CSE 551 - Foundations of Algorithms

Assignment 5

Submitted by: Sannidhya Pathania (spathan3)

1

Assuming that the numbers are sorted.

- Number of skiers = n, Number of skis = m;
- The array containing the heights of skiers = skiers[].
- The array containing the heights of skis = skis[].
- The 2D array containing the optimal values = c[][].
- The 2D array containing the positions where the optimal values occur = p[][].

Algorithm

for i := 0 to n **do** ----------------------- *TC = O(nm) and SC = O(nm)*

  for j := 0 to m **do**

    // CASE 1: (i = 0 or j = 0) and (i <= j)

    if (i = 0 **or** j = 0) **and** (i <= j)

    then c[i][j] := 0 ----------------------- *TC = O(1)*

    // CASE 3: i > j >= 0

    else if i > j

    then c[i][j] := ∞ *//infinity* ----------------------- *TC = O(1)*

    // CASE 2: 1 <= i <= j

    else diagonalValue = c[i - 1][j - 1] + abs(skiers[i] - skis[j]) -------------- *TC = O(1)*

      previousValue = c[i][j - 1] -------------- *TC = O(1)*

      if diagonalValue <= previousValue -------------- *TC = O(1)*

      then c[i][j] = diagonalValue

        p[i][j] = "+" *// positive (+) to indicate the diagonal value taken*

      else c[i][j] = previousValue

        p[i][j] = "-" *// negative (-) to indicate previous column value taken*

[2]

We will construct the optimal solution using the 2D arrays *c* and *p* from part 1. We will make recursive calls to the same function and will decide on the basis of the value stored in p. If the value stored is “+” then we will move diagonally, if it is “-” then we will move to the previous column. We will print the answer every time we are taking a diagonal move. This is because we are decrementing ***n*** *(Skier)* only when we have found the optimal solution for that particular skier.

Algorithm

constructOptimalSolution(c, p, n, m, skis, skiers) --------------- *TC = O(m) and SC = O(m)*

if n = 0 **or** m = 0

then return

if p[i][j] = “+” ----------------------- *TC = O(1)*

then constructOptimalSolution(c, p, n - 1, m - 1, skis, skiers)

Print n -> m

else if p[i][j] = “-” ----------------------- *TC = O(1)*

then constructOptimalSolution(c, p, n, m - 1, skis, skiers)

return

//As m is going from m to 0 so m+1 calls so SC = O(m+1) => O(m).

[3]

We are assuming that the numbers are entered in a sorted manner.

IDE used = Visual Studio Code

```
import java.util.Arrays;
import java.util.Scanner;

public class App {
   public static void main(String[] args) throws Exception {
       System.out.println("-----------------------------------");
       System.out.println(" Assignment 5: Dynamic Programming");
       System.out.println("-----------------------------------");

       Scanner sc = new Scanner(System.in);

       System.out.print("Enter the number of Skis: ");
       int m = sc.nextInt();
```

```
        float[] skis = new float[m + 1];
        System.out.println("Enter the heights of Skis");
        // ski i is stored at i position
        for (int i = 1; i <= m; i++)
            skis[i] = sc.nextFloat();

        System.out.print("Enter the number of Skiers: ");
        int n = sc.nextInt();

        float[] skiers = new float[n + 1];
        // skier i is stored at i position
        System.out.println("Enter the heights of Skiers");
        for (int i = 1; i <= n; i++)
            skiers[i] = sc.nextFloat();

        sc.close();

        // PART 1: Algorithm to compute c[i, j] that contains the value indicating the
        // optimal solution
        // Introducing another 2D array (p) containing the positions where the optimal
        // values occur.
        float[][] c = new float[n + 1][m + 1];
        char[][] p = new char[n + 1][m + 1];

        for (int i = 0; i <= n; i++)
            for (int j = 0; j <= m; j++) {
                // CASE 1: (i = 0 or j = 0) and (i <= j)
                if ((i == 0 || j == 0) && i <= j)
                    c[i][j] = 0;

                // CASE 3: i > j >= 0
                else if (i > j)
                    c[i][j] = Integer.MAX_VALUE;

                // CASE 2: 1 <= i <= j
                else {
                    float diag = c[i - 1][j - 1] + Math.abs(skiers[i] - skis[j]);
                    float prev = c[i][j - 1];
                    if (diag <= prev) {
                        c[i][j] = diag;
                        p[i][j] = '+';
                        // positive (+) value to indicate we have taken the diagonal
value
                    } else {
```

```
                        c[i][j] = prev;
                        p[i][j] = '-';
                        // negative (-) value to indicate we have taken the previous
column value
                    }
                }
            }

        System.out.println("-----------------------------------");
        System.out.println("Optimal cost for matching: " + String.format("%.2f",
c[n][m]));
        System.out.println("-----------------------------------");
        System.out.println("Assignments of Skier and Ski are as follows");
        // PART 2: Algorithm that constructs an optimal solution
        constructOptimalSolution(c, p, n, m, skis, skiers);
        System.out.println("-----------------------------------");
    }

    private static void constructOptimalSolution(float[][] c, char[][] p, int i, int j,
float[] skis, float[] skiers) {
        if (i == 0 || j == 0)
            return;
        if (p[i][j] == '+') {
            constructOptimalSolution(c, p, i - 1, j - 1, skis, skiers);
            System.out.println("Skier " + i + " (" + skiers[i] + ") -> Ski " + j + " ("
+ skis[j] + ")");
        } else if (p[i][j] == '-') {
            constructOptimalSolution(c, p, i, j - 1, skis, skiers);
        }
        return;
    }
}
```

Input for Code:

Let the 12 Skis be:

