**ASSIGNMENT-01**

**🔹 Problem Overview:**

Create a **menu-driven program** to manage user records on a telecom system or similar platform. The operations include:

* Accepting user data.
* Sorting records using **Heap Sort** and **Quick Sort**.
* Searching records using **Linear Search**, **Binary Search (Iterative)**, and **Binary Search (Recursive)**.

**🔹 Concepts Used:**

* **Structures** (for holding record data).
* **Vectors** (dynamic arrays).
* **Heap Sort** (comparison-based max-heap sorting).
* **Quick Sort** (divide-and-conquer strategy).
* **Binary Search** (logarithmic search on sorted data).
* **Linear Search** (sequential search).
* **Recursion** (used in binary search and quick sort).
* **STL Formatting** (setw, setprecision for output).

**🔹 Detailed Explanation of Main Logic Blocks:**

**1. Heap Sort – heapify() + heapSort()**

cpp

CopyEdit

void heapify(vector<Record>& d, int n, int i, const string& key)

✅ **Purpose**: Maintains the heap property at node i.

* Calculates left l = 2i + 1 and right r = 2i + 2 children.
* Compares current node with its children based on the key (bill\_amt or mobile\_number).
* If either child is greater than the current, swap and recursively heapify.

cpp

CopyEdit

void heapSort(vector<Record>& d, const string& key)

✅ **Purpose**: Converts array into a heap and performs sorting.

* First builds a max heap.
* Then repeatedly swaps the max element to the end and re-heapifies.

**2. Quick Sort – partition() + quickSort()**

cpp

CopyEdit

int partition(vector<Record>& d, int low, int high, const string& key)

✅ **Purpose**: Selects a pivot (last element), partitions array around it.

* All smaller (or equal) elements to the left.
* All greater elements to the right.

cpp

CopyEdit

void quickSort(vector<Record>& d, int low, int high, const string& key)

✅ **Purpose**: Recursive QuickSort using the pivot index returned by partition().

**3. Binary Search (Iterative & Recursive)**

cpp

CopyEdit

int binarySearch(const vector<Record>& d, const string& key, const string& value)

✅ Classic iterative approach on sorted vector:

* Compare middle element to value.
* Adjust search range accordingly.

cpp

CopyEdit

int binarySearchRecursive(const vector<Record>& d, const string& value, int low, int high)

✅ Recursive version of binary search.

**🔹 Greedy Strategy Justification:**

🟡 *Not applicable directly*.  
This is not a greedy algorithm implementation but rather **comparison-based sorting** and **search** operations.

**🔹 Common Viva Questions & Answers:**

**Q1. What is the time complexity of Heap Sort and Quick Sort?**

* **Heap Sort**:
  + Building heap: O(n)
  + Sorting: O(n log n)
* **Quick Sort**:
  + Average: O(n log n)
  + Worst: O(n²) (when the pivot is smallest/largest every time)

**Q2. Why do we sort before binary search?**

* Binary Search requires a **sorted array** to work correctly. If not sorted, it may give wrong answers or not terminate correctly.

**Q3. How is recursion used in your code?**

* For **recursive binary search** and **quick sort**.
* Each recursive call solves a subproblem, narrowing the input range.

**Q4. What's the difference between linear and binary search?**

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Sorted Input Needed | ❌ No | ✅ Yes |
| Approach | Sequential | Divide & Conquer |

**Q5. How does the partition function work?**

* It places the **pivot** in its correct position.
* All elements < pivot go left, ≥ pivot go right.

**Q6. What happens if cin >> ws; is not used before getline()?**

* getline() may read the leftover newline character (\n) from previous cin and terminate immediately. ws clears that.

**🔹 Time & Space Complexities Summary:**

| **Operation** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Heap Sort | O(n log n) | O(1) |
| Quick Sort (avg) | O(n log n) | O(log n) |
| Linear Search | O(n) | O(1) |
| Binary Search (Iter.) | O(log n) | O(1) |
| Binary Search (Recur.) | O(log n) | O(log n) |

**ASSIGNMENT-02**

**💡 Core Concepts Behind the Code**

**🔗 1. Graph Representation**

In your program, a **graph** models a **social network**, where:

* **Vertices** (users) represent **individual people**
* **Edges** (friendships) show **bidirectional relationships**

**▶️ Why use Adjacency List?**

cpp

CopyEdit

vector<vector<int>> adjList;

* It’s memory-efficient for **sparse graphs** (like real-world networks).
* Each index represents a user, and the nested vector holds all their friends.

**🚧 2. Building the Graph**

cpp

CopyEdit

void addEdge(int u, int v) {

adjList[u].push\_back(v);

adjList[v].push\_back(u);

}

* This function **adds an undirected edge** between user u and v.
* Since friendships are **mutual**, we add both u → v and v → u.

**🔍 3. Breadth-First Search (BFS)**

**🧠 Concept:**

* BFS explores **level by level** — like **wavefront expansion**
* Good for finding shortest paths, minimum hops, etc.

