwise order:

Convex Hull=[(1,2),(2,0),(6,2),(5,3),(2,3)]

Thus, the convex hull of the points (1,2),(2,3),(5,3),(3,2),(2,0),(6,2) is formed by the points (1,2),(2,0),(6,2),(5,3),(2,3)

An animation of how it works(Click to watch)

The End

Made By:

MD. ASHIKUZZ ZAMAN

ID; 220119