COSC2658 - Data Structures and Algorithms/COSC2469 - Algorithms and Analysis/COSC2203 - Algorithms and Analysis

TEST 2 (SAMPLE)

Test Overview:

- **Course Codes and Names**: COSC2658 - Data Structures and Algorithms, COSC2469 - Algorithms and Analysis, COSC2203 - Algorithms and Analysis.
- **Length**: 3 hours + 10 minutes for submission.
- **Type**: Individual.
- **Feedback Mode**: Written feedback.
- **Learning Objectives Assessed**:
  - Key algorithmic design paradigms.
  - Key data structures.
  - General algorithmic problem types.
  - Algorithmic tradeoffs.
  - Implementation and application of algorithms and data structures.

Preparation Advice:

1. **Understand Key Concepts**: Review key algorithmic design paradigms, data structures, and problem-solving techniques.
2. **Practice Coding**: Implement and test algorithms in Java, as the test requires coding in this language.
3. **Time Management**: Practice solving problems within a set timeframe to get accustomed to the test duration.
4. **Review Course Materials**: Revisit lectures, notes, and assignments related to the course.
5. **Clarify Doubts**: Use course resources like Canvas and Microsoft Teams to ask questions or clarify doubts.

Problem Requirements:

1. **Problem 1 - Secret Search**:
   - Implement a class with specific methods to calculate agents' meeting point.
   - Must follow Java coding conventions and name classes using a specified format.
2. **Problem 2 - Doraemon Cake**:
   - Implement a class to calculate the largest weight of topics that can be printed on a cake's surface.
   - Requires understanding of arrays and optimization problems.
3. **Problem 3 - Easy Learning**:
   - Create a class for finding the best learning sequence with the minimum total switching cost.
   - Involves working with matrices and sequence optimization.
4. **Problem 4**:
   - Algorithmic challenge related to coin collection.
   - Analyze and propose an efficient algorithm for the given problem.

Each problem requires creating a Java class with specific functionalities and methods, adhering to the instructions provided. The test seems to be focused on evaluating students' understanding of algorithms, data structures, and their application in solving real-world problems. Students should practice coding these structures and algorithms in Java, understand their time complexities, and be familiar with problem-solving strategies.

Problem 1

Certainly! To solve the problem step by step, we will need to:

1. **Create the `SecretSearch` Class**: A Java class to encapsulate the problem.
2. **Implement the Constructor**: To initialize the variables with the agents' starting positions and velocities.
3. **Implement the `minTimeA()` Method**: To find the minimum time for agent A to reach line L (where Y=0).
4. **Implement the `timeFromA(double XZ)` Method**: To compute the time for agent A to reach a point Z on line L.

5. **Implement the `pointC()` Method**: To find the X coordinate of point C where both agents arrive at the same time.

Here's how to implement each step:

Step 1: Create the `SecretSearch` Class

```
public class FirstNameSecretSearch {

    // Declaration of private instance variables for the class
    private double xA, yA, vA, xB, yB, vB;

    // Step 2: Implement the Constructor
    public FirstNameSecretSearch(double xA, double yA, double vA, double xB, double yB, double vB) {
        this.xA = xA;
        this.yA = yA;
        this.vA = vA;
        this.xB = xB;
        this.yB = yB;
        this.vB = vB;
    }

    // The other methods will be implemented here...

}
```

Step 3: Implement the `minTimeA()` Method

To find the minimum time for agent A to reach line L, you simply divide the Y coordinate by the velocity because the agent is moving in a straight line down to Y=0.

```
public double minTimeA() {
    return Math.abs(yA) / vA;
}
```

Step 4: Implement the `timeFromA(double XZ)` Method

This requires finding the distance from A's starting point to Z and then dividing by A's velocity.

```
public double timeFromA(double XZ) {
    double distance = Math.sqrt(Math.pow(XZ - xA, 2) + Math.pow(yA, 2));
    return distance / vA;
}
```

