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

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 $1 \le a_i \le b_i \le n$, for all $1 \le i \le n$. Design a divide-and-conquer algorithm such that for every interval $[a_i, b_i]$, all values $m_i = \min\{x_j \mid a_i \le j \le b_i\}$ 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)**

CSE 321

INTRODUCTION TO ALGORITHM DESIGN

FINAL EXAM (TAKE-HOME)

DILARA KARAKAŞ
131044010

QUESTION 1:

If we examine the problem requested from us, we notice some situations:

Case 0: (n=0) It is vacuously true that an empty string is a palindrome

Case 1: (n=1) Since a letter by itself is a palindrome, all substrings of length 1 are a palindrome.

Case 2: (n=2) Substrings of length 2 can only be a palindrome if the first and last letters are the same

Case 3. (n=3) Substrings of length 3 are palindromes if the first and last letters are the same. The middle letter doesn't matter because we know that a single letter by itself is a palindrome. It seems like we can determine Case 3 by using the results of Case 1 and Case 2.

Case 4: (n=4) Substrings of length 4 are palindromes if the first and last letters are same. and two remaining letters make up a palindrome.

It seems like for words to be palindromes, the first and last letters have to be the same and the string excluding the first and last letters must also be a palindrome.

Let's examine how we solved this problem with dynamic programming.

→ Maintain a boolean table [n] [n] that is filled in bottom up manner.

→ The value of table [i] [j] is true, if the substring is palindrome, otherwise false.

→ To calculate table [i] [j], check the value of table [i+1] [j-1] if the value is true and str[i] is same str[j], then we make table [i] [j] true.

→ Otherwise, the value of table [i] [j] is made false

→ We have to fill table previously for substring of length=1 and length=2 because as we are finding, if table[i+1][j-1] is true or false, so in case of

(i) length==1, lets say i=2, j=2 and i+1, j-1 doesn't lies between [i][j]

(ii) length==2, lets say i=2, j=3 and i+1, j-1 again doesn't lies between [i][j].

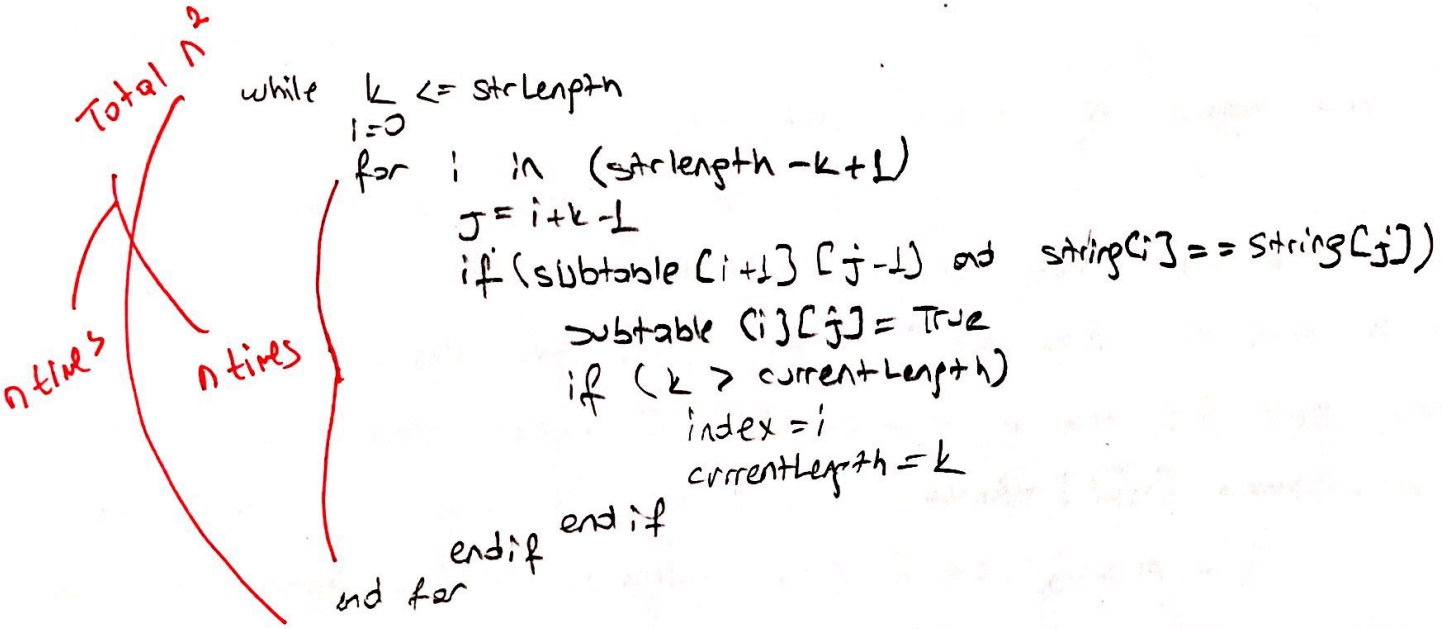
		0	1	2	3	4	5	6
		k	a	r	a	k	a	s
0	k	1	0	0	0	1	0	0
1	a	0	1	0	1	0	1	0
2	r	0	0	1	0	0	0	0
3	a	0	1	0	1	0	1	0
4	k	1	0	0	0	1	0	0
5	a	0	1	0	1	0	1	0
6	s	0	0	0	0	0	0	1