**🛠️ Implementation Breakdown:**

cpp

CopyEdit

vector<bool> visited(vertices, false);

queue<int> q;

* visited: Tracks whether a user has already been visited
* q: Standard queue for BFS logic

cpp

CopyEdit

visited[start] = true;

q.push(start);

* Start from a user → mark them visited and enqueue them

cpp

CopyEdit

while (!q.empty()) {

int user = q.front(); q.pop();

cout << user << " ";

for (int friend\_id : adjList[user]) {

if (!visited[friend\_id]) {

visited[friend\_id] = true;

q.push(friend\_id);

}

}

}

✅ **Logic Summary**:

* Visit a user
* Print them
* Enqueue all their unvisited friends
* Repeat until the queue is empty

BFS uses a **queue (FIFO)** – exploring all immediate friends before going deeper.

**🌊 4. Depth-First Search (DFS)**

**🧠 Concept:**

* DFS explores **as far as possible along each branch** before backtracking.
* Like diving deep into one friend group before moving to another.

**🔁 Recursive Utility Function:**

cpp

CopyEdit

void dfsUtil(int user, vector<bool>& visited) {

visited[user] = true;

cout << user << " ";

for (int friend\_id : adjList[user]) {

if (!visited[friend\_id]) {

dfsUtil(friend\_id, visited);

}

}

}

* Uses **recursion** to dive deep.
* Each call handles one user and triggers calls for all their unvisited friends.

cpp

CopyEdit

void dfs(int start) {

vector<bool> visited(vertices, false);

dfsUtil(start, visited);

}

✅ **Logic Summary**:

* Visit a user
* Print them
* Recursively visit all unvisited friends

DFS uses **recursion or an explicit stack (LIFO)** – depth over breadth.

**🏁 5. Main Function — Tying It All Together**

cpp

CopyEdit

int users, connections;

cin >> users;

Graph facebook(users);

* Dynamic graph creation based on input user count.

cpp

CopyEdit

for (int i = 0; i < connections; ++i) {

int u, v;

cin >> u >> v;

facebook.addEdge(u, v);

}

* Dynamically adds edges based on real-time input.

cpp

CopyEdit

int start;

cin >> start;

facebook.bfs(start);

facebook.dfs(start);

* Performs both BFS and DFS from the starting user to demonstrate traversal.

**🧠 Conceptual Takeaway**

| **Concept** | **BFS** | **DFS** |
| --- | --- | --- |
| **Strategy** | Level-order | Deep-dive |
| **Uses** | Shortest path, spreading | Backtracking, connected components |
| **Data Structure** | Queue | Stack/Recursion |
| **Real-world analogy** | News Feed propagation | Exploring close friend groups |

**✨ Bonus Tips for Viva**

* BFS is like **exploring layers** outward — like Google Maps suggesting places near you.
* DFS is like **going down a rabbit hole** — like watching YouTube, one deep recommendation after another.
* Mention that your graph is **undirected** and **unweighted**.

**ASSIGNMENT 03:**

**🌐 Problem Overview – What the Code is Solving**

This C++ program:

* Models a network of cities as a graph using an **adjacency matrix**.
* Asks the user to define city names and costs between connected cities.
* Uses **Prim's Algorithm**, a greedy algorithm, to compute the **Minimum Spanning Tree (MST)** of the graph — i.e., a way to connect all cities with **minimum total cost** without forming cycles.

This is useful in real-world applications like:

* Designing cost-effective road/railway networks
* Laying down minimum wiring/cabling for networks

**🔧 Concepts Used – Algorithms / Data Structures Involved**

1. **Graph Representation**:
   * Using a **2D adjacency matrix** to represent undirected weighted graphs.
2. **Greedy Algorithm**:
   * **Prim's Algorithm** is a greedy algorithm that always picks the minimum weight edge at each step.
3. **Data Structures**:
   * Arrays (nearest[], t[][]) for MST construction and selection logic.
   * cities[] array to map city names to indices.

**🧠 Detailed Explanation of Main Logic (Prim's Algorithm)**

Let’s unpack the heart of the code: Graph::prims(int start\_v)

**❓ Goal:**

To construct the Minimum Spanning Tree starting from city start\_v.

**🔑 Key Arrays:**

* nearest[10]:  
  For each city (vertex), keeps track of the nearest connected city already in the MST.
* t[10][3]:  
  Stores the final MST edges in the format: from, to, cost.

**✅ Initialization:**

cpp

CopyEdit

fill(begin(nearest), end(nearest), -1);

for (int i = 0; i < n; i++)

if (i != start\_v)

nearest[i] = start\_v;

* All cities are initialized to consider start\_v as the closest (initial guess).
* start\_v is marked as already in MST by setting nearest[start\_v] = -1.

**🔁 Main Loop (n-1 iterations):**

cpp

CopyEdit

for (int i = 1; i < n; i++)

Each loop iteration adds one new edge to the MST.

**💡 Find the next minimum edge:**

cpp

CopyEdit

for (int k = 0; k < n; k++)

{

if (nearest[k] != -1 && cost[k][nearest[k]] < min)

{

min = cost[k][nearest[k]];

j = k;

}

}

* Among all cities not in the MST (nearest[k] != -1), pick the city j which has the smallest connecting edge cost[j][nearest[j]].

