Module 4 Greedy Method and Backtracking

4.1. Greedy Method:

The greedy method is an algorithmic approach used to solve optimization problems. The fundamental principle behind the greedy method is to make a sequence of choices, each of which looks best at the moment, hoping that these local optimal choices will lead to a globally optimal solution. Here is a general outline of the greedy method:

1. **Define the Problem:** Specify the optimization problem you need to solve. This involves identifying the objective function you want to maximize or minimize and any constraints that need to be satisfied.
2. **Determine a Greedy Choice Property:** Identify a local optimal choice that can be made at each step. This involves selecting the best possible option available now without considering future consequences.
3. **Prove that a Greedy Choice Leads to an Optimal Solution:** Show that making the greedy choice at each step results in an optimal solution. This typically involves proving that the problem has the optimal substructure and that the greedy choice property is safe to use.
4. **Develop a Greedy Algorithm:** Construct an algorithm that iteratively makes greedy choices until a solution is found. This usually involves initializing an empty solution set and adding elements based on the greedy choice property until the problem constraints are satisfied.
5. **Analyze the Algorithm:** Evaluate the algorithm's time and space complexity. Ensure that it is efficient and suitable for the given problem size.

Steps to Apply the Greedy Method

1. **Understand the Problem:**
   - Define the objectives of the problem (e.g., maximize profit, minimize cost).
   - Identify all the constraints that must be satisfied.
2. **Determine the Greedy Choice Property:**

- Identify the local optimal choice that appears best at the current step.
- Ensure that this option leads to a global optimal solution.

3. **Prove the Greedy Choice is Safe:**
   - Demonstrate that a greedy choice at each step maintains the feasibility and optimality of the solution.
   - Show that the problem has an optimal substructure, meaning that an optimal solution to the problem contains optimal solutions to subproblems.
4. **Design the Greedy Algorithm:**
   - Initialize the solution set (it could be empty or a starting feasible solution).
   - Iteratively make the greedy choice and add it to the solution set.
   - Ensure that the solution set remains feasible after each addition.
5. **Analyze the Algorithm:**
   - Evaluate the time complexity to ensure it is efficient.
   - Assess the space complexity.
6. **Implement the Algorithm:**
   - Code the algorithm following the designed steps.
   - Ensure correctness by handling edge cases and constraints.
7. **Verify the Solution:**
   - Test the algorithm with various inputs, including edge cases.
   - Compare the solution with known optimal solutions if available.

Example Algorithm:

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

4.2 Knapsack Problem:

The Knapsack problem is a classic optimization problem with several variations, including the **0/1 Knapsack problem** and the **Fractional Knapsack problem**. Let us outline the greedy method specifically for the Fractional Knapsack problem and then discuss why it does not work for the 0/1 Knapsack problem.

Fractional Knapsack Problem

Problem Statement: You are given a set of items, each with a weight and a value. You need to maximize the total value of your knapsack without exceeding its weight capacity. You can take fractions of items.

Greedy Method for Fractional Knapsack Problem

1. **Understand the Problem:**

   Objective: Maximize the knapsack total value.

   Constraints: The total weight of selected items must be at most the knapsack's capacity.

2. **Determine the Greedy Choice Property:**

Greedy Choice: Select items respecting on the highest value to weight ratio first.

3. **Prove the Greedy Choice is Safe:**

In the fractional knapsack problem, the greedy choice of taking items based on the highest value-to-weight ratio can be proven to lead to an optimal solution. This is because you can take fractions of items.

4. **Design the Greedy Algorithm:**

Initialization: Start with an empty knapsack and a total value of 0.

Iteration: Sort items in descending order by using their value-to-weight ratio.

For each item, if the item can fit entirely in the knapsack, add it; otherwise, add the fraction that fits.

5. **Analyze the Algorithm:**

Time Complexity: Sorting the items takes O(nlogn), and iterating through them takes O(n), so the overall complexity is O(nlogn).
Space Complexity: O(n) for storing the items and their value-to-weight ratios.

6. **Verify the Solution:**

Test with different sets of items and capacities.

Compare results with known optimal solutions to ensure correctness.

