Elif Akgün
1801042251

CSE 321 HW5

(1) First month, we can either run our business from office in NY or from office in SF. Therefore, I calculate both costs; if we begin from NY or begin from SF. I added the each costs on the path. When I choose office, I think like that: Which office would be more cost effective to come from? So thinking like this, I decide the path and I add previous path to current index. Last I choose the minimum cost path.

For example:

NY = [1, 3, 20, 30] , SF = [50, 20, 2, 4]

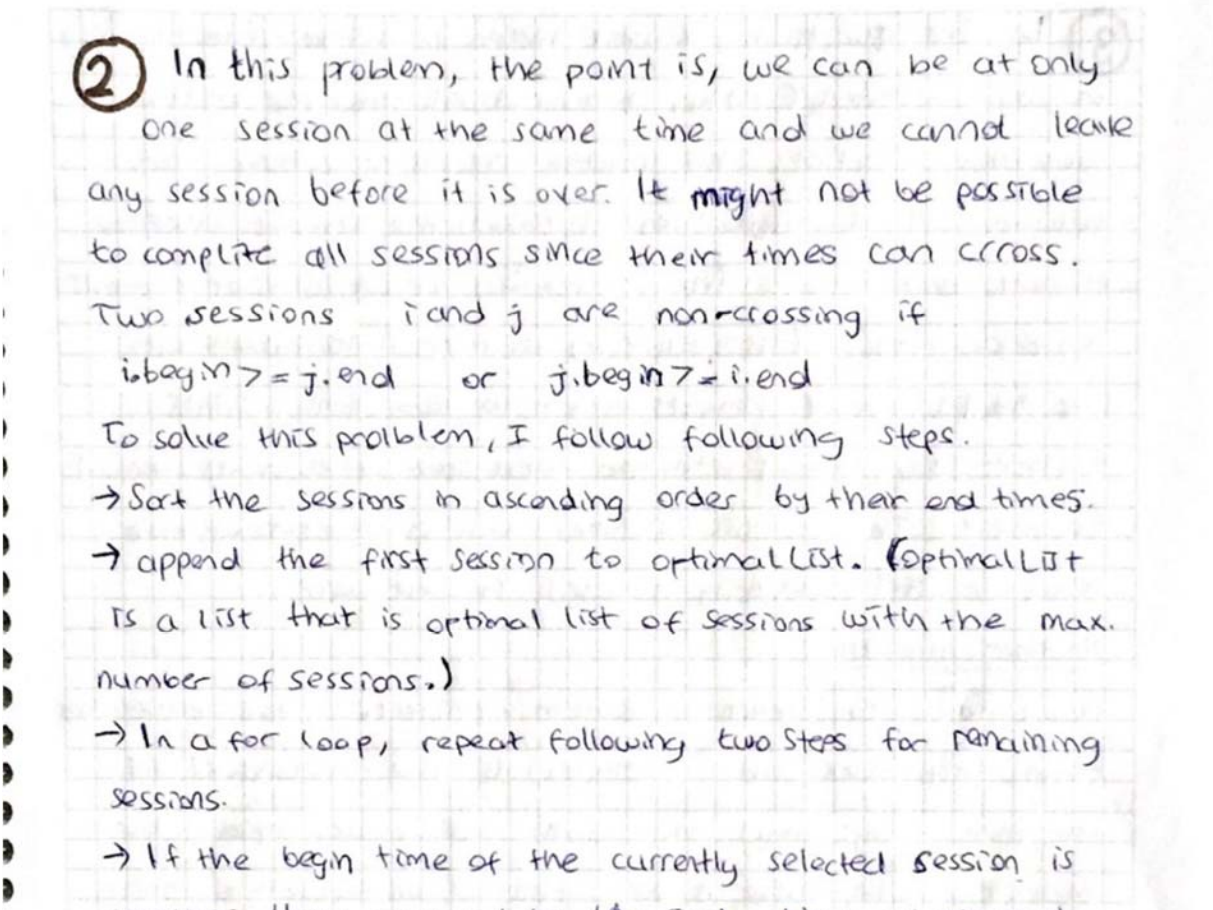
costNY = 1 , costSF = 50, temp = costNY = 1

```
for i=1 → i=NY.length
    i=1
    costNY = 3 + min(1, 50+10) = 3+1=4
    costSF = 20 + min(50, 1+10) = 20+11 = 31
    temp = 4

    i=2
    costNY = 20 + min(4, 31+10) = 24
    cost SF = 2 + min(31, 4+10) = 16

    i=3
    cost NY = 30 + min(24, 16+10) = 30+24 = 54
    cost SF = 4 + min(16, 24+10) = 16+4 = 20
```

return min(20,54) → $\boxed{20}$ → min cost.