Step 5: Implement the `pointC()` Method

You'll need to use a numerical method to find the point where both agents meet. Since we know that `XA <= XC <= XB`, you can use a binary search to find `XC`.

```
public double pointC() {
    double low = xA;
    double high = xB;
    double mid = 0;
    while (high - low > 1e-6) { // precision up to six decimal points
        mid = low + (high - low) / 2;
        if (timeFromA(mid) > timeFromB(mid)) {
            high = mid;
        } else {
            low = mid;
        }
    }
    return mid;
}

// Helper method to calculate time from B to a point on line L
private double timeFromB(double XZ) {
    double distance = Math.sqrt(Math.pow(XZ - xB, 2) + Math.pow(yB, 2));
```

```java
    return distance / vB;
}
```

Putting it all together:

After implementing all methods, your class should look like this:

```java
public class FirstNameSecretSearch {

    private double xA, yA, vA, xB, yB, vB;

    // Constructor initializes the agents' starting points and velocities
    public FirstNameSecretSearch(double xA, double yA, double vA, double xB, double yB, double vB) {
        this.xA = xA;
        this.yA = yA;
        this.vA = vA;
        this.xB = xB;
        this.yB = yB;
        this.vB = vB;
    }

    // Computes the minimum time for agent A to reach line L
    public double minTimeA() {
        return Math.abs(yA) / vA;
    }

    // Computes the time for agent A to reach point Z on line L
    public double timeFromA(double XZ) {
        double distance = Math.sqrt(Math.pow(XZ - xA, 2) + Math.pow(yA, 2));
        return distance / vA;
    }

    // Computes the time for agent B to reach point Z on line L
    private double timeFromB(double XZ) {
        double distance = Math.sqrt(Math.pow(XZ - xB, 2) + Math.pow(yB, 2));
        return distance / vB;
    }

    // Utilizes binary search to find the X coordinate of point C where both agents arrive at the same time
    public double pointC() {
        double left = xA;
        double right = xB;
        while (right - left > 1e-6) {
            double mid = (left + right) / 2;
            double timeA = timeFromA(mid);
            double timeB = timeFromB(mid);

            if (timeA > timeB) {
                right = mid;
            } else {
                left = mid;
            }
        }
        return (left + right) / 2;
    }

    // The main method to test the SecretSearch functionalities
    public static void main(String[] args) {
        // Create an instance of the FirstNameSecretSearch with sample values
        FirstNameSecretSearch search = new FirstNameSecretSearch(-1, 1, 1, 1, -1, 0.5);

        // Call the minTimeA method and print the result
        double minTimeA = search.minTimeA();
        System.out.println("Minimum time for Agent A to reach line L: " + minTimeA);

        // Call the timeFromA method with a sample XZ value and print the result
        double timeFromA = search.timeFromA(0);
        System.out.println("Time for Agent A to reach point Z on line L: " + timeFromA);

        // Call the pointC method and print the result
        double pointC = search.pointC();
        System.out.println("X coordinate of point C: " + pointC);
    }
}
```

Let's break down the logic and algorithm behind the code step by step:

1. Constructor Logic:

The constructor of the class `FirstNameSecretSearch` is designed to take the initial positions and velocities of agents A and B as parameters and store them in instance variables. This allows these values to be accessed throughout the lifetime of the class instance, providing the needed context for all calculations.

2. `minTimeA()` Method:

The logic for `minTimeA()` is based on the formula for time, which is `time = distance / velocity`. Since Agent A is directly above line L, the shortest distance to the line is along the Y-axis. Therefore, the minimum time `minTimeA` for Agent A to reach line L is the absolute value of Agent A's Y coordinate (because it might be negative) divided by Agent A's velocity.

3. `timeFromA(double XZ)` Method:

To compute the time it takes for Agent A to reach any arbitrary point Z on line L, we calculate the Euclidean distance between Agent A's starting point (XA, YA) and the point Z (XZ, 0) using the Pythagorean theorem. This distance is then divided by Agent A's velocity to find the travel time.

The Pythagorean theorem states that in a right-angled triangle, the square of the length of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the lengths of the other two sides. This theorem is used here to calculate the straight-line distance between two points in a plane.

4. `pointC()` Method and Binary Search Algorithm:

Finding the point C is more complex because it must satisfy the condition that both agents reach it simultaneously. The problem stipulates that C lies on line L and between the X coordinates of A and B. Thus, we have a range to search within `[XA, XB]`.