3, 5, 8, 10, 11, 12, 13, 14, 16, 17, 18, 21

Let the 10 Skiers be:

2.2, 5.4, 7, 9.7, 10.5, 12.4, 13, 15.3, 16.7, 20

Output from Code:

Optimal cost for matching: 5.40

Assignments of Skier and Ski are as follows

Skier 1 (2.2) -> Ski 1 (3.0)

Skier 2 (5.4) -> Ski 2 (5.0)

Skier 3 (7.0) -> Ski 3 (8.0)

Skier 4 (9.7) -> Ski 4 (10.0)

Skier 5 (10.5) -> Ski 5 (11.0)

Skier 6 (12.4) -> Ski 6 (12.0)

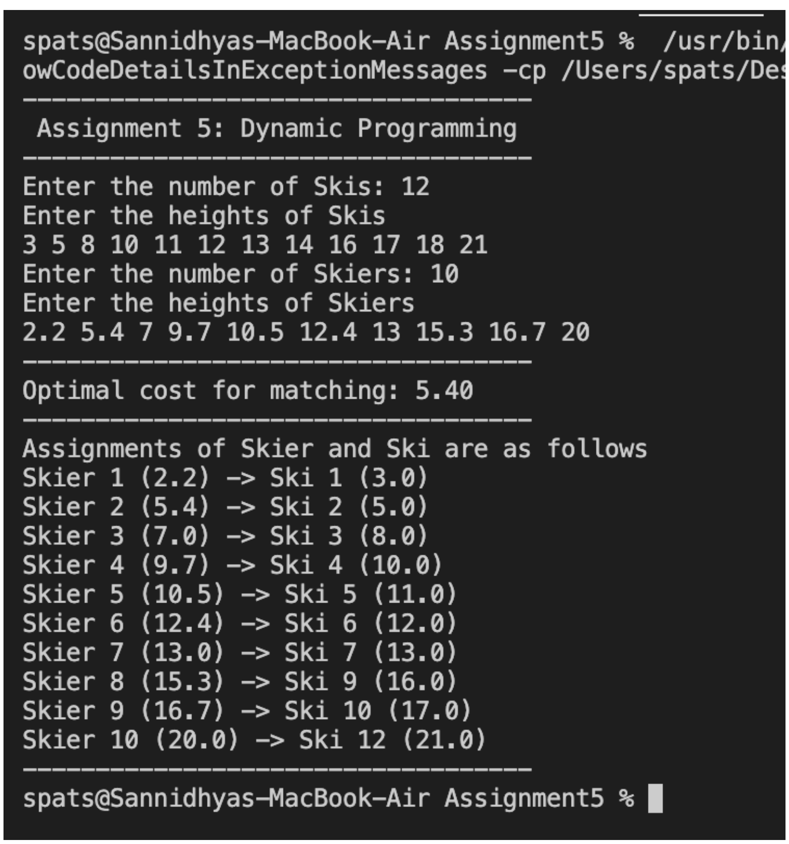
Skier 7 (13.0) -> Ski 7 (13.0)

Skier 8 (15.3) -> Ski 9 (16.0)

Skier 9 (16.7) -> Ski 10 (17.0)

Skier 10 (20.0) -> Ski 12 (21.0)

Screenshot of output:



```
spats@Sannidhyas-MacBook-Air Assignment5 %  /usr/bin/
owCodeDetailsInExceptionMessages -cp /Users/spats/Des
-----------------------------------
 Assignment 5: Dynamic Programming
-----------------------------------
Enter the number of Skis: 12
Enter the heights of Skis
3 5 8 10 11 12 13 14 16 17 18 21
Enter the number of Skiers: 10
Enter the heights of Skiers
2.2 5.4 7 9.7 10.5 12.4 13 15.3 16.7 20
-----------------------------------
Optimal cost for matching: 5.40
-----------------------------------
Assignments of Skier and Ski are as follows
Skier 1 (2.2) -> Ski 1 (3.0)
Skier 2 (5.4) -> Ski 2 (5.0)
Skier 3 (7.0) -> Ski 3 (8.0)
Skier 4 (9.7) -> Ski 4 (10.0)
Skier 5 (10.5) -> Ski 5 (11.0)
Skier 6 (12.4) -> Ski 6 (12.0)
Skier 7 (13.0) -> Ski 7 (13.0)
Skier 8 (15.3) -> Ski 9 (16.0)
Skier 9 (16.7) -> Ski 10 (17.0)
Skier 10 (20.0) -> Ski 12 (21.0)
-----------------------------------
spats@Sannidhyas-MacBook-Air Assignment5 %
```

4

Time complexity (TC): O(nm)

- Taking inputs for Skiers and Skis take O(n+1) and O(m+1) => **O(n) + O(m)**
- For part 1 of the algorithm, TC is O((n+1)(m+1)) => **O(nm)**
- For part 2 of the algorithm, TC is O(m+1) as we are going from j = m to j = 0 => **O(m)**
- All other logic will take constant time **O(1)**.

So overall TC of the program in Q3 => **O(n) + 2*O(m) + O(nm) + O(1) => O(nm)**

Space complexity (SC): O(nm)

- Storing inputs for Skiers and Skis take O(n+1) and O(m+1) => **O(n) + O(m)**
- For part 1 of the algorithm, Space complexity(SC) is O(2*(n+1)(m+1)) as we need to store two 2D arrays c and p of size (n+1) * (m+1) => **O(nm)**
- For part 2 of the algorithm, SC is O(m) as we are making **m** recursive calls to the functions=> **O(m)**
- All other logic will take constant time **O(1)**.

So overall SC of the program in Q3 => **O(n) + 2*O(m) + O(nm) + O(1) => O(nm)**