Greedy Algorithm

A **Greedy Algorithm** is a method for solving problems where the solution is built **step by step**. At each step, the algorithm **picks the best option available at that moment**, without worrying about the future. It hopes that by making the best local choice at each step, it will end up with the best overall solution. However, this **doesn't always guarantee the best solution** for every problem.

The two main types of Greedy Algorithms are:

1. **Fractional Greedy Algorithm**
2. **Integral Greedy Algorithm**

A **Fractional Greedy Algorithm** is a type of greedy algorithm where the solution allows **splitting items or decisions into smaller parts**. This approach is used when **it's possible to take fractions of the elements** involved in the problem, rather than requiring whole units.

Example: **Fractional Knapsack Problem**

The **Fractional Knapsack Problem** is a classic optimization problem where you are **given a set of items, each with a weight and a value**, and a knapsack with a weight capacity. The **goal is to determine the maximum value you can carry in the knapsack**, allowing you to take fractions of items (not just whole items).

Example:

- Items:
  - Item 1: weight = 10, value = 60
  - Item 2: weight = 20, value = 100
  - Item 3: weight = 30, value = 120
- Knapsack capacity = 50

1. Calculate ratios:
   - Item 1: 60/10 = 6
   - Item 2: 100/20 = 5
   - Item 3: 120/30 = 4
2. Sort by ratio: Item 1 (6), Item 2 (5), Item 3 (4).
3. Start filling the knapsack:
   - Take Item 1 fully (weight 10, value 60).
   - Take Item 2 fully (weight 20, value 100).
   - Knapsack remaining capacity = 50 - 10 - 20 = 20.
   - Take 2/3 of Item 3 (weight 20 out of 30), contributing value = (2/3) * 120 = 80.
4. Total value = 60 + 100 + 80 = 240.

Algorithm:

1. **Calculate the value-to-weight ratio** for each item: `ratio[i] = v[i] / w[i]`.
2. **Sort the items** by their value-to-weight ratio in descending order.
3. **Start filling the knapsack** by selecting items:
    - Take the whole item if the knapsack can accommodate it.
    - If the knapsack cannot accommodate the whole item, take the fraction that fits.
4. **Stop when the knapsack is full** or all items have been considered.
5. Return the total value.

```
FractionalKnapsack(weights, values, capacity)
    n = length of weights
    items = array of (value-to-weight ratio, weight, value)
    for i = 0 to n - 1
        ratio = values[i] / weights[i]
        items[i] = (ratio, weights[i], values[i])
    sort(items by ratio in descending order)
    totalValue = 0
    for each item in items
        if capacity == 0
            break
        if item.weight <= capacity
            capacity = capacity - item.weight
            totalValue = totalValue + item.value
        else
            fraction = capacity / item.weight
            totalValue = totalValue + (item.value * fraction)
            capacity = 0
    return totalValue
```

Time Complexity:

- Sorting the items takes O(nlogn)
- Iterating through the items takes O(n)
- Thus, the overall time complexity is O(nlogn)

Optimal Resource Allocation:

Optimal Resource Allocation is a problem where divisible resources (like time, bandwidth, or budget) are distributed among various tasks to maximize the total benefit or efficiency.

1. Compute the benefit-to-resource ratio for each task.
2. Sort tasks in descending order of their ratio.
3. Initialize totalBenefit = 0 and remainingResources = capacity.
4. For each task in the sorted list:
    - If the task requires less than or equal to remainingResources:
        - Add the full benefit to totalBenefit.
        - Reduce remainingResources by the task's requirement.
    - Else:
        - Add a fraction of the benefit proportional to the remaining resources.
        - Set remainingResources = 0.
5. Return totalBenefit.

```
OptimalResourceAllocation(tasks, resources)
    n = length of tasks
    items = array of (benefit-to-resource ratio, resource required,
benefit)
    for i = 0 to n - 1
        ratio = benefit[i] / resourceRequired[i]
        items[i] = (ratio, resourceRequired[i], benefit[i])
    sort(items by ratio in descending order)
    totalBenefit = 0
    for each item in items
        if resources == 0
            break
        if item.resourceRequired <= resources
            resources = resources - item.resourceRequired
            totalBenefit = totalBenefit + item.benefit
        else
            fraction = resources / item.resourceRequired
            totalBenefit = totalBenefit + (item.benefit * fraction)
            resources = 0
    return totalBenefit
```

- ### Time Complexity:
- Sorting the items takes O(n log n).
- Iterating through the items to allocate resources takes O(n).
- The space complexity is O(n).