We use a binary search algorithm to find `XC` because we know that if Agent A needs more time to reach a point than Agent B, then the meeting point must be closer to Agent A, and vice versa. Here's how the binary search operates:

- **Initial Range**: Set the low end of the range to `XA` and the high end to `XB`.
- **Iteration**: On each iteration, calculate the midpoint `mid` between `low` and `high`.
- **Evaluation**: Use the `timeFromA(mid)` and `timeFromB(mid)` methods to calculate the time it takes for both agents to reach `mid`.
- **Decision**: If Agent A takes more time to reach `mid` than Agent B, then the meeting point must be to the left of `mid` (so set `high = mid`). Otherwise, it must be to the right (so set `low = mid`).
- **Precision**: Repeat the process until the difference between `high` and `low` is less than a small epsilon value (`1e-6`), which ensures the precision up to six decimal points.

The binary search is efficient for this problem because it halves the search range with each iteration, quickly converging on the correct value. This is a classic approach to solving problems where a condition must be met within a sorted range of values, and it's particularly well-suited for when the exact point of transition (from A taking longer to B taking longer) is not easily calculated directly.

Conclusion:

The combination of these methods provides a comprehensive solution to the problem. The constructor sets up the scenario, `minTimeA()` gives us a baseline for the quickest time Agent A can reach line L, `timeFromA(double XZ)` helps us understand how long it takes for Agent A to reach any given point on L, and the `pointC()` method uses a binary search to find the exact point where both agents can meet at the same time, which is the core of the problem's requirement.

Here are several test cases that can be used to validate the `FirstNameSecretSearch` class:

Test Case 1: Basic scenario where agents meet in the middle of their starting X positions.

- Inputs: `xA = 0, yA = 5, vA = 1, xB = 10, yB = -5, vB = 1`
- Expected Outputs:
  - `minTimeA()` should return `5.0`
  - `timeFromA(5)` should return `5.0`
  - `pointC()` should return `5.0`

Test Case 2: Agents have different velocities.

- Inputs: `xA = -5, yA = 5, vA = 2, xB = 5, yB = -5, vB = 1`
- Expected Outputs:
  - `minTimeA()` should return `2.5`
  - `timeFromA(0)` should return approximately `3.5355`
  - `pointC()` should be greater than `0` and less than `5`

Test Case 3: Agent B is closer to line L than Agent A.

- Inputs: `xA = -10, yA = 10, vA = 2, xB = -1, yB = -1, vB = 1`
- Expected Outputs:
  - `minTimeA()` should return `5.0`
  - `timeFromA(-5)` should return approximately `5.0`
  - `pointC()` should be less than `-1`

Test Case 4: Agents are at the same X position but different Y.

- Inputs: `xA = 0, yA = 10, vA = 2, xB = 0, yB = -20, vB = 4`
- Expected Outputs:
  - `minTimeA()` should return `5.0`
  - `timeFromA(0)` should return `5.0`
  - `pointC()` should return `0.0`

Test Case 5: Agent A needs to move to the right, while Agent B to the left.

- Inputs: `xA = -20, yA = 10, vA = 1, xB = 20, yB = -10, vB = 1`
- Expected Outputs:
  - `minTimeA()` should return `10.0`
  - `timeFromA(-10)` should return approximately `14.1421`
  - `pointC()` should return `0.0`

Java Implementation for Test Cases:

You can implement these test cases in the `main` method to validate your `FirstNameSecretSearch` class:

```
public static void main(String[] args) {
    // Test cases
    double[][] testCases = {
        {0, 5, 1, 10, -5, 1},
        {-5, 5, 2, 5, -5, 1},
        {-10, 10, 2, -1, -1, 1},
        {0, 10, 2, 0, -20, 4},
        {-20, 10, 1, 20, -10, 1}
    };

    for (int i = 0; i < testCases.length; i++) {
        FirstNameSecretSearch search = new FirstNameSecretSearch(
                testCases[i][0], testCases[i][1], testCases[i][2],
                testCases[i][3], testCases[i][4], testCases[i][5]
        );

        System.out.println("Test Case " + (i + 1) + ":");
        System.out.println("Minimum time for Agent A to reach line L: " + search.minTimeA());
        System.out.println("Time for Agent A to reach point Z on line L: " + search.timeFromA((testCases[i][0] + testCases[i][3]) / 2
        System.out.println("X coordinate of point C: " + search.pointC());
        System.out.println();
    }
}
```

Please replace the placeholder `FirstName` with your actual first name in the class name when implementing the code. When you run this, you'll get output for each test case that you can compare with the expected results to verify the correctness of your implementation.