* In my algorithm there is a for loop from 1 to NY.length. Assume that NY.length = n. So its complexity is <u>O(n)</u>. (worst case)

#



(2) In this problem, the point is, we can be at only one session at the same time and we cannot leave any session before it is over. It might not be possible to complite all sessions since their times can cross.

Two sessions i and j are non-crossing if
i.begin >= j.end or j.begin >= i.end

To solve this problem, I follow following steps.

→ Sort the sessions in ascending order by their end times.

→ append the first session to optimalList. (optimalList is a list that is optimal list of sessions with the max. number of sessions.)

→ In a for loop, repeat following two steps for remaining sessions.

→ If the begin time of the currently selected session is greather than or equal to the end time of previously selected session, append it to optimalList.

→ Select the next session in the list.

For example

sessions = [[5,9],[1,2],[3,4],[0,6],[5,7],[8,9]]

sorted List = [[1,2],[3,4],[0,6],[5,7],[5,9],[8,9]]

optimal List = [[1,2]]

2<3 ✓ optimal List = [[1,2],[3,4]]

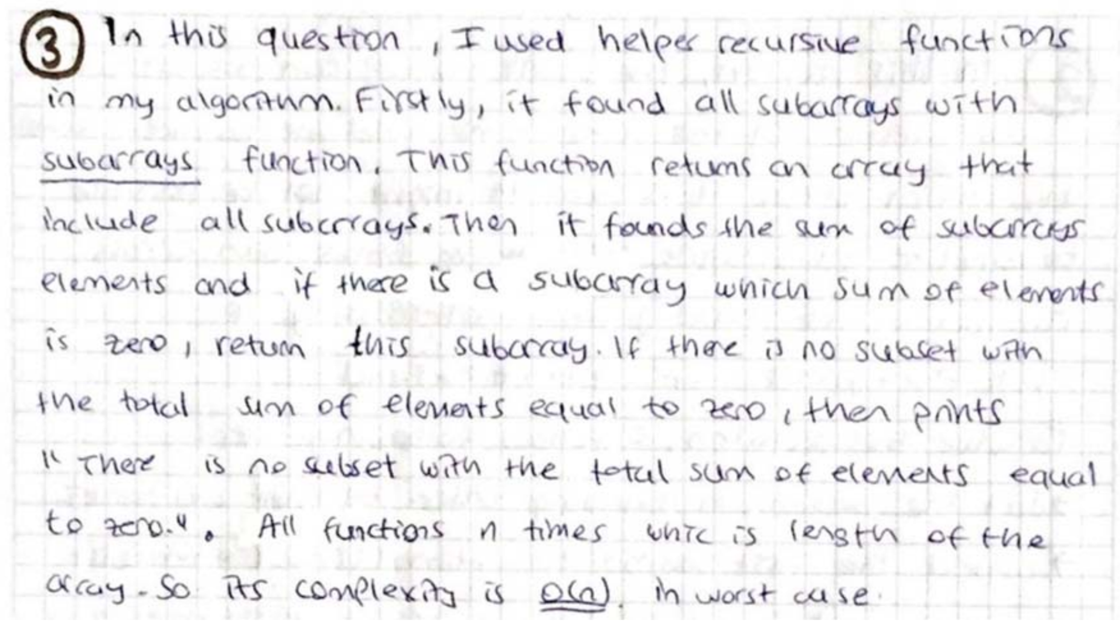
4<5 ✓ optimalList = [[1,2],[3,4],[5,7]]

7<8 ✓ optimalList = [[1,2],[3,4],[5,7],[8,9]]

$\boxed{\text{optimalList} = [[1,2],[3,4],[5,7],[8,9]]}$

* Sort() function's complexity is O(nlogn).
optimalListOfSessions() function's complexity is O(n) because for loop (1 to n).
O(n) + O(nlogn) ∈ O(nlogn)
So the worst case complexity is $\boxed{O(n\log n)}$

#



3 In this question, I used helper recursive functions in my algorithm. Firstly, it found all subarrays with subarrays function. This function returns an array that include all subarrays. Then it founds the sum of subarrays elements and if there is a subarray which sum of elements is zero, return this subarray. If there is no subset with the total sum of elements equal to zero, then prints "There is no subset with the total sum of elements equal to zero.". All functions n times whic is length of the array. So its complexity is $O(n)$ in worst case.