**📌 Add edge to MST:**

cpp

CopyEdit

t[r][0] = nearest[j];

t[r][1] = j;

t[r][2] = min;

* Store this edge (nearest[j] -> j) with its cost into MST.
* Mark j as now part of the MST:

cpp

CopyEdit

nearest[j] = -1;

**🔁 Update nearest[]:**

For all other cities not in the MST yet, check if this new vertex j provides a closer connection:

cpp

CopyEdit

if (nearest[k] != -1 && cost[k][j] < cost[k][nearest[k]])

{

nearest[k] = j;

}

This ensures that the MST is grown in the most efficient way at every step.

**📤 Output:**

At the end, print all selected edges and the total minimum cost.

**💡 Why This Is a Greedy Algorithm**

* **Greedy Choice**: At each step, the algorithm picks the **least-cost edge** from the current MST to a new vertex.
* **Locally Optimal Choice**: It doesn't look ahead; it builds the MST by extending it one edge at a time in the cheapest way.
* **Globally Optimal Solution**: For MSTs, greedy strategy **guarantees optimality** — proven by **Cut Property** in graph theory.

**🎓 Common Viva Q&A (Conceptual & Technical)**

| **❓ Question** | **✅ Answer** |
| --- | --- |
| What is Prim’s algorithm? | It’s a greedy algorithm to find the minimum spanning tree by growing it from a starting vertex. |
| Time complexity of Prim's algorithm? | O(n²) for adjacency matrix version (as in this code). Faster (O(E log V)) using priority queues & adjacency lists. |
| Why is Prim’s algorithm greedy? | It picks the next least-cost edge without considering future consequences. |
| Difference between Prim's and Kruskal's? | Prim's grows MST from a single vertex; Kruskal's adds smallest edges without forming cycles. |
| What is a spanning tree? | A subgraph that connects all vertices with the minimum number of edges (n-1) and no cycles. |
| Can Prim’s work on disconnected graphs? | No. MST can only exist in **connected** graphs. |
| What happens if all costs are equal? | Any spanning tree is also a MST in this case. |

**🧩 Summary of Code Flow**

plaintext

CopyEdit

main()

└── Graph g;

└── constructor (initialize matrix)

└── g.create() -> get user input & form adjacency matrix

└── g.prims(start\_city) -> build MST using greedy strategy

└── nearest[] tracks current best connections

└── t[][] stores final MST edges

**ASSIGNMENT 04:**

**🔍 AVL Tree Dictionary - Code Explanation**

**📌 Overview**

This C++ program implements a **Dictionary using an AVL Tree**. It stores words and their meanings while ensuring the binary tree remains balanced for optimal search efficiency. Each node stores:

* A word
* Its meaning
* Pointers to its left and right children
* Its height in the tree

Balancing ensures that operations like insertion and lookup are efficient (O(log n) time complexity).

**🧱 Class Structure**

**1. avl\_node Class**

Represents each node in the AVL Tree.

cpp

CopyEdit

class avl\_node {

public:

string word, meaning;

avl\_node \*left, \*right;

int height;

...

};

* Constructor initializes the word and meaning.
* Initially sets left, right as NULL and height to 1.

**2. AVL Class**

Manages the AVL Tree and implements all operations.

cpp

CopyEdit

class AVL {

avl\_node \*root;

public:

AVL(); // Constructor

void insert(string, string);

avl\_node\* create(avl\_node \*, avl\_node \*);

void display();

void inorder(avl\_node \*);

...

};

**🔧 Core Functions**

**🔁 insert(word, meaning)**

* Creates a new node and inserts it into the tree while maintaining AVL balance.

**🌲 create(node, temp)**

* Recursive function to insert a new node into the tree.
* Ensures **BST properties** and calls balance() after each insertion.

**⚖️ balance(node)**

* Calculates the balance factor:

cpp

CopyEdit

int balanceFactor = height(left) - height(right)

* Applies appropriate **rotations**:
  + LL, RR, LR, RL

**🔄 Rotations**

Used to rebalance the tree when it's unbalanced after insertion:

* **LL Rotation**: Right rotate
* **RR Rotation**: Left rotate
* **LR Rotation**: Left-Right rotate
* **RL Rotation**: Right-Left rotate

cpp

CopyEdit

node->height = max(height(node->left), height(node->right)) + 1;

**🌿 inorder(node)**

* In-order traversal prints the dictionary in **sorted order**.
* Also shows the **balance factor** of each node.

**📜 display()**

* Wrapper around inorder() to print the dictionary.

**🧪 main() Function**

Takes user input in a loop to:

1. Insert word-meaning pairs.
2. Display the entire dictionary using in-order traversal.
3. Show the balance factor of the root node.

cpp

CopyEdit

char ch;

do {

...

cout << "Do you want to enter more words? (y/n): ";

cin >> ch;

} while (ch == 'y');

**✅ Sample Output**

pgsql

CopyEdit

Enter word: apple

Enter meaning: a fruit

Do you want to enter more words? (y/n): y

Enter word: book

Enter meaning: something to read

Do you want to enter more words? (y/n): n

Dictionary (Inorder Traversal with Balance Factor):

apple : a fruit (Balance Factor: 0)

book : something to read (Balance Factor: 0)

Balance factor of root: 0

**💡 Key Points**

* AVL Trees guarantee balanced structure.
* Ensures optimal search, insertion, and deletion times.
* Useful for implementing **efficient dictionaries, spell checkers, and auto-complete engines**.

