Gebze Technical University
Department of Computer Engineering
CSE 321 Introduction to Algorithm Design
Fall 2020
Final Exam (Take-Home)
January 18th 2021-January 22nd 2021

Student ID and Name	Q1 (20)	Q2 (20)	Q3 (20)	Q4 (20)	Q5 (20)	Total
Nevzat Seferoglu 171044024						

Read the instructions below carefully

- You need to submit your exam paper to Moodle by January 22nd, 2021 at 23:55 pm as a single PDF file.

- You can submit your paper in any form you like. You may opt to use separate papers for your solutions. If this is the case, then you need to merge the exam paper I submitted and your solutions to a single PDF file such that the exam paper I have given appears first. Your Python codes should be in a separate file. Submit everything as a single zip file. Please include your student ID, your name and your last name both in the name of your file and its contents.

Q1. Suppose that you are given an array of letters and you are asked to find a subarray with maximum length having the property that the subarray remains the same when read forward and backward. Design a dynamic programming algorithm for this problem. Provide the recursive formula of your algorithm and explain the formula. Provide also the pseudocode of your algorithm together with its explanation. Analyze the computational complexity of your algorithm as well. Implement your algorithm as a Python program. **(20 points)**

Q2. Let $A = (x_1, x_2, \ldots, x_n)$ be a list of n numbers, and let $[a_1, b_1], \ldots, [a_n, b_n]$ be n intervals with $1 \leq a_i \leq b_i \leq n$, for all $1 \leq i \leq n$. Design a divide-and-conquer algorithm such that for every interval $[a_i, b_i]$, all values $m_i = \min\{x_j \mid a_i \leq j \leq b_i\}$ are simultaneously computed with an overall complexity of $O(n\log(n))$. Express your algorithm as pseudocode and explain your pseudocode. Analyze your algorithm, prove its correctness and its computational complexity. Implement your algorithm using Python. **(20 points)**

Q3. Suppose that you are on a road that is on a line and there are certain places where you can put advertisements and earn money. The possible locations for the ads are x_1, x_2, …, x_n. The length of the road is M kilometers. The money you earn for an ad at location x_i is $r_i > 0$. Your restriction is that you have to place your ads within a distance more than 4 kilometers from each other. Design a dynamic programming algorithm that makes the ad placement such that you maximize your total money earned. Provide the recursive formula of your algorithm and explain the formula. Provide also the pseudocode of your algorithm together with its explanation. Analyze the computational complexity of your algorithm as well. Implement your algorithm as a Python program. **(20 points)**

Q4. A group of people and a group of jobs is given as input. Any person can be assigned any job and a certain cost value is associated with this assignment, for instance depending on the duration of time that the pertinent person finishes the pertinent job. This cost hinges upon the person-job assignment. Propose a polynomial-time algorithm that assigns exactly one person to each job such that the maximum cost among the assignments (not the total cost!) is minimized. Describe your algorithm using pseudocode and implement it using Python. Analyze the best case, worst case, and average-case performance of the running time of your algorithm. **(20 points)**

Q5. Unlike our definition of inversion in class, consider the case where an inversion is a pair $i < j$ such that $x_i > 2 x_j$ in a given list of numbers $x_1, \ldots, x_n$. Design a divide and conquer algorithm with complexity O(n log n) and finds the total number of inversions in this case. Express your algorithm as pseudocode and explain your pseudocode. Analyze your algorithm, prove its correctness and its computational complexity. Implement your algorithm using Python. **(20 points)**

1)

Nevzat
Seferoglu
171044024

* In this problem, we are asked to implement checking palindrome of given string and returned. In dynamic programming we are breaking it down to a collection of simpler subproblems, solving each of those subproblems just once, and storing their solution using some kind of storage. Each of the subproblem solution indexed typically based on values. Later on, we do not have to recompute those value to fill table. It makes our code efficient.

```
function maxlengthPalindrome ( givingString) :
        size = length of givenString
        table = [size x size] // filled with False initially

         a = 0 , b = 0
         for i to (size-1) :
               table[i][i] = True
               a = b = i
         end for

         for i to (size-2) :
               if givingString[i] equals givingString[i+1] :
                        table[i][i+1] = True
                        a = i , b = i+1
               end if
         end for

         for row from 3 to size :
               for col to size -(row -1) :
                    if givenString[col] equals givenString[col + (row-1)] and
                                        table[col+1][col+(row-2)] :

                        table[col][col + (row -1)] = True
                        a = col , b = col + (row-1)
                    end if
               end for
         end for

         return givenString [a : b +1]
end function
```

$O(n^2)$
complexity
comes from
here.