Integral Greedy Algorithm

An **Integral Greedy Algorithm** is used when the resources or elements involved in the problem cannot be divided, meaning the solution must select whole items or resources. It is applied to problems where the decision involves selecting only complete items.

Activity Selection Problem:

- Select the maximum number of activities that can be performed within a given time frame without overlapping.

Job Sequencing Problem:

- Schedule jobs with deadlines to maximize total profit, taking whole jobs only.

Huffman Coding:

- Construct an optimal binary tree to minimize the number of bits needed to represent data, by selecting whole elements.

Prim's Algorithm:

- Find the Minimum Spanning Tree (MST) by adding edges one by one, where the edges are selected based on minimum weight.

Kruskal's Algorithm:

- Another MST algorithm where edges are sorted by weight, and edges are added to the tree as long as they don't form a cycle.

Dijkstra's Algorithm:

- Used for finding the shortest path between nodes in a graph with non-negative edge weights.

Coin Change Problem:

- Given denominations, find the minimum number of coins required to make a given amount. Only whole coins are allowed.

Set Cover Problem:

- Select the minimum number of subsets that cover all the elements in the universal set.

EXAMPLE:

JOB	DEADLINE	PROFIT
J1	2	100
J2	1	50
J3	2	200
J4	1	20

Step 1: Sort Jobs by Profit in Descending Order

JOB	DEADLINE	PROFIT
J3	2	200
J1	2	100
J2	1	50
J4	1	20

Step 2: Allocate Time Slots

Initialize Slots:
Total slots available = 2 (maximum deadline = 2).
Slots: [Empty, Empty]
Iterate Through Jobs:

J3: Deadline = 2. Place in Slot 2.
Slots: [Empty, J3]
Profit = 200
J1: Deadline = 2. Place in the next available slot (Slot 1).
Slots: [J1, J3]
Profit = 200 + 100 = 300
J2: Deadline = 1. Slot 1 is already taken. Skip this job.
J4: Deadline = 1. Slot 1 is already taken. Skip this job.

Step 3: Result

- Selected Jobs: J1, J3
- Total Profit: 300
- Final Slots: [J1, J3]

Activity Selection Problem:

The Activity Selection Problem involves selecting the maximum number of activities that can be performed within a given time frame, such that no two activities overlap. Each activity has a start time and a finish time, and the goal is to choose the maximum number of activities that can be scheduled without any conflict**.**

Steps:

1. **Sort** all activities by their finish time in non-decreasing order.
2. **Select the first activity** (the one with the earliest finish time).
3. For each subsequent activity, **select the activity** if its start time is greater than or equal to the finish time of the last selected activity.
4. **Repeat** until all activities are processed.

```
ActivitySelection(activities):

    Sort activities by finish time

    selectedActivities = []

    lastFinishTime = -1

    for activity in activities:

        if activity.start >= lastFinishTime:

            selectedActivities.append(activity)

            lastFinishTime = activity.finish

    return selectedActivities
```

Time Complexity:

- Sorting the activities takes **O(n log n)**.
- Iterating through the activities takes **O(n)**. Thus, the total time complexity is **O(n log n)**.

Space Complexity:

- The space complexity is **O(n)** to store the selected activities.

- **Example Input**

Activity	**Start Time**	**Finish Time**
A	1	4
B	3	5
C	0	6
D	5	7
E	8	9
F	5	9

Step 1: Sort Activities by Finish Time

Activity	**Start Time**	**Finish Time**
A	1	4
B	3	5
C	0	6
D	5	7
E	8	9
F	5	9

Step 2: Greedy Selection Process

1. **Initialize Current Time = 0** (No activity has been selected yet).
2. Iterate through the sorted activities.
    - If an activity's **start time ≥ Current Time**, **select the activity**.
    - Update **Current Time** to the finish time of the selected activity.