#

#

#

#

#

#

#

#

#

#

#

#

#

#

#

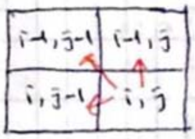
#

4 In sequence allignment, two sequences can be alligned by writing then across a page in two rows. Similar characters are placed in the same column, and non-identical's can either be placed in the same column as a mismatch or against a gap in the other sequence. Sequences that are aligned in this manner are sated to be similar. Sequence alignment is useful for discovering functional and structural information in the biological sequences.

⇒ m= length of Sequence 1

n= length of Sequence 2

The computation is arranged into an (m+1)×(n+1) matrix where entry (i,j) contains similarity between Seq 1 and Seq2. my algorithm computes the value of entry (i,j) by looking at just three previous entries.



```
arr[i][j] = max ( max [arr[i-1][j-1] + mismatch_score,
                       arr[i-1][j] + gap_score ],
                  arr[i][j-1] + gap_score )
```

arr(i)(j) is minimum cost.

* In this algorithm, I used 2D array and it has inner for loop.

```
for( 1→ m ){
    for( 1→ n ){
    } }...
```

So its complexity is O(m*n) in worst case

#

For example

Sequence 1: TREE

Sequence 2: DEER

	-	T	R	E	E
-	0	-1	-2	-3	-4
D	-1	-2	-3	-4	-5
E	-2	-3	-4	-5	-2
E	-3	-4	-1	-2	-3
R	-4	-5	-2	1	$\boxed{0}$

→ arr[4][4] = 0 = min cost

s1_last = [0, 95, 95, 95, 84, 82, 69, 69, 95]

$$s1\_last = [\ ,\_,\_,\underset{\uparrow i=3}{\_},\underbrace{\underset{\uparrow i=4}{T},R,E,E,\_}]$$

s2_last = [0, 95, 95, 95, 95, 68, 69, 69, 82]

$$s2\_last = [\ ,\_,\_,\_,\underset{\uparrow i=3}{\_},\underbrace{\underset{\uparrow i=4}{D},E,E,R}]$$

when $\underset{i=3}{\downarrow}$ both sequence's char equals '_'. So:

write from $(i+1)^{th}$ to last index.

```
∴   T R E E _
    | | | | |
    _ D E E R
```

$$2*2+1*(-2)+2*(-1)$$

$$= 4-2-2 = 0 \ //$$ → cost

#

(5) In my algorithm, I followed, following steps to calculate the sum of the array with the minimum number of operations.

→ Do following steps while array's length bigger than one.

→ sort the array.

→ Sum the arr[0] and arr[1] (sum = arr[0] + arr[1]

→ pop first two element

→ insert the sum to first index.

* I sort the array's elements for each iteration. Because summing the smaller elements firstly is more cost effective. An example can help to understand:

arr = [1, 2, 3, 4]

1+2 = 3 → 3 operations

arr = [3, 3, 4]

3+3 = 6 → 6 operations

arr = [4, 6]

4+6 = 10 → 10 operations

$3+6+10 = \underline{\underline{19}}$ → # of operations

arr = [10] → Sum of array

- - - - - - - - - -

arr = [4, 3, 2, 1]

4+3 = 7 → 7 operations

arr = [7, 2, 1]

7+2 = 9 → 9 operations

arr = [9, 1]

9+1 = 10 → 10 operations

$10+9+7 = \underline{\underline{26}}$ operations

arr = [10] → sum of array

$19 < 26$

* My algorithm include a while loop from n to 1. And it include sort() function in while loop. So its complexity is:
$O(n) * O(n \log n) = \underline{O(n^2 \log n)}$
in worst case.