**ASSIGNMENT 05:**

**🔍 Problem Overview**

The code solves the classic **0-1 Knapsack Problem**, where:

* You are given n items, each with a **weight** and a **profit/value**.
* You have a knapsack with a **maximum weight capacity W**.
* The goal is to **maximize total profit** without exceeding the weight limit.
* You can either **include** or **exclude** each item (no partial selections allowed — hence *0-1*).

**📘 Concepts Used**

* **Dynamic Programming (DP)** – Bottom-up tabulation
* **2D Array** – To store intermediate profit results (B[i][w])
* **Backtracking** – To determine which items were selected for the optimal profit

**🧠 Detailed Explanation of Main Logic**

**1. DP Table Initialization**

cpp

CopyEdit

for (int w = 0; w <= W; w++) B[0][w] = 0;

for (int i = 1; i <= n; i++) B[i][0] = 0;

* First row and column are set to 0, representing base cases:
  + No items or
  + Zero capacity knapsack → profit = 0

**2. Filling the DP Table**

cpp

CopyEdit

for (int i = 1; i <= n; i++) {

for (int w = 1; w <= W; w++) {

if (weights[i - 1] <= w)

B[i][w] = max(profits[i - 1] + B[i - 1][w - weights[i - 1]], B[i - 1][w]);

else

B[i][w] = B[i - 1][w];

}

}

* If the item can fit in the remaining capacity w, we check:
  + **Including the item**: Add its profit + optimal value for remaining weight
  + **Excluding the item**: Take the previous best without it
* Store the max of both.

**3. Traceback to Find Selected Items**

cpp

CopyEdit

int i = n, k = W;

while (i > 0 && k > 0) {

if (B[i][k] != B[i - 1][k]) {

cout << i << " ";

k -= weights[i - 1];

}

i--;

}

* Start from B[n][W] and move up:
  + If value changed → item i was included
  + Reduce weight k accordingly

**✅ Greedy Strategy Justification**

**Greedy is *not* used** for this version.

**Why?**

* Greedy fails for 0-1 Knapsack (it works for **Fractional Knapsack**).
* In 0-1, items must be taken *whole*; profit/weight ratio alone doesn't ensure optimality.
* Hence, **DP is necessary** for global optimal solution.

**💬 Common Viva Q&A**

**1. What is the time complexity?**

* **O(n \* W)**  
  (n = number of items, W = knapsack capacity)

**2. Space complexity?**

* **O(n \* W)** (due to 2D DP table)

**3. Why do we use weights[i - 1] and profits[i - 1]?**

* Arrays are 0-indexed, but DP table is 1-indexed (we shift accordingly).

**4. Why is greedy not suitable here?**

* Because we can’t take partial items — greedy may leave behind combinations with higher total profit.

**5. What happens if all item weights > W?**

* Knapsack remains empty, profit is 0.

**6. Can this be optimized further?**

* Yes, space can be reduced to **1D DP** (O(W)), but traceback becomes complex.

**7. What is the base case in this DP approach?**

* B[0][w] = 0 for all w → 0 items means 0 profit.
* B[i][0] = 0 for all i → 0 capacity means 0 profit.

**ASSIGNMENT 06:**

**🔍 Overview**

* Stores student data: Roll No, Name, Marks.
* Uses **hashing with linear probing**.
* Implements **with-replacement** and **without-replacement** strategies.
* Stores data in a binary file student.txt.

**📦 Classes Explained**

**class record**

cpp

CopyEdit

class record {

public:

int rollno;

int marks;

char name[20];

...

};

* Holds student info.
* Default constructor initializes rollno as -1 (unused).

**class hashing**

cpp

CopyEdit

class hashing {

int rollno, pos;

...

};

* Represents entries in the **hash table**.
* rollno and pos track the student and their file location.

**class student**

cpp

CopyEdit

class student {

record rec;

hashing h[10];

int s, relt;

...

};

* Main class controlling file handling and hashing.
* s: size of each record.
* relt: total records stored so far.

**🧩 Hashing Strategies**

**🔁 create\_wr() – Insert With Replacement**

* Inserts records into a file using hash table.
* If a record hashes to a spot already occupied:
  + If the existing record is a **misplaced** entry, it's kicked out and relocated.
  + Otherwise, the new record is inserted into the next available spot.

**🔁 create\_wor() – Insert Without Replacement**

* Similar logic, **but doesn’t displace** existing records.
* It linearly probes to the next empty slot.

**🛠️ Other Functionalities**

**modify()**

* Edits existing record by roll number.
* Locates the record in the file using the hash table.

**retrieve()**

* Searches and displays a record from the file.

**display() and displayall()**

* Displays one or all records.
* Also shows current hash table state.

**📂 File Operations**

* File used: "student.txt" in binary mode.
* Every new record is written at relt \* sizeof(record) position.
* File is updated during insertion and modification.

**📋 Menu (main)**

cpp

CopyEdit

Menu:

1. Insert with Replacement

2. Insert without Replacement

3. Modify Record

4. Retrieve Record

5. Display All Records

6. Exit

**🧠 Important Concepts**

* **Hashing Function:** rollno % 10
* **Collision Handling:** Linear probing
* **With Replacement:** Prefers correct home slot
* **Without Replacement:** Keeps existing record in its position

**🧪 Test Case Suggestions**

Try these inputs:

* Insert: Roll No 21 → goes to index 1.
* Insert: Roll No 11 → also hashes to 1, tests probing.
* Insert with create\_wr() to test replacement logic.
* Modify and retrieve roll numbers like 11, 21.
* Attempt inserting more than 10 records to test overflow.