STEP	ACTIVITY	START TIME	FINISH TIME	SELECTED ?	REASON	CURRENT TIME
1	A	1	4	YES	Start ≥ Current	4
2	B	3	5	NO	Start < Current	4
3	C	0	6	NO	Start < Current	4
4	D	5	7	YES	Start ≥ Current	7
5	E	8	9	YES	Start ≥ Current	9
6	F	5	9	NO	Start < Current	9

Step 3: Output Result

- **Selected Activities**: A (1-4), D (5-7), E (8-9).
- **Maximum Number of Activities**: **3**.

Job Sequencing Problem

Problem Statement

You are given **n jobs**, each with a **deadline** and a **profit**. The goal is to schedule jobs to maximize the **total profit**, ensuring that no two jobs are scheduled at the same time and that each job is completed within its deadline.

Example Input

Job	Deadline	Profit
J1	2	100
J2	1	19
J3	2	27
J4	1	25
J5	3	15

Step 1: Sort Jobs by Profit in Descending Order

Job	Deadline	Profit
J1	2	100
J3	2	27
J4	1	25
J2	1	19
J5	3	15

Step 2: Allocate Jobs Using Greedy Approach

1. **Initialize a Time Slot Array:**
    - Create an array for time slots up to the maximum deadline, initialized to `-1` (indicating free slots).
    - Maximum deadline = 3 → Time Slot Array: `[-1, -1, -1]`.
2. **Iterate Over Jobs (in sorted order):**
    - For each job, find the **latest available time slot** (starting from its deadline).
    - Assign the job to that time slot if available.

Allocation Process

Job	Deadline	Profit	Time Slot Status	Assigned Slot
J1	2	100	[-1, J1, -1]	Slot 2
J3	2	27	[J3, J1, -1]	Slot 1
J4	1	25	[J4, J1, -1]	Slot 1 (Skipped: Already Filled)
J2	1	19	[J4, J1, -1]	Slot 1 (Skipped: Already Filled)
J5	3	15	[J4, J1, J5]	Slot 3

Step 3: Output Result

- **Scheduled Jobs**: J1 (Slot 2), J3 (Slot 1), J5 (Slot 3).
- **Maximum Profit**: **100 + 27 + 15 = 142**.

Time Complexity

1. **Sorting Jobs by Profit: O(n log n)**.
2. **Allocating Jobs to Slots: O(n × m)**, where **m** is the maximum deadline.
    - In most cases, **m ≤ n**, so the allocation process is **O(n²)**.

Overall Time Complexity: **O(n log n + n²)**.

Algorithm for Job Sequencing Problem

1. **Input**: Array of jobs, each with a profit and deadline.
2. **Sort** the jobs in descending order of profit.
3. **Initialize** a time slot array of size equal to the maximum deadline, filled with `-1` (indicating free slots).
4. **For each job** (in sorted order):
   a. Find the **latest available slot** (starting from its deadline).
   b. If a slot is found, assign the job to the slot and mark it as filled.
5. **Output** the scheduled jobs and the maximum profit.

```
JobSequencing(jobs[], n):

    Sort jobs[] in descending order of profit

    maxDeadline = maximum deadline in jobs[]

    slots[] = array of size maxDeadline, initialized to -1

    totalProfit = 0



    for each job in jobs[]:

        for j = job.deadline to 1:

            if slots[j] == -1:

                slots[j] = job.id

                totalProfit += job.profit

                break



    return totalProfit, slots
```

Huffman Coding Problem

Huffman Coding is a **lossless data compression** algorithm used to minimize the number of bits needed to represent data by assigning shorter codes to frequently occurring symbols and longer codes to less frequent symbols.

Example

Let’s take an example of 5 symbols with their frequencies:

Symbol	Frequency
A	5
B	9
C	12
D	13
E	16

Step-by-Step Procedure

Step 1: Build Min-Heap

Initially, the heap will look like:

Symbol	Frequency
A	5
B	9
C	12
D	13
E	16

Step 2: Construct Huffman Tree

1. Extract the two nodes with the smallest frequencies: A (5) and B (9).
    - Create a new node with frequency 14 (5 + 9).
    - Insert this new node back into the heap.