4.3 Job Sequencing with Deadlines

Problem Statement: You are given n jobs, each with a deadline and a profit. Each job takes one unit of time to complete, and it can only be performed if it is completed by its deadline. The goal is to schedule the jobs to maximize the total profit.

```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Greedy Method for Job Sequencing with Deadlines

1. **Understand the Problem:**

   Objective: Maximize the total profit.

   Constraints: Each job takes one unit of time and must be completed by its deadline.

2. **Determine the Greedy Choice Property:**

   Greedy Choice: Select jobs in descending order of profit and schedule them at the latest possible time slot before their deadline.

3. **Prove the Greedy Choice is Safe:**

   The greedy choice of selecting jobs based on the highest profit first and scheduling them at the latest possible time slot ensures that we maximize profit while still meeting deadlines.

4. **Design the Greedy Algorithm:**

   Sort the jobs in descending order of profit.

   Use a time slot array to keep track of free time slots.

   Iterate through the jobs, and for each job, find the latest available time slot before its deadline and schedule the job there if possible.

5. **Analyze the Algorithm:**

   Time Complexity: Sorting the jobs takes O(n log n), and scheduling the jobs takes O(n d) where d is the maximum deadline. The overall complexity is O(n log n + n d).

   Space Complexity: O(n+d) stores the jobs and time slots.

Example 4.2 Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

4.4 Huffman Codes

Huffman coding is an optimal prefix code commonly used for lossless data compression. Building Huffman codes involves creating a binary tree of nodes and using the properties of binary trees to generate the codes. Here's a step-by-step explanation and implementation in Java.

Huffman Coding Algorithm

1. **Calculate the Frequency of Each Character:**

   Determine how often each character appears in the input data.

2. **Build a Priority Queue:**

Create a priority queue (min-heap) where each node is a character and its frequency.

3. **Build the Huffman Tree:**

   While there is more than one node in the queue:

   - Remove the two nodes of the highest priority (lowest frequency).
   - Create a new internal node with these two nodes as children and with a frequency equal to the sum of the two nodes' frequencies.
   - Add the new node to the priority queue.

   The remaining node is the root of the Huffman tree.

4. **Generate Huffman Codes:**

   Traverse the tree from the root to the leaves, assigning codes (0 for left, 1 for right) to each character.

5. **Encode the Data:**

   Replace each character in the input data with its corresponding Huffman code.

6. **Decode the Data:**

   Use the Huffman tree to decode the encoded data to the original characters.

4.5 Single Source Shortest Paths Algorithm

The Single Source Shortest Path (SSSP) algorithm is used to find the shortest path from a single source node to all other nodes in a weighted graph. The two most common algorithms for solving this problem are Dijkstra's Algorithm and the Bellman-Ford Algorithm. Here, we will provide a detailed explanation and implementation of both algorithms in Java.

Dijkstra's Algorithm is used for graphs with non-negative weights. It uses a priority queue to select the node with the smallest tentative distance greedily.

Dijkstra's Algorithm Steps

1. **Initialize:**

   Set the distance to the source node to 0 and all other nodes to infinity.

   Use a priority queue to keep track of the minimum distance node.

2. **Process Nodes:**

   Extract the node with the smallest distance from the priority queue.

   Update the distances to its neighboring nodes.

   Repeat until all nodes have been processed.

Bellman-Ford Algorithm Steps

The Bellman-Ford algorithm is a single-source shortest path algorithm that can handle graphs with negative weights and detect negative weight cycles. It works by repeatedly relaxing all edges in the graph and checking for negative weight cycles.

Bellman-Ford Algorithm Steps

1. **Initialization:**

   Set the distance to the source node to 0 and all other nodes to infinity.

2. **Relax Edges:**

   Repeat this process V−1 times (where V is the number of vertices):

   - For each edge (u,v) if the distance to the destination v can be shortened by taking the edge (u,v).
   - update the distance to v.

3. **Check for Negative Cycles:**

   Repeat the edge relaxation one more time. If any distance can be updated, a negative weight cycle exists.

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                       at least one incoming edge do
10                  for each ⟨i, u⟩ in the graph do
11                      if dist[u] > dist[i] + cost[i, u] then
12                          dist[u] := dist[i] + cost[i, u];
13  }
```

4.6 Optimal Merge Patterns

Optimal merge patterns are used to find the most efficient way to combine multiple lists that are sorted into one sorted list with the minimum number of comparisons. This problem is commonly found in applications like external sorting and data compression.