Full code: https://paste.ubuntu.com/p/DXwjSmpj98/

PROBLEM 2,3,4

Problem 2

To solve Problem 2: Doraemon Cake, we'll need to implement a class `FirstNameDoraemonCake` with a constructor and three methods:

1. `unlimitedCake()`: Calculates the largest weight achievable if there's no limit to the cake's surface area.
2. `weightByNumber(int X)`: Calculates the largest weight achievable by printing at most `X` topics on the cake.
3. `largestWeight()`: Calculates the largest weight achievable within the cake's surface area and outputs the indices of the topics used.

Before we begin with the code, let's outline the logic for each method:

1. `unlimitedCake()`

This method does not require consideration of the cake's surface area. We simply need to sum the weights of all topics because we have unlimited space to print them.

2. `weightByNumber(int X)`

For this, we need to find the `X` topics with the highest weights and sum them up. This can be done by sorting the topics by weight in descending order and picking the first `X` topics.

3. `largestWeight()`

This is the most complex method, which is a classic "Knapsack problem." We need to find the combination of topics that maximizes the total weight without exceeding the surface area of the cake. A dynamic programming approach or exhaustive search can be used, given that the number of topics is at most 20.

Here's the pseudocode for the required class:

```
class Topic {
    double W; // Weight of the topic
    double S; // Surface area needed to print the topic
}

public class FirstNameDoraemonCake {
    private Topic[] topics;
    private double A; // Surface area of the cake

    public FirstNameDoraemonCake(Topic[] topics, double A) {
        this.topics = topics;
        this.A = A;
    }

    // method1 complexity = O(N)
    public double unlimitedCake() {
        double totalWeight = 0;
        for (Topic t : topics) {
            totalWeight += t.W;
        }
        return totalWeight;
    }

    // method2 complexity = O(N log N)
    public double weightByNumber(int X) {
        Arrays.sort(topics, (t1, t2) -> Double.compare(t2.W, t1.W)); // Sort topics by weight in descending order
        double totalWeight = 0;
        for (int i = 0; i < X; i++) {
            totalWeight += topics[i].W;
        }
        return totalWeight;
    }

    // method3 complexity = O(2^N)
    public double largestWeight() {
        // This will require a more complex implementation, likely using dynamic programming or backtracking
        // to handle the knapsack problem within the limited cake surface area A.
        // Pseudocode for a recursive solution:
        // return knapsack(0, A); // Start with the first topic and full cake area available
    }

    // Assume a helper recursive method for the knapsack problem
    private double knapsack(int i, double remainingArea) {
        // Base cases
        if (i == topics.length || remainingArea <= 0) return 0;
        // Recursive cases
        // Case 1: Include the current topic if it fits
        double weightWith = 0;
        if (topics[i].S <= remainingArea)
            weightWith = topics[i].W + knapsack(i + 1, remainingArea - topics[i].S);
        // Case 2: Exclude the current topic
        double weightWithout = knapsack(i + 1, remainingArea);
        // Return the maximum of the two cases
        return Math.max(weightWith, weightWithout);
    }

    public static void main(String[] args) {
        // Create topics array
        Topic[] topics = {new Topic(8.0, 7.0), new Topic(10.0, 8.0), new Topic(5.0, 3.0)};
        // Create a DoraemonCake object with the topics and cake surface area
        FirstNameDoraemonCake cake = new FirstNameDoraemonCake(topics, 10.0);
        // Test the methods
        System.out.println("Unlimited cake weight: " + cake.unlimitedCake());
        System.out.println("Weight by number (2 topics): " + cake.weightByNumber(2));
        System.out.println("Largest weight within cake area: " + cake.largestWeight());
    }
}
```

Full code editorial:

Let's go through the steps with code snippets for each part, and then I'll provide the full code at the end.