Symbol	Frequency
C	12
D	13
E	16
NewNode (14)	14

2. Extract the two nodes with the smallest frequencies: C (12) and D (13).
    - Create a new node with frequency 25 (12 + 13).
    - Insert this new node back into the heap.

Symbol	Frequency
E	16
NewNode (14)	14
NewNode (25)	25

3. Extract the two nodes with the smallest frequencies: E (16) and NewNode (14).
    - Create a new node with frequency 30 (16 + 14).
    - Insert this new node back into the heap.

Symbol	Frequency
NewNode (25)	25
NewNode (30)	30

4. Extract the two nodes with the smallest frequencies: NewNode (25) and NewNode (30).
    - Create a new node with frequency 55 (25 + 30).
    - This node becomes the root of the tree.

Symbol	Frequency
RootNode (55)	55

Step 3: Assign Codes

- Start from the root and assign `0` for left edges and `1` for right edges.
- The tree will look like this:

```
   [55]
      /     \
    [25]     [30]
   /   \     /    \
[12]  [13] [14]  [16]
 C      D    A+B    E
              / \
            [5] [9]
            A    B
```

```
       [55]
         /    \
       [25]     [30]
      /  \     /   \
   [12]  [13] [14]  [16]
    C      D    A+B    E
                / \
              [5] [9]
               A    B
```

We assign codes by traversing the tree: **Left = 0** and **Right = 1**.

1. **A**:
   - Path: Right → Left → Left
   - Code: 100
2. **B**:
   - Path: Right → Left → Right
   - Code: 101
3. **C**:
   - Path: Left → Left
   - Code: 00
4. **D**:
   - Path: Left → Right
   - Code: 01
5. **E**:
   - Path: Right → Right
   - Code: 11

Final Huffman Codes

Symbol	Code
A	100
B	101
C	00
D	01
E	11

Steps to Solve Huffman Coding

1. **Input**: A set of symbols and their corresponding frequencies.
2. **Build a Min-Heap**: Create a min-heap (priority queue) where each node contains a symbol and its frequency. The heap is ordered by frequency in ascending order.
3. **Construct Huffman Tree**:
   - Extract two nodes with the lowest frequencies from the heap.
   - Create a new internal node with the sum of their frequencies as the new frequency. This node becomes their parent.
   - Insert the new node back into the heap.
   - Repeat this process until only one node remains, which becomes the root of the Huffman tree.
4. **Generate Huffman Codes**:
   - Traverse the tree from the root to the leaves.
   - Assign a `0` for left edges and a `1` for right edges. The code for each symbol is the path from the root to the corresponding leaf node.

Pseudocode for Huffman Coding

```
HuffmanCoding(symbols[], frequencies[], n):
    Create a min-heap with symbols and their frequencies
    while heap size > 1:
        Extract two nodes with the smallest frequencies
        Create a new internal node with the sum of the two frequencies
        Insert the new node back into the heap
    Generate Huffman codes from the tree by traversing from root to leaves
    return the Huffman codes
```

Time Complexity

1. **Building Min-Heap**: **O(n log n)**, where `n` is the number of symbols.
2. **Constructing the Tree**: **O(n log n)**, because we perform n-1 extractions from the heap.
3. **Generating the Codes**: **O(n)**, since we visit each node of the tree.

Overall Time Complexity: **O(n log n)**.

Prim's Algorithm (for Minimum Spanning Tree)

Definition:
Prim's Algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a connected, weighted graph. The MST connects all vertices with the minimum total edge weight, without forming a cycle.

Algorithm:

1. Start with any vertex as the initial node.
2. Mark it as visited.
3. Add the edge with the smallest weight that connects a visited node to an unvisited node.
4. Mark the newly connected node as visited.
5. Repeat steps 3–4 until all vertices are included in the tree.

