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
161044056 Muhammed Yasir Fidan						

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)**

Q1) For this question I use a $n \times n$ 2D matrix for DP, where n is the length of giving string. In this matrix if [i][j] is true, it means there is a palindrome sub string where start index i to the index j.

Firstly all [i][i] index is true in this matrix because its represent one element and one character always palindrome, so, for instance if our string is "meee" our matrix is 4×4 because "meee" length 4

	0	1	2	3
0	1			
1		1		
2			1	
3				1

First fill [x][x] indices true. Then we must fill only one side of this diagonal because we are not consider substrings in reverse