Optimal Merge Pattern Problem

Given a set of sorted lists, the goal is to merge them so that the total number of comparisons is minimized. This problem can be solved using a greedy approach with a priority queue (min-heap).

Optimal Merge Pattern Algorithm for Two-way Merge Tree:

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

1   Algorithm Tree(n)
2   // list is a global list of n single node
3   // binary trees as described above.
4   {
5       for i := 1 to n − 1 do
6       {
7           pt := new treenode; // Get a new tree node.
8           (pt → lchild) := Least(list); // Merge two trees with
9           (pt → rchild) := Least(list); // smallest lengths.
10          (pt → weight) := ((pt → lchild) → weight)
11                  +((pt → rchild) → weight);
12          Insert(list, pt);
13      }
14      return Least(list); // Tree left in list is the merge tree.
15  }
```

Steps to Solve Optimal Merge Pattern

1. **Initialize a Min-Heap:**

   Insert the sizes of all sorted lists into a min-heap.

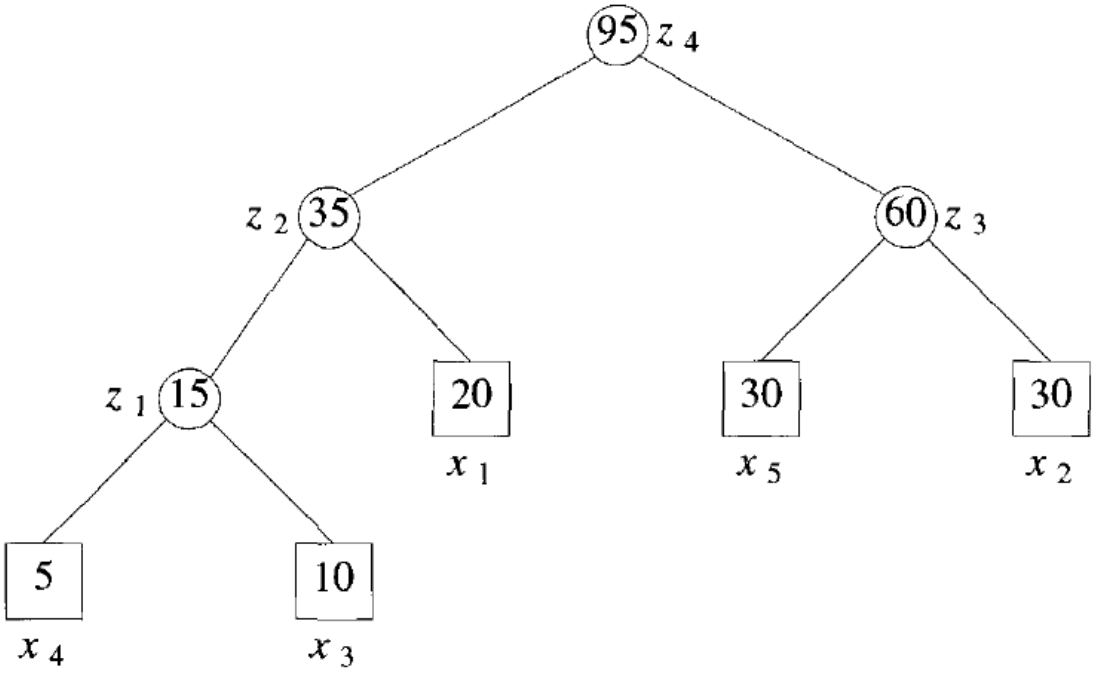
2. **Merge Lists:**

   While there are more than one list in the heap:

   - Extract the two smallest lists from the heap.
   - Merge these two lists.
   - Insert the size of the merged list back into the heap.
   - Keep track of the total comparisons made.

3. **Result:** The total number of comparisons answers the optimal merge pattern problem.

Binary Merge Tree Example:



Explanation

1. **Min-Heap Initialization:**

   The sizes of the sorted lists are added to a min-heap, which allows efficient extraction of the smallest elements.

2. **Merge Process:**

   The two smallest lists are repeatedly extracted and merged, and their combined size is added back to the heap.

   The number of comparisons required to merge two lists equals the sum of their sizes.

3. **Total Comparisons:**

   The total number of comparisons is accumulated during the merge process.

The greedy approach ensures that the smallest lists are always merged first, minimizing the total number of comparisons. This method is efficient and optimal for solving the merge pattern problem.