* Firstly, lets have a table to store our values. if we have string like "ababc" → expected output → "baab", table size is depends on the size of the given string in this situation we have size as six.

j

i

6x6	0	1	2	3	4	5
0	T	F	T	F	F	F
1	F	T	F	F	T	F
2	F	F	T	T	F	F
3	F	F	F	T	F	F
4	F	F	F	F	T	F
5	F	F	F	F	F	T

→ Contains the combinations of the string

a	b	a	a	b	c
0	1	2	3	4	5

* we are storing True or False in given table these True and False will be our value.

* (i, j) → implies that the index i to j in given string is a palindrome if cell (i, j) is True otherwise is not a palindrome.

substring [0][0] = a
" [1][1] = b
" [2][2] = a
" [3][3] = a
" [4][4] = b
" [5][5] = c

Single characters are already palindrome. we need to reflect that to our table as True. State

[i][j] where $\underline{j = i+1}$ } these are will be false for our example string

Algorithm:

```
↳ givingString[i] = givenString[j]
           &&
↳ (table[i+1][j-1] equals True)(query) ?
     if i-j ≥ 2
```

According to algorithm, we only check the first and the last character to be equal. Characters in the middle are <u>already checked</u> at the previous steps.

<u>Recursive Formula:</u>

$D(i, j) = (D(i+1, j-1) \text{ and } K_i == K_j)$

$D(i, j)$ is True if substring $K_i == K_j$ otherwise $D(i, j)$ is False.

<u>Base Case.</u>

$D(i, i) = \text{True}$

$D(i, i+1) = (K_i == K_{i+1})$

<u>Time Complexity</u>

$O(n^2)$

<u>Space Complexity</u>

$O(n^2)$

where n is size of the string length

(3) In this problem, we are asked to implement the algorithm that increase the advertisement income. There are some restrictions. these are we have to place to each ad at least four (4) unit intervals from each other. we also know the length of the entire road. As I expain at question one, dynamic programming preserve us to calculate values and appropriote solution again and again. For this situation we have the road and we need to simulate this road as we do in real case. Then at each kilometer, we need to do some logic based on designed algorithm. Logic operate total income of advertisement. At the end of our journey, we will already calculate the max income at the end of our stange type. Lets look at the algorithm.

```
function maxAdIncome(x, ad, m, size, dist):

    adIncomePerKm = [0 for _ to m]
    realDist = 0

    for i from 1 to m:
        if (realDist < size):
            if (x[realDist] not equals i):
                adIncomePerKm[i] = adIncomePerKm[i-1]
            else:
                if (dist > i):
                    adIncomePerKm[i] = max(adIncomePerKm[i],
                                           ad[realDist])

                else:
                    adIncomePerKm[i] = max(adIncomePerKm[i-dist-1] +
                                           ad[realDist],
                                           adIncomePerKm[i-1])
                realDist = realDist + 1

                end if-else
            end if-else
        else:
            adIncomePerKm[i] = adIncomePerKm[i-1]
```

Explanation:

For every single kilometer on the road, we need to investigate whether current km is on proper option for our advertisement.

if not proper:

maximum income that taken till that kilometer will be the same as maximum income generated one kilometer before.

if it is proper:

1) we put the advertisement, do not care in previous kilometers, and increase the income of the placed advertisement.

2) Do not care current ad. Therefore;

(i → current km) index will be i-dest-1

adIncomePerKm[i] ← maximum of two different
↳ not include previous km income + ad[i] —— current km income
→ previously gained income, where i is

Scanned with CamScanner

Time Complexity :

O(m) where m is the road distance.