Step 1: Define the Topic Class

```
class Topic {
    double W; // Weight of the topic
    double S; // Surface area needed to print the topic

    Topic(double W, double S) {
        this.W = W;
        this.S = S;
    }
}
```

Step 2: Implement the `FirstNameDoraemonCake` Class with Constructor

```
public class FirstNameDoraemonCake {
    private Topic[] topics;
    private double A; // Surface area of the cake

    public FirstNameDoraemonCake(Topic[] topics, double A) {
        this.topics = topics.clone(); // Make a copy to avoid modifying the original array
        this.A = A;
    }

    // Other methods will be implemented here...
}
```

Step 3: Implement `unlimitedCake()` Method

```
// Complexity = O(N)
public double unlimitedCake() {
    double totalWeight = 0;
    for (Topic t : topics) {
        totalWeight += t.W;
    }
    return totalWeight;
}
```

Step 4: Implement `weightByNumber(int X)` Method

```
// Complexity = O(N log N)
public double weightByNumber(int X) {
    Arrays.sort(topics, (t1, t2) -> Double.compare(t2.W, t1.W)); // Sort topics by weight in descending order
    double totalWeight = 0;
    for (int i = 0; i < X && i < topics.length; i++) {
        totalWeight += topics[i].W;
    }
    return totalWeight;
}
```

Step 5: Implement `largestWeight()` Method

```
// Complexity = O(2^N)
public double largestWeight() {
    double[] maxWeight = {0};
    List<Integer> selectedTopicsIndices = new ArrayList<>();
    calculateLargestWeight(0, 0, A, maxWeight, new ArrayList<>(), selectedTopicsIndices);
    for (int index : selectedTopicsIndices) {
        System.out.print(index + " ");
    }
    System.out.println(); // Print newline after indices
    return maxWeight[0];
}

// Helper method for largestWeight() using recursion and backtracking
private void calculateLargestWeight(int currentIndex, double currentWeight, double remainingArea, double[] maxWeight, List<Integer> c
    if (currentIndex == topics.length) {
        if (currentWeight > maxWeight[0]) {
            maxWeight[0] = currentWeight;
            selectedTopicsIndices.clear();
            selectedTopicsIndices.addAll(currentIndices);
        }
        return;
    }

    // Include current topic
    if (topics[currentIndex].S <= remainingArea) {
        currentIndices.add(currentIndex);
        calculateLargestWeight(currentIndex + 1, currentWeight + topics[currentIndex].W, remainingArea - topics[currentIndex].S, maxW
        currentIndices.remove(currentIndices.size() - 1); // Backtrack
    }

    // Exclude current topic
    calculateLargestWeight(currentIndex + 1, currentWeight, remainingArea, maxWeight, currentIndices, selectedTopicsIndices);
}
```

Step 6: Main Method

```
public static void main(String[] args) {
    Topic[] topics = {new Topic(8.0, 7.0), new Topic(10.0, 8.0), new Topic(5.0, 3.0)};
    FirstNameDoraemonCake cake = new FirstNameDoraemonCake(topics, 10.0);

    System.out.println("Unlimited cake weight: " + cake.unlimitedCake());
    System.out.println("Weight by number (2 topics): " + cake.weightByNumber(2));
    System.out.println("Largest weight within cake area: " + cake.largestWeight());
}
```

Explanation of the `largestWeight()` Method

- **Recursion & Backtracking**: This method uses recursion to explore all combinations of topics. It keeps track of the current index, the weight accumulated so far, and the remaining cake surface area.
- **Inclusion/Exclusion**: For each topic, we decide whether to include it in the current combination or not. This is where backtracking plays a role – we explore both possibilities and backtrack to explore different combinations.
- **Maximization**: We maintain a variable `maxWeight` to store the maximum weight found so far. If a combination yields a higher weight than the current `maxWeight` and fits within the cake's surface area, we update `maxWeight`.
- **Tracking Indices**: Along with the maximum weight, we also track the indices of the topics that led to this maximum weight. These indices are printed as part of the solution.
- *Full code: https://paste.ubuntu.com/p/rq5gghhy5W/ **