**ASSIGNMENT 7.1:**

**Problem Overview**

The **Fractional Knapsack** problem is a classic optimization problem where we are given a set of items, each with a value and weight, and a knapsack with a fixed capacity. The goal is to maximize the total value of items that can be placed in the knapsack.

Unlike the **0/1 Knapsack Problem**, where each item is either fully included or excluded, in the **Fractional Knapsack Problem**, we can take fractional parts of items. This means that if an item can't fit entirely, we can take a proportion of it, making this problem solvable via a **greedy approach**.

**Concepts Used**

1. **Greedy Algorithm**: The problem is solved by picking items in a greedy manner, choosing the item with the highest value-to-weight ratio at each step. This ensures that we get the most value per unit of weight.
2. **Sorting**: Items are sorted based on the value-to-weight ratio before selecting them.
3. **Fractional Calculation**: If an item cannot be completely accommodated in the knapsack, a fraction of the item is included based on its weight.
4. **Vectors**: Used to store the items efficiently.
5. **Operators Overloading**: The < operator is overloaded for the Item struct to facilitate sorting based on value-to-weight ratio.

**Detailed Explanation of Main Logic**

**Key Part:**

1. **Sorting the Items**: We first sort the items by their value-to-weight ratio in descending order. This ensures that items providing the highest value per unit weight are considered first.

cpp

CopyEdit

sort(items.begin(), items.end());

The custom sorting is done using the overloaded < operator, which compares items based on their value-to-weight ratio:

cpp

CopyEdit

bool operator<(Item other) {

return (double)value / weight > (double)other.value / other.weight;

}

1. **Greedy Item Selection**: We then iterate over the sorted items:
   * If the item can fit entirely in the knapsack (W >= item.weight), we subtract its weight from the knapsack capacity and add its full value to the total.
   * If the item cannot fit entirely, we take the fraction of the item that fits in the remaining capacity and add the corresponding value fraction to the total.

cpp

CopyEdit

for (Item& item : items) {

if (W >= item.weight) {

W -= item.weight;

totalValue += item.value;

} else {

totalValue += item.value \* ((double)W / item.weight);

break;

}

}

1. **Final Result**: The function returns the total value accumulated in the knapsack.

**Greedy Strategy Justification**

The **Greedy Algorithm** is optimal for the **Fractional Knapsack Problem** because:

1. **Greedy Choice Property**: By always picking the item with the highest value-to-weight ratio, we ensure that each step contributes maximally to the overall value, making this a locally optimal strategy.
2. **Optimal Substructure**: The optimal solution to the problem can be constructed from optimal solutions to subproblems, as we are only concerned with the items that remain after each step, and the greedy choice ensures that we’re making the best possible decision at every stage.

This greedy approach does not guarantee an optimal solution for the **0/1 Knapsack Problem**, but it works perfectly for the **Fractional Knapsack Problem**, as items can be broken into fractions.

**Common Viva Q&A**

**1. What is the time complexity of the solution?**

* **Time Complexity**: The time complexity is dominated by the sorting step.
  + Sorting the items takes **O(n log n)**.
  + The greedy item selection loop takes **O(n)**. Thus, the overall time complexity is **O(n log n)**.

**2. Can this problem be solved using dynamic programming?**

* No, dynamic programming is suitable for problems where you must make binary choices (like the 0/1 knapsack problem). However, in the fractional knapsack problem, the items can be taken in fractional amounts, which makes dynamic programming unnecessary and inefficient compared to the greedy approach.

**3. What if items can't be broken into fractions?**

* If items cannot be broken into fractions (as in the 0/1 Knapsack Problem), the greedy strategy would no longer work, and you would need to use **dynamic programming** or **backtracking** to find the optimal solution.

**4. What happens if we modify the problem so that items cannot be broken down?**

* This would be a **0/1 Knapsack Problem**, which can’t be solved using a greedy approach. Instead, dynamic programming or a brute-force solution is used to find the optimal solution.

**5. Can the algorithm be improved?**

* The algorithm is already optimal for this problem because sorting takes **O(n log n)**, and that's the most time-consuming operation. Further optimization isn't possible for this problem using the current approach.

**6. What are the advantages of using the greedy algorithm here?**

* The greedy algorithm is efficient (with time complexity of **O(n log n)**) and directly leads to the optimal solution for the **Fractional Knapsack Problem**. It’s simple to implement and fast compared to other methods like dynamic programming for this type of problem.

**7. What are the space complexities involved?**

* **Space Complexity**: The space complexity is **O(n)** because we store n items in a vector, and we use a constant amount of extra space for variables like totalValue and W.

**ASSIGNMENT 7.2:**

**Problem Overview**

The **Huffman Coding** algorithm is a lossless data compression technique used to assign variable-length codes to characters, based on their frequencies. Characters that occur more frequently are assigned shorter codes, while those that occur less frequently are assigned longer codes. This method is widely used in data compression formats such as ZIP files, JPEG, and MP3.