Pseudo Code:

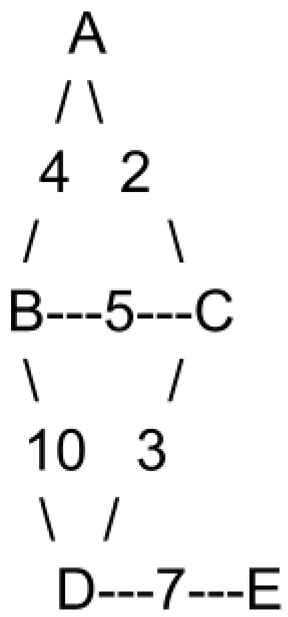
```
Prim(graph, V):
    Initialize visited[] as false for all vertices
    Initialize MST as empty
    Start from any vertex, mark it as visited
    While MST does not contain V - 1 edges:
        Find the minimum weight edge (u, v) such that:
            u is visited, and v is not visited
        Add (u, v) to MST
        Mark v as visited
    Return MST
```

Example Problem:

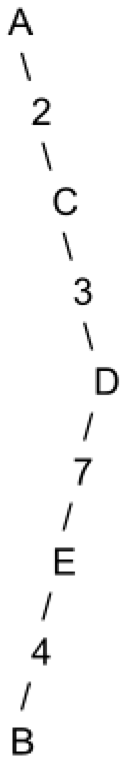
Graph Representation (Edge Weights):

From	**To**	**Weight**
A	B	4
A	C	2
B	C	5
B	D	10
C	D	3
D	E	7

Original Tree



Minimum spanning Tree



Step-by-Step Execution:

1. **Start at A**:
   - Mark A as visited.
   - Choose the smallest edge: **A → C (Weight 2)**.
2. **Now, C is visited**:
   - Edges to consider: **A → B (4)**, **C → D (3)**.
   - Pick **C → D (Weight 3)**.
3. **Now, D is visited**:
   - Edges to consider: **A → B (4)**, **D → E (7)**.
   - Pick **A → B (Weight 4)**.
4. **Now, B is visited**:
   - Edge to consider: **D → E (7)**.
   - Pick **D → E (Weight 7)**.

Result:

The **Minimum Spanning Tree** consists of the edges:

- **A → C (2)**, **C → D (3)**, **A → B (4)**, **D → E (7)**.

Total Weight = 2 + 3 + 4 + 7 = 16.

Time Complexity:

- Using adjacency matrix: **$O(V^2)$**
- Using adjacency list with a priority queue: **$O(E \log V)$**

Kruskal’s Algorithm

Kruskal’s Algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a connected, weighted graph. It selects edges in increasing order of weight and ensures no cycles are formed.

Algorithm

1. **Sort** all the edges in non-decreasing order of their weights.
2. Initialize an empty MST and mark each vertex as its own set (for disjoint-set operations).
3. For each edge in the sorted list:
    - If adding the edge does not form a cycle (vertices are in different sets), add the edge to the MST.
    - Use **Union-Find** to merge sets of the connected vertices.
4. Stop when the MST contains exactly V−1V - 1 edges, where VV is the number of vertices.

Pseudo Code

```
Kruskal(graph, V):
    Sort all edges by increasing weight
    Initialize MST as empty
    Initialize disjoint sets for all vertices
    For each edge (u, v) in sorted edges:
        If u and v are in different sets:
            Add edge (u, v) to MST
            Union sets of u and v
        If MST contains V - 1 edges:
            Break
    Return MST
```

Example Problem

Graph Representation (Edge Weights):

Edge	Weight
A-B	4
A-C	2
B-C	5
B-D	10
C-D	3
D-E	7

ORIGINAL TREE

```
     A
    / \
   4   2
  /     \
 B---5---C
  \     /
   10  3
    \ /
     D---7---E
```

Minimum Spanning Tree

```
   A
  / \
 2   4
/     \
C      B
\
 3
  \
   D
    \
     7
      \
       E
```

Step-by-Step Execution

1. **Sort edges by weight:**
   - Sorted order: **A → C (2)**, **C → D (3)**, **A → B (4)**, **B → C (5)**, **D → E (7)**, **B → D (10)**.
2. **Initialize MST as empty**.
3. Process edges in sorted order:
   - **A → C (2)**: No cycle → Add to MST.
   - **C → D (3)**: No cycle → Add to MST.
   - **A → B (4)**: No cycle → Add to MST.
   - **D → E (7)**: No cycle → Add to MST.
   - **B → C (5)**: Forms a cycle → Skip.
   - **B → D (10)**: Forms a cycle → Skip.
4. **Stop when MST has V−1=4V - 1 = 4 edges**.