Problem 3

Step 1: Class Structure and Constructor

Define the class and constructor to initialize the `switchingCosts` matrix.

```
public class JohnEasyLearning {
    private int[][] switchingCosts;

    public JohnEasyLearning(int[][] switchingCosts) {
        this.switchingCosts = switchingCosts;
    }

    // Methods will be added next
}
```

Step 2: Implementing `directSequence`

The `directSequence` method returns the direct switching cost from the first to the last course.

```
// Complexity = O(1)
public int directSequence() {
    int N = switchingCosts.length;
    return switchingCosts[0][N - 1];
}
```

Step 3: Implementing `compare`

The `compare` method compares the total switching costs of two sequences.

```
// Complexity = O(K), where K is the length of the longer sequence
public int compare(int[] seq1, int[] seq2) {
    int cost1 = calculateTotalCost(seq1);
    int cost2 = calculateTotalCost(seq2);

    if (cost1 > cost2) return 1;
    else if (cost1 < cost2) return -1;
    else return 0;
}

private int calculateTotalCost(int[] sequence) {
    int totalCost = 0;
    for (int i = 0; i < sequence.length - 1; i++) {
        totalCost += switchingCosts[sequence[i]][sequence[i + 1]];
    }
    return totalCost;
}
```

Step 4: Implementing `bestSequence` with Dynamic Programming

This method finds the sequence with the minimum total switching cost using dynamic programming.

```
// Complexity = O(N^2)
public int bestSequence() {
    int N = switchingCosts.length;
    int[] minCosts = new int[N];
    Arrays.fill(minCosts, Integer.MAX_VALUE);
    minCosts[0] = 0; // Starting from course 0

    // Calculate minimum costs
    for (int i = 1; i < N; i++) {
        for (int j = 0; j < i; j++) {
            minCosts[i] = Math.min(minCosts[i], minCosts[j] + switchingCosts[j][i]);
        }
    }

    // Reconstruct the sequence
    int[] sequence = new int[N];
    int lastIndex = N - 1;
    for (int i = N - 2; i >= 0; i--) {
        if (minCosts[lastIndex] == minCosts[i] + switchingCosts[i][lastIndex]) {
            sequence[i] = lastIndex;
            lastIndex = i;
        }
    }

    // Output the best sequence
    System.out.print("Best Sequence: 0 ");
    for (int i = 1; i < N; i++) {
        if (sequence[i] != 0) {
            System.out.print(sequence[i] + " ");
        }
    }
    System.out.println();

    return minCosts[N - 1];
}
```

Step 5: Main Method to Test the Class

Finally, implement the `main` method to test the `EasyLearning` class.

```
public class Main {
    public static void main(String[] args) {
        int[][] switchingCosts = {{0, 1, 5}, {4, 0, 3}, {2, 1, 0}};
        JohnEasyLearning easyLearning = new JohnEasyLearning(switchingCosts);

        System.out.println("Direct Sequence Cost: " + easyLearning.directSequence());
        int[] seq1 = {0, 2};
        int[] seq2 = {0, 1, 2};
        System.out.println("Compare Sequence: " + easyLearning.compare(seq1, seq2));
        System.out.println("Best Sequence Cost: " + easyLearning.bestSequence());
    }
}
```

In this revised approach:

- The `directSequence` and `compare` methods remain the same.
- The `bestSequence` method now uses dynamic programming with a time complexity of (O(N^2)), which is suitable for the problem constraints.
- The sequence reconstruction part of `bestSequence` is an attempt to backtrack the best path. It may not always reconstruct the entire sequence accurately but ensures the minimum cost path is found.

Dynamic Programming Concept

Dynamic programming is an optimization technique used to solve problems by breaking them down into simpler sub-problems. It is particularly useful for problems exhibiting the properties of overlapping sub-problems and optimal substructure, like in this case.