In this program, the goal is to generate **Huffman Codes** for a given set of characters and their frequencies. It uses a **priority queue (min-heap)** to build a binary tree, which is then used to assign binary codes to characters.

**Concepts Used**

1. **Priority Queue (Min-Heap)**: A priority queue is used to build the Huffman tree. The characters are sorted by frequency, and the two least frequent nodes are merged iteratively until only one node remains.
2. **Binary Tree**: The tree structure is used to represent the Huffman codes, with the leaf nodes representing characters and internal nodes representing combined frequencies.
3. **Recursion**: A recursive function is used to traverse the tree and generate the Huffman codes.
4. **Hash Map (unordered\_map)**: An unordered map is used to store the frequency of each character.
5. **Custom Comparator**: A custom comparator is used for the priority queue to ensure it functions as a min-heap based on the frequency of the characters.

**Detailed Explanation of Main Logic**

**Key Part:**

1. **Node Structure**: The Node structure represents a node in the Huffman tree. Each node contains:
   * ch: The character.
   * freq: The frequency of the character.
   * left and right: Pointers to the left and right children in the tree.

cpp

CopyEdit

struct Node {

char ch;

int freq;

Node \*left, \*right;

Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}

};

1. **Building the Huffman Tree**:
   * We first insert all characters and their frequencies into a **priority queue (min-heap)**. The priority queue is arranged such that the node with the lowest frequency is always at the top.

cpp

CopyEdit

priority\_queue<Node\*, vector<Node\*>, Compare> pq;

for (auto& pair : freqMap)

pq.push(new Node(pair.first, pair.second));

* + Then, we repeatedly extract the two nodes with the lowest frequencies, merge them into a new node, and insert this merged node back into the priority queue. This process continues until only one node remains, which becomes the root of the Huffman tree.

cpp

CopyEdit

while (pq.size() > 1) {

Node\* l = pq.top(); pq.pop();

Node\* r = pq.top(); pq.pop();

Node\* merged = new Node('$', l->freq + r->freq);

merged->left = l;

merged->right = r;

pq.push(merged);

}

1. **Printing the Huffman Codes**: Once the Huffman tree is built, the printCodes() function recursively traverses the tree and prints the binary code for each character. The left child corresponds to appending a '0', and the right child corresponds to appending a '1'.

cpp

CopyEdit

void printCodes(Node\* root, string code) {

if (!root) return;

if (root->ch != '$') cout << root->ch << ": " << code << "\n";

printCodes(root->left, code + "0");

printCodes(root->right, code + "1");

}

**Greedy Strategy Justification**

The **Greedy Algorithm** is used here to build the Huffman tree because it ensures that we always merge the two least frequent items, minimizing the overall weighted sum of the frequencies in the tree. The greedy strategy is optimal for the Huffman coding problem because:

1. **Optimal Substructure**: The optimal encoding for a set of characters depends on the optimal encodings for smaller sets of characters.
2. **Greedy Choice Property**: By always combining the least frequent characters first, we ensure that shorter codes are assigned to more frequent characters, and longer codes are assigned to less frequent characters.

This greedy approach guarantees an optimal solution for generating Huffman codes.

**Common Viva Q&A**

**1. What is the time complexity of the solution?**

* **Time Complexity**:
  + Inserting n nodes (one for each character) into the priority queue takes **O(n log n)**.
  + Each operation in the priority queue (extracting the two nodes and inserting a merged node) takes **O(log n)**.
  + As the algorithm performs **n-1** merge operations, the overall time complexity is **O(n log n)**.

**2. Can you explain why you used a min-heap?**

* The **min-heap** (priority queue) ensures that we can efficiently extract the two nodes with the smallest frequencies. By always merging the least frequent nodes, we minimize the overall cost of the encoding. The priority queue's logarithmic time complexity for insertion and extraction ensures that the algorithm runs efficiently.

**3. What if we didn’t use a min-heap?**

* Without the min-heap, finding the two smallest nodes would take linear time, which would make the algorithm inefficient. The **min-heap** guarantees that we can find the two smallest nodes in **O(log n)** time, significantly improving the performance.

**4. What if characters have the same frequency?**

* If two characters have the same frequency, the tie is broken by their order of insertion into the priority queue. Since Huffman coding is a **greedy algorithm**, it does not matter how the tie is resolved, as long as the final tree structure respects the frequencies.

**5. How can the Huffman code be represented?**

* The Huffman code can be represented as a dictionary (or hash map) mapping each character to its binary code. This would allow efficient encoding and decoding of messages.

**6. What are the space complexities involved?**

* **Space Complexity**: The space complexity is **O(n)** due to storing the nodes in the priority queue and the tree structure. We also store the frequency map, which requires **O(n)** space.

**7. What are some applications of Huffman coding?**

* **Huffman coding** is used in file compression algorithms (e.g., ZIP, GZIP), image compression (e.g., JPEG), and sound compression (e.g., MP3), among other applications.

**8. How does Huffman coding compare to other compression algorithms?**

* Huffman coding is particularly efficient for compressing data with a highly skewed distribution of characters (i.e., some characters appear very frequently while others appear very rarely). For other types of data, algorithms like **LZ77** or **LZ78** might perform better.