Result

The **Minimum Spanning Tree** consists of the edges:

- **A → C (2)**, **C → D (3)**, **A → B (4)**, **D → E (7)**.

Total Weight = 2 + 3 + 4 + 7 = 16.

Time Complexity

- Sorting edges: **O(E log E)**
- Union-Find operations: **O(E log V)** (with path compression and union by rank).
- **Overall Complexity: O(E log V)**, where EE is the number of edges, and VV is the number of vertices.

Dijkstra's Algorithm

Dijkstra's Algorithm is a **greedy algorithm** used for finding the **shortest path** from a starting vertex (source) to all other vertices in a **weighted graph** with **non-negative edge weights**.

Steps of Dijkstra's Algorithm

1. **Initialization**:
    - Set the distance to the source vertex as 0 (`dist[source] = 0`).
    - Set the distance to all other vertices as infinity (`dist[v] = ∞` for all vertices `v ≠ source`).
    - Mark all vertices as unvisited.
    - Create a priority queue (or min-heap) to store the vertices based on their distances, starting with the source vertex.
2. **Visit the Nearest Vertex**:
    - Select the unvisited vertex with the smallest tentative distance (initially the source vertex).
    - For the current vertex `u`, examine its unvisited neighbors.
    - For each neighbor `v`, calculate the tentative distance: `dist[v] = min(dist[v], dist[u] + weight(u, v))`, where `weight(u, v)` is the edge weight between `u` and `v`.
3. **Mark the Current Vertex as Visited**:
    - Once all neighbors of the current vertex are processed, mark it as visited.
    - A visited vertex will not be checked again.
4. **Repeat**:
    - Repeat steps 2 and 3 until all vertices have been visited or the smallest tentative distance in the priority queue is infinity (which means the remaining vertices are unreachable).
5. **Termination**:
    - Once all vertices have been visited, the algorithm terminates. The `dist[]` array contains the shortest distances from the source vertex to all other vertices.

Pseudo Code

```
Dijkstra(graph, source):
    Initialize distance[source] = 0, distance[v] = ∞ for all v ≠ source
    Create a priority queue (min-heap) Q with all vertices, prioritized by distance
    While Q is not empty:
        u = vertex with the smallest distance in Q
        Remove u from Q
        For each neighbor v of u:
            if v is still in Q:
                alt = distance[u] + weight(u, v)
                if alt < distance[v]:
                    distance[v] = alt
                    update the priority of v in Q
    Return distance[] array
```

Example

Consider the following graph:

```
A --(1)-- B --(3)-- D
 |        |        |
(4)      (2)      (1)
 |        |        |
 C --(5)-- E --(6)-- F
```

- **Vertices**: A, B, C, D, E, F
- **Edges**: (A → B: 1), (A → C: 4), (B → D: 3), (B → E: 2), (C → E: 5), (D → F: 1), (E → F: 6)

Execution of Dijkstra's Algorithm (Source = A)

Initialization:

dist[A] = 0, dist[B] = ∞, dist[C] = ∞, dist[D] = ∞, dist[E] = ∞, dist[F] = ∞

1.
2. **Start at A** (smallest distance is 0):
    - A → B (distance = 1)
    - A → C (distance = 4)

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = ∞, dist[E] = ∞, dist[F] = ∞

3.
4. **Next, visit B** (smallest distance is 1):
    - B → D (distance = 1 + 3 = 4)
    - B → E (distance = 1 + 2 = 3)

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = 4, dist[E] = 3, dist[F] = ∞

5.
6. **Next, visit E** (smallest distance is 3):
    - E → F (distance = 3 + 6 = 9)
    - E → C (distance = 3 + 5 = 8, but C already has distance 4, so ignore it)

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = 4, dist[E] = 3, dist[F] = 9

7.
8. **Next, visit C** (smallest distance is 4):
    - No updates to distances.

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = 4, dist[E] = 3, dist[F] = 9

9.
10. **Next, visit D** (smallest distance is 4):
    - D → F (distance = 4 + 1 = 5)

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = 4, dist[E] = 3, dist[F] = 5

11.
12. **Finally, visit F** (smallest distance is 5).