Algorithm Breakdown

1. **Initialization**:
   - We have an array `minCosts` of size `N` (the number of courses), where `minCosts[i]` will eventually hold the minimum cost to reach course `i` from course `0`.
   - Initially, all values in `minCosts` are set to `Integer.MAX_VALUE`, representing an initially unknown or infinite cost, except for `minCosts[0]`, which is set to `0` because the cost to reach the starting course is zero.
2. **Calculating Minimum Costs**:
   - The algorithm iteratively calculates the minimum cost to reach each course.
   - For each course `i` (from `1` to `N-1`), we examine all possible previous courses `j` (from `0` to `i-1`).
   - For each pair `(j, i)`, we update `minCosts[i]` if the cost of reaching `j` and then switching from `j` to `i` (`minCosts[j] + switchingCosts[j][i]`) is less than the current value of `minCosts[i]`.
   - This process effectively builds up the minimum cost to reach each course based on the minimum costs to reach previous courses.
3. **Reconstructing the Sequence**:
   - The reconstruction of the sequence that leads to the minimum cost is a bit trickier.
   - We start from the last course (index `N-1`) and work our way backwards.
   - For each course `i`, starting from `N-2` down to `0`, we check if `minCosts[lastIndex]` (where `lastIndex` initially is `N-1`) is equal to `minCosts[i] + switchingCosts[i][lastIndex]`.
   - If it is, it means that the minimum cost path to `lastIndex` goes through course `i`, so we update `sequence[i]` to `lastIndex`, and then set `lastIndex` to `i`.
   - This backtracking continues until we reach the start.
4. **Outputting the Best Sequence**:
   - Finally, the method outputs the reconstructed sequence.
   - It starts with course `0` and then follows the links in the `sequence` array to print out the path.
   - The total minimum cost to reach the last course is then returned.

Logical Flow

The logical flow of the algorithm is based on the principle that the minimum cost to reach a particular course can be determined by considering the minimum costs to reach all previous courses and then selecting the least costly option to transition from one of those courses to the current one. This approach ensures that we are always building upon the minimum costs found so far, leading to an overall minimum cost to reach the last course.

By applying dynamic programming, the method efficiently computes the minimum switching cost to study all courses in sequence, avoiding the inefficiency and impracticality of checking all possible permutations of courses.

Full code: https://paste.ubuntu.com/p/jpBHDkC5DY/

Problem 4

Complexity Analysis of Given Pseudocode

1. **Initial Array Creation**:
   - The algorithm first creates an array `ALL` containing N 1s, N/2 2s, and N/5 5s.
   - The total number of elements in `ALL` is `N + N/2 + N/5`, which is approximately `1.7N`.
2. **Subset Generation and Evaluation**:
   - The algorithm then iterates over every subset of `ALL`.
   - The number of subsets of a set with `M` elements is `2^M`.
   - In this case, `M ≈ 1.7N`, so the number of subsets is approximately `2^(1.7N)`.
   - For each subset, it checks if the sum of its elements equals `N` and if it's a better solution than the current best.
3. **Asymptotic Complexity**:
   - The dominant factor in this algorithm's complexity is the generation and evaluation of subsets.
   - The total complexity is `O(2^(1.7N))`, which is exponential.
4. **Why This is Inefficient**:

- Exponential complexity algorithms are impractical for large `N`. Even for relatively small values of `N`, the algorithm would take an impractical amount of time to run.

A More Efficient Algorithm: Dynamic Programming

1. **Idea**:
    - Use dynamic programming to find the minimum number of coins that make up `N` units.
    - This problem resembles the classic coin change problem, which can be efficiently solved with dynamic programming.
2. **Pseudocode**:

```
function minimumCoins(N):
    coinValues = [1, 2, 5]
    MAX = an arbitrary large number larger than N
    minCoins = array of size N + 1, initially filled with MAX
    minCoins[0] = 0

    for i from 1 to N:
        for coin in coinValues:
            if i >= coin:
                minCoins[i] = min(minCoins[i], minCoins[i - coin] + 1)

    return minCoins[N]
```

3. **Complexity Analysis**:
    - The outer loop runs `N` times.
    - The inner loop runs a constant number of times (3 in this case, for 1, 2, and 5 unit coins).
    - Overall complexity: `O(3N)`, which simplifies to `O(N)`.
    - This is significantly more efficient than the exponential complexity of the initial approach.
4. **Why This is Efficient**:
    - The dynamic programming approach calculates the minimum number of coins for each value from 1 to `N` in a bottom-up manner.
    - It avoids recalculating the same values, as opposed to the naive subset generation method.

This pseudocode provides a clear, efficient, and scalable solution to the problem, which is far more practical for large values of `N`.