Pseudocode

```
pal_sub_arr ( string):

strLength = len(string)
subtable[strlength][strlength]
current+Length=1
i=0
for i in strLength
    subtable[i][i]=true
end for
index=0
i=0
for i in strlength-1
    if(string[i] == string[i+1])
        subtable[i][i+1] = true
        index=i
        current+Length=2
    end if
end for
k=3
```

n times




```
while k <= strLength
    i=0
    for i in (strLength - k + 1)
        j = i + k - 1
        if (subtable [i+1] [j-1] and string[i] == string[j])
            subtable [i][j] = True
            if (k > currentLength)
                index = i
                currentLength = k
            end if
        endif
    end for
    k++
endwhile
print ( string [index : index + currentLength])
return currentLength // Length for substring
end
```

Time Complexity : $O(n^2)$ Two nested traversals are needed.

QUESTION 2:

The best way to solve this problem in nlogn time is to create a sparse table.

The idea is to precompute a minimum of all subarrays of size 2^j where j varies from 0 to logn. We make a tempArr table. Here tempArr[i][j] contains a min of range starting from i and of size 2^j. For example, tempArr[1][4] contains a min of range [1][8] (starting with 0 and of size 2^3)

The idea is simple, fill in bottom-up manner using previously computed values. For example, to find a min of range [1][8], we can use a min of the following two.

1) Min of range [1][3]
2) Min of range [4][8]

Based on the above example, below is the formula

If arr[tempArr[i][j-1]] <= arr[tempArr[$i+2^{j-1}$][j-1]]

tempArr[i][j] = tempArr[i][j-1]

Else

tempArr[i][j] = tempArr[$i+2^{j-1}$][j-1]

Pseudocode interval(index1, index2) ⇒ O(1)

```
j = int (math.log2(index1 - index1+1))
if (tempArr[index1][j] <= tempArr[index2 - (1<<j)+1][j]
    return tenArr[index1][j]
else
    return tempArr[index2 - (1<<j)+1][j]
endif
```

end

Pseudocode fillTable(arr[0:n]) ⇒ O(n log n)

```
for i in range(0,n)
    tempArr[i][0] = arr[i]
end if
j=0
while ((1<<j) <= n)
    i=0
    while (i + (1<<j)-1) < n)
        if (tempArr[i][j-1] < tempArr[i+(1<<(j-1))][j-1]
            tempArr[i][j] = tempArr[i][j-1]
        else
            tenArr[i][j] = tempArr[i+(1<<j-1)][j-1]
        end if
        i++
    end while
    j++
end while
```

n log n

end

Pseudocode print_min (intervals, arr)

```
fill Table(arr)  → nlogn
for i in range(0, lenIntervals)
    l = intervals[i][0]
    r = intervals[i][1]
    print(interval(l, r))
end for
end
```

n → for loop

1 → print(interval(l, r))

$n + n\log n = O(n \log n)$

Time Complexity = $O(n \log n)$ //

QUESTION 3:

Consider a length of the road is M. The task is to place billboards on the road such that money is max. The possible certain places for billboards are given by number $x_1 < x_2 < \ldots < x_n$, specifying positions in length from one end of the road. If we place a billboard at position x_i, we earne a money of $ri > 0$. There is a restriction that no two billboards can be placed within k kilometers or less that it.

Let maxMoney[i], $1 <= i <= M$, be the max money generated from begining to i kilometers on the road. Now for each kilometers on the road, we need to check whether this kilometer has the option for any billboard, if not then the max money generated till that kilometer would same as max money generated till one kilometer before. But if that kilometer has the option for billboard then we have 2 options:

(i) Either we will place the billboard, ignore the billboard in previous t kilometers, and add the money of the billboard placed.