Auxiliary Space :

O(m) → single

Recursive Formula

* Because of the reasons that I explain previously. Formula will be;

$$road\text{-}Sum[i] \longleftarrow max\left(\underbrace{income[i]}_{\text{y advertisement incomes at each kilometer}} + \underbrace{roadSum[j]}_{\text{previous calculated roadSum}}, roadSum[i]\right)$$

(5)

In divide and conquer method. In this method as we do in merge sort we are breaking the problem into one or more smaller instance of the same problem and each smaller instance is solved recursively. I keep on breaking the sub problems to a level at which we can solve them directly. Then the solutions of the sub problems are combined to get the solution for the main problem.

```
function merge(arrP0, arrP1):
    inversions = 0
    result = []
    while (len(arrP0) > 0 and len(arrP1) > 0):
        if arrP0[0] ≤ arrP1[0]:
            result.append(arrP0.pop(0))
        else:
            if arrP0[0] > 2 * arrP1[0]:
                inversions += len(arrP0)
            end if
            result.append(arrP1.pop(0))
        end if else.

    if len(arrP0) == 0:
        result = result + arrP1
    else:
        if len(arrP1) == 0:
            result = result + arrP0

    return result, inversions

function mergeSort(arr):
    length = len(arr)
    mid = length // 2
    if length ≥ 2:
        sortedP0, countP0 = mergeSort(arr[:mid])
        sortedP1, countP1 = mergeSort(arr[mid:])
        result, counts = merge(sortedP0, sortedP1)
        return result, counts + countP0 + countP1
    else
        return arr, 0

function numberOfInversion(a):
    resultArray, inversions = mergeSort(a)
    return inversions.
```

Explanation

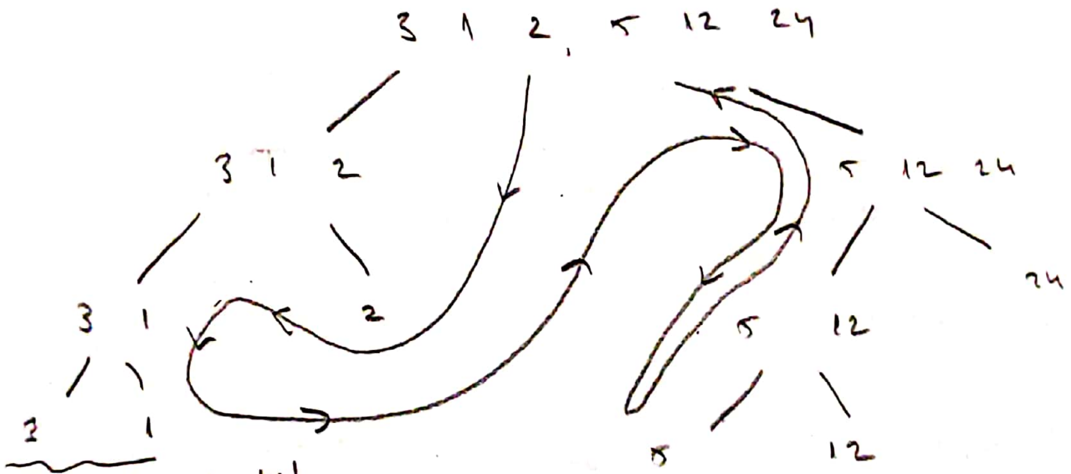
* In algorithm I used divide and conquer method. In merge sort we normally divide the till a number remain at a level. Then we check the relation of two number without changing the location but the conquering has to continue. Each subarray intervals will be checked after the point that all elements are queried. But we donot have to consider all elements each other.

In algorithm from deepest subarray (leftone) will be checked with its neighbour. after that layer. Another subarray takes that result and go through actual array. After left part done the right part will be executed with same method

* At each level we are going to check arrP0[0] and arrP1[0], if the situation is satisfied, we pop the zero index from left subarray which is arrP0, otherwise we pop the zero index from right because we donot have to look those elements which are popped. They are not satisfied our solutions.

Example

arr = [3, 1, 2, 5, 12, 24]



will be checked if satisfied intervals released

Then going up if(3 ≤ 0)

if(3 > 2 * 0)

else

result

* I removed the element from the sub arrays, because we donot have to sort. we just need to count intervals.

Complexity

We simply use divide and conquer and the time complexity be nlogn because of the merge sort. O(nlogn)

Space Complexity

I used list and pop handled number from the lis so:

O(logn)

Scanned with CamScanner