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));
```

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

Putting it all together:

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

```
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

↳ 3 cells hidden