Ignore this billboard. So maxMoney[i] = maxMoney[i-t-1]+ money[i]

, maxMoney[i-1]

Pseudocode maxMoney(M, locations, money, distance)

```
n = len(location)
maxMoney = [0] * (n+1)
temp = 0
for i in range(1, M+1)
    if (temp < n)
        if (location[temp] == i)
            maxMoney[i] = max(maxMoney[i-1], money[temp])
        else
            maxMoney[i] = max(maxMoney[i - distance - 1]
                              + money[temp], maxMoney[i-1])
        end if
        temp++
    else
        maxMoney[i] = maxMoney[i-1]
    end if
end for
return maxMoney[M]
end
```

M times

Time Complexity: $O(m)$ ↳ length of road.

QUESTION: 4:

First of all, I will talk a little bit about my algorithm. I created the person and job class to solve the problem. I randomly allocated and created the time that a person can do any job in the class. Then I made a distribution of work by putting myself as an employer. Let's think of it this way, we have a network job. We commissioned whoever did this as soon as possible. At the same time, I kept an object that keeps the status of doing business or not in the class, so that two jobs are not for one person. In this way, we were able to minimize the work.

Pseudocode. Assignment (people, jobs)

fill People Jobs (people, jobs) → O(n.m)

i=0

while i < length(people)

n times

index = findIndex (people, i) → O(n)

people[index].set-job(jobs[i])

i++

end while → O(n.m + n²) = O(n²)

end

Pseudocode fillPeopleJobs (people , jobs)

```
i=0
while  i < length (people)  → n is people[] length
        j=0
        while j < length (job)  → m is job[] length
                people[i].add_jobs(jobs[j])  → O(1)
                j++
        end while
        i++
end while                    → O(n.m)
end
```

n. times

Pseudocode findIndex (people, index)

```
i=0
minValue
returnIndex
while  i < length (people)  → n is people[] length
        if people[i].hasjob == false
                if people[i].jobs[index] < minValue → O(1)
                        returnIndex = i
                        minValue = people[i].jobs[index]
                end if
        end if
        i++
end while                    → O(n)
return returnIndex
end
```

n times

→ The while loop of the findIndex function always returns n times. There is no case that could break this. So, $O(n)$

→ With some logic, the time complexity of the fillPeopleJobs function is $O(n^2)$

→ finaly, the while loop of the function we need to examine returns n times. But it calls findIndex function on every return. So $O(n^2)$ comes from that loop. The sum is $O(n^2+n^2) = O(n^2)$ with $O(n^2)$ coming from the fillPeopleJobs function called before loop.

So Time complexity = $O(n^2)$

QUESTION 5:

The way I solved this problem was with BST. BST controls were added according to our requirement and a specialized BST insert was prepared. But the conditions we added did not change the time complexity of the BST insert. In this way, the insert complexity alone remained $O(\log n)$. Since we do this insert operation for n elements, the time complexity of the problem is $O(n \log n)$.

The first element of the directory is added as root. Subsequent elements are added sequentially with BST. The left child of the BST holds elements smaller than it. Its right child holds elements loger than it.

Our condition is checked for small elements. If the element to be added is twice less than the new element, the value of our global variable is increased. If the element to be added does not have a left child, a node is created and added. If there is, the function is called again on the left child. If there is a right child, it is called again on the right child, provided that no element is added.

In large elements, if there is no right child and if adding is desired, the right child node is created and added to the tree. If there is a right child, the function can be continued on the right child.

```
number =0  // Global

class  Node :

        init (self, data)
           self.left= None
           self. right = None
           self.data_ =data

        insert (self , data, insert_control)
            global number
            if (self. data_)
                if (data < self.data_)

                        if ( 2* data < self.data_)
                              number  += 1

                         if (insert_control)
                            if (self.left is None)

                                  self. left = Node (data)
                                  if (self.right is Not None)
                                       self. right. insert ( data, false)
                            else
                                  self. left.insert (data , insert_control)
                                  if (self.right is Not None)
                                       self. right. insert ( data , false)
                            endif

                        else
                            if (self.left is Not  None)
                               self.left insert (data , insert_control)
                            if (self.right  is Not None)
                               self. right. insert (data,  insert.control)

                        end if
                elif  (data> self. data_)
                        if ( self.right is None)
                            if (insert_control)
                                 self. right = None (data)
                         else
                        endif self.right. insert ( data , insert_control)
                endif
        else
            if (insert_control)
                 self. data_ =data
end     end if
```

```
Pseudo code    inversion ( Arr[0:n]
        root = Node (Arr[0])
        for  i  in  range (1, n)
                root. insert. (arr[i], True) → log n
        end for
end
```

ntimes

Time Complexity : O(n log n)