Final Shortest Distances from A:

dist[A] = 0, dist[B] = 1, dist[C] = 4, dist[D] = 4, dist[E] = 3, dist[F] = 5

Time Complexity

- **Priority Queue Operations**: Each vertex is processed once, and for each vertex, we perform a decrease-key operation (or priority queue update) for its neighbors.
- **Total Complexity**:
  - Using an **adjacency list** and a **min-heap** (priority queue), the time complexity is:
    O((V + E) log V)
    where VV is the number of vertices, and EE is the number of edges.

Coin Change Problem

The **Coin Change Problem** is a classic dynamic programming problem where, given a set of coin denominations and a target amount, the task is to determine the minimum number of coins needed to make that amount.

Problem Definition

Given:

- A set of coin denominations.
- A target value (amount).

The goal is to find the **minimum number of coins** required to make the target value, using any number of coins from the given denominations.

Example

Consider the following denominations and target value:

- Denominations: {1, 2, 5}
- Target amount: 11

Output:

- Minimum coins needed: **3**
    - Explanation: The optimal way to form the amount 11 is by using the coins: 5 + 5 + 1.

Dynamic Programming Approach

1. **Initialization**:

    - Create an array `dp[]` where `dp[i]` will store the minimum number of coins required to make amount `i`.
    - Set `dp[0] = 0` (since no coins are needed to make amount 0).
    - Set all other `dp[i]` values initially to infinity (`∞`), as we don't know the minimum coins for these amounts yet.

2. **State Transition**:
    - For each coin in the denominations, for every amount `i` from the coin value to the target amount:
        - Update `dp[i] = min(dp[i], dp[i - coin] + 1)`.
3. **Result**:
    - The final answer will be stored in `dp[target]`.

Pseudo Code

```
CoinChange(coins[], target):
    Initialize dp[target + 1]
    dp[0] = 0
    For i = 1 to target:
        dp[i] = ∞
    For each coin in coins:
        For i = coin to target:
            dp[i] = min(dp[i], dp[i - coin] + 1)
    If dp[target] == ∞:
        Return -1  // No solution
    Else:
        Return dp[target]
```

Example Walkthrough (Denominations: {1, 2, 5}, Target: 11)

Initialization:

dp[0] = 0, dp[1] = ∞, dp[2] = ∞, dp[3] = ∞, dp[4] = ∞, dp[5] = ∞, ..., dp[11] = ∞

Using coin 1:

Update `dp[i]` for all `i` from 1 to 11:

dp[1] = min(dp[1], dp[0] + 1) = 1
dp[2] = min(dp[2], dp[1] + 1) = 2
dp[3] = min(dp[3], dp[2] + 1) = 3
dp[4] = min(dp[4], dp[3] + 1) = 4
dp[5] = min(dp[5], dp[4] + 1) = 5
dp[6] = min(dp[6], dp[5] + 1) = 6
dp[7] = min(dp[7], dp[6] + 1) = 7
dp[8] = min(dp[8], dp[7] + 1) = 8
dp[9] = min(dp[9], dp[8] + 1) = 9
dp[10] = min(dp[10], dp[9] + 1) = 10
dp[11] = min(dp[11], dp[10] + 1) = 11

Using coin 2:

Update `dp[i]` for all `i` from 2 to 11:

dp[2] = min(dp[2], dp[0] + 1) = 1
dp[3] = min(dp[3], dp[1] + 1) = 2
dp[4] = min(dp[4], dp[2] + 1) = 2
dp[5] = min(dp[5], dp[3] + 1) = 3
dp[6] = min(dp[6], dp[4] + 1) = 3
dp[7] = min(dp[7], dp[5] + 1) = 4
dp[8] = min(dp[8], dp[6] + 1) = 4
dp[9] = min(dp[9], dp[7] + 1) = 5
dp[10] = min(dp[10], dp[8] + 1) = 5
dp[11] = min(dp[11], dp[9] + 1) = 6

Using coin 5:

Update `dp[i]` for all `i` from 5 to 11:

dp[5] = min(dp[5], dp[0] + 1) = 1
dp[6] = min(dp[6], dp[1] + 1) = 2
dp[7] = min(dp[7], dp[2] + 1) = 3
dp[8] = min(dp[8], dp[3] + 1) = 4
dp[9] = min(dp[9], dp[4] + 1) = 5
dp[10] = min(dp[10], dp[5] + 1) = 2
dp[11] = min(dp[11], dp[6] + 1) = 3

1. **Result**: The minimum number of coins needed to make 11 is `dp[11] = 3`.