**ASSIGNMENT 7.3:**

**Problem Overview**

The **Job Sequencing Problem** involves scheduling a set of jobs, each with a specific deadline and profit, to maximize the total profit. The jobs are given with their deadlines and profits. The goal is to schedule the jobs such that:

* Each job is scheduled to complete before its deadline.
* A job can be scheduled only if a time slot is available.
* The aim is to maximize the total profit by scheduling the jobs with the highest profits first, without exceeding their respective deadlines.

This is a classic problem in greedy algorithms, where the best local solution (choosing the job with the highest profit) leads to the globally optimal solution.

**Concepts Used**

1. **Sorting**: The jobs are sorted in decreasing order of their profits, as higher profit jobs should be given priority.
2. **Greedy Algorithm**: The algorithm greedily assigns jobs to available time slots starting from the latest possible slot (i.e., the one closest to the job's deadline).
3. **Array (Vector)**: Two arrays are used to keep track of the results:
   * result[]: To store the scheduled jobs.
   * slot[]: To mark whether a time slot is occupied.
4. **Time Complexity**: The sorting step ensures that the algorithm runs efficiently, making it suitable for large datasets.

**Detailed Explanation of Main Logic**

**Key Part:**

1. **Job Structure**: The Job structure stores the job's ID, deadline, and profit.

cpp

CopyEdit

struct Job {

char id;

int deadline, profit;

};

1. **Sorting Jobs by Profit**: To ensure we prioritize high-profit jobs, we use the custom comparator cmp() to sort the jobs in decreasing order of profit. This step ensures that the highest-profit jobs are considered first.

cpp

CopyEdit

bool cmp(Job a, Job b) {

return a.profit > b.profit;

}

1. **Job Scheduling**: The main scheduling algorithm iterates through the sorted jobs and attempts to schedule each job starting from the latest available time slot before its deadline. If a time slot is available, the job is assigned to that slot, and the slot is marked as occupied.

cpp

CopyEdit

void jobSequencing(vector<Job>& jobs) {

sort(jobs.begin(), jobs.end(), cmp);

int n = jobs.size();

vector<char> result(n, '-'); // Stores job ids in the final schedule

vector<bool> slot(n, false); // Marks available slots

for (Job& job : jobs) {

for (int j = min(n, job.deadline) - 1; j >= 0; --j) {

if (!slot[j]) {

slot[j] = true;

result[j] = job.id;

break;

}

}

}

cout << "Job order for max profit: ";

for (char c : result)

if (c != '-') cout << c << " ";

cout << endl;

}

* + The outer loop iterates through all the jobs, sorted by profit.
  + The inner loop checks for an available time slot for each job, starting from the latest available slot.
  + If a slot is found, the job is scheduled and the slot is marked as occupied.

1. **Output**: The result is printed as a sequence of job IDs, which represents the schedule that maximizes the total profit.

**Greedy Strategy Justification**

The **Greedy Algorithm** is used in this solution because the problem exhibits the **greedy choice property** and **optimal substructure**:

* **Greedy Choice Property**: By selecting the highest-profit job first, we maximize profit at each step. The choice made does not prevent future optimal choices, ensuring that the final schedule maximizes total profit.
* **Optimal Substructure**: The problem can be broken down into smaller subproblems where we optimally select jobs for available time slots. The overall solution is built by combining the solutions to these subproblems.

Thus, the greedy approach ensures that we make the best local choice (scheduling the highest-profit job in the available slot) to reach the global optimum.

**Common Viva Q&A**

**1. What is the time complexity of this algorithm?**

* **Time Complexity**: The sorting of the jobs takes **O(n log n)**. After sorting, we iterate through each job and try to find a slot, which in the worst case takes **O(n)** operations. Hence, the overall time complexity is **O(n log n)** due to the sorting step.

**2. What happens if there are multiple jobs with the same profit?**

* The jobs with the same profit will be scheduled based on the order in which they are given in the input (after sorting by profit). The algorithm does not specify a priority for jobs with the same profit, so they are scheduled in their given order, as long as there is space available.

**3. Why does the algorithm check slots starting from the latest possible time (i.e., job.deadline - 1)?**

* The reason for starting from the latest available slot is to ensure that the jobs are scheduled as close to their deadlines as possible. This allows earlier slots to be free for other jobs, maximizing the number of jobs that can be scheduled.

**4. What if a job's deadline is larger than the number of available slots?**

* In such cases, the job will still be considered, but it will be scheduled only in a slot before its deadline. If no slots are available before its deadline, the job will not be scheduled.

**5. Can this algorithm handle jobs with zero profit or deadlines that exceed the total number of jobs?**

* The algorithm does not explicitly handle jobs with zero profit or deadlines that exceed the number of available slots, but jobs with zero profit will simply not be scheduled as they do not contribute to the final result. Jobs with deadlines larger than the total number of available slots will only be scheduled if there are available slots before the deadline.

**6. What is the space complexity of this solution?**

* **Space Complexity**: The space complexity is **O(n)**, where n is the number of jobs. This space is used to store the result[] array (which holds the scheduled jobs) and the slot[] array (which tracks whether a slot is occupied).

**7. What are some practical applications of job sequencing problems?**

* Job sequencing problems are common in task scheduling, resource allocation, job shop scheduling, and any domain where multiple tasks need to be completed within limited time slots or resources, and the objective is to maximize profit or efficiency.

**ASSIGNMENT 8:**

**Problem Overview**

The **N-Queens Problem** is a classic combinatorial problem where you are tasked with placing N queens on an N x N chessboard such that no two queens threaten each other. The queens should be placed in such a way that:

* No two queens share the same row.
* No two queens share the same column.
* No two queens share the same diagonal.

The objective is to find a configuration of queens on the board where none of them can attack each other, and if such a configuration exists, print it. If no configuration exists, indicate that no solution is possible.

**Concepts Used**

1. **Backtracking**: The algorithm uses backtracking to explore all potential configurations of queen placements. If placing a queen in a particular position leads to a conflict later on, the algorithm backtracks and tries another position.
2. **Recursive Search**: The solution uses recursion to place queens row by row, checking for conflicts as it progresses.
3. **Safety Check**: A helper function isSafe() ensures that a queen can be safely placed in a given position by checking the current column and both diagonals.
4. **Time Complexity**: The time complexity is exponential, with a worst-case complexity of **O(N!)** since there are N! ways to arrange N queens on the board.

**Detailed Explanation of Main Logic**

**Key Part:**

1. **isSafe() Function**: This function checks whether it is safe to place a queen at board[row][col] by verifying:
   * The current column (there must not be any queen already placed in the same column).
   * The upper left diagonal (i.e., cells that are diagonally left of the current cell).
   * The upper right diagonal (i.e., cells that are diagonally right of the current cell).

cpp

CopyEdit

bool isSafe(int board[N][N], int row, int col)

{

// Check column and diagonals for conflicts

for (int i = 0; i < row; i++)

if (board[i][col]) return false;

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j]) return false;

for (int i = row, j = col; i >= 0 && j < N; i--, j++)

if (board[i][j]) return false;

return true;

}

1. **solveNQueens() Function**: This function is the recursive function that attempts to place queens in each row. If it finds a valid position for a queen in a given row, it recursively tries to place queens in the subsequent rows. If a queen cannot be placed in any column of a row, it backtracks by removing the queen and trying the next possible position.

cpp

CopyEdit

bool solveNQueens(int board[N][N], int row)

{

if (row >= N)

return true; // All queens have been placed

for (int col = 0; col < N; col++)

{

if (isSafe(board, row, col)) // Try placing queen

{

board[row][col] = row + 1; // Place queen with a label (Q1, Q2, etc.)

if (solveNQueens(board, row + 1))

return true; // Recur to place next queen

board[row][col] = 0; // Backtrack if placement fails

}

}

return false; // No valid position found in this row

}

1. **printSolution() Function**: Once a valid solution is found, this function prints the board. It prints each queen's label (Q1, Q2, etc.) at its position. Empty cells are represented by dots (.).

cpp

CopyEdit

void printSolution(int board[N][N])

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

if (board[i][j] == 0)

cout << setw(3) << '.'; // Empty cell

else

cout << " Q" << board[i][j]; // Queen with label

}

cout << endl;

}

}

1. **Main Function**: The main function initializes the board and calls the solveNQueens() function to solve the problem. If a solution is found, it prints the solution using printSolution(). Otherwise, it prints a message saying that no solution exists.

cpp

CopyEdit

int main()

{

int board[N][N] = {0}; // Initialize the board

if (solveNQueens(board, 0))

printSolution(board); // Print solution if found

else

cout << "Solution does not exist\n"; // No solution found

return 0;

}

**Greedy Strategy Justification**

This solution **does not use a greedy approach**. Instead, it employs **backtracking**, which is a more suitable method for problems like the N-Queens problem where you must explore all possible configurations and backtrack when a configuration leads to a dead-end. Greedy algorithms are typically used when making a locally optimal choice leads to a globally optimal solution, which is not the case here. Instead, backtracking explores all possible solutions, ensuring that the best configuration is found.

**Common Viva Q&A**

**1. What is the time complexity of the backtracking solution?**

* **Time Complexity**: The time complexity is **O(N!)**, as there are N! possible ways to arrange N queens on the board. In the worst case, we might need to check all these possibilities.

**2. What happens if no solution exists for N = 2 or N = 3?**

* The algorithm will correctly return that no solution exists for N = 2 or N = 3 because it will exhaust all possible configurations and find that no valid solutions exist for these board sizes.

**3. Can the algorithm handle larger values of N efficiently?**

* The algorithm's efficiency decreases as N increases due to the exponential time complexity. For very large N, the algorithm may take a long time to find a solution. However, optimizations such as constraint propagation or using more sophisticated techniques like constraint satisfaction algorithms can improve performance for larger values of N.

**4. What is the space complexity of this solution?**

* **Space Complexity**: The space complexity is **O(N^2)** due to the storage of the board, which is an N x N matrix. Additionally, the recursion stack may require space proportional to O(N) in the worst case.

**5. How does the backtracking algorithm work in this case?**

* Backtracking tries placing queens row by row. For each row, it attempts to place a queen in each column and checks if it’s safe. If it finds a safe position, it moves to the next row. If it cannot place a queen in any column of a row, it backtracks by removing the queen from the previous row and tries a different position.