Time Complexity

- Sorting coins: **O(C)** where C is the number of coin denominations.
- Filling the dp array: **O(target × C)** where `target` is the amount and `C` is the number of coin denominations.

Thus, the **time complexity** is **O(target × C)**.

Space Complexity

- The space complexity is **O(target)** due to the `dp[]` array storing the minimum coins required for every amount from 0 to `target`.

Set Cover Problem

The Set Cover Problem is a classical problem in computer science and optimization where, given a universal set and a collection of subsets, the goal is to find the smallest number of subsets that together cover the entire universal set.

Problem Definition

- Input:
  - A universal set U consisting of elements u1, u2, ..., un.
  - A collection of subsets S1, S2, ..., Sm such that each subset is a subset of U.
- Output:
  - A collection of subsets such that the union of these subsets equals U, and the number of subsets is minimized.

Example

Given:

- Universal set U = {1, 2, 3, 4, 5}
- Subsets:
  - S1 = {1, 2, 3}
  - S2 = {2, 4}
  - S3 = {1, 4, 5}
  - S4 = {3, 5}

We need to find the smallest collection of subsets whose union is the entire set U.

Solution:

- The optimal solution is to pick S1 and S3, as their union is U:
  - S1 ∪ S3 = {1, 2, 3} ∪ {1, 4, 5} = {1, 2, 3, 4, 5}

Thus, the smallest number of subsets required to cover U is 2.

Greedy Approach to Solve Set Cover Problem

1. Initialization:
    - Start with an empty set C (the set of selected subsets).
    - Let U' be the set of uncovered elements initially equal to U.
2. Greedy Selection:
    - At each step, choose the subset Si that covers the maximum number of uncovered elements in U'.
    - Add Si to C and update U' by removing the elements covered by Si.
    - Repeat the process until U' becomes empty.
3. Output:
    - The collection C will contain the selected subsets, and the size of C will be the number of subsets needed to cover U.

Pseudo Code

```
SetCover(U, S):
    C = empty set  // collection of subsets
    U' = U         // set of uncovered elements
    While U' is not empty:
        max_cover = 0
        best_set = null
        For each set in S:
            coverage = number of uncovered elements in set
            If coverage > max_cover:
                max_cover = coverage
                best_set = set
        Add best_set to C
        Remove elements covered by best_set from U'
    Return C
```

Given:

- Universal set U = {1, 2, 3, 4, 5}
- Subsets:
    - S1 = {1, 2, 3}
    - S2 = {2, 4}
    - S3 = {1, 4, 5}
    - S4 = {3, 5}

1. Initialization:
    - U' = {1, 2, 3, 4, 5}
    - C = empty set
2. First Iteration:
    - Coverage for each set:
        - S1: covers {1, 2, 3}, so coverage = 3.
        - S2: covers {2, 4}, so coverage = 2.
        - S3: covers {1, 4, 5}, so coverage = 3.
        - S4: covers {3, 5}, so coverage = 2.
    - Select S1 or S3 (both cover 3 elements). Let's choose S1.
    - Add S1 to C, update U' = {4, 5}.
3. Second Iteration:
    - Coverage for each remaining set:
        - S2: covers {4}, so coverage = 1.
        - S3: covers {4, 5}, so coverage = 2.
        - S4: covers {5}, so coverage = 1.
    - Select S3 (covers the most elements).
    - Add S3 to C, update U' = empty set.
4. Result:
    - C = {S1, S3}, and U' = empty set, so the set is covered.

Time Complexity

- Greedy Selection: In each step, we go through all subsets to calculate coverage. For each subset, we check the uncovered elements.
- Time Complexity: O(m * n), where m is the number of subsets and n is the size of the universal set.

Space Complexity

- We use space for storing the sets U' and the collection C, so the space complexity is O(n + m), where n is the size of the universal set and m is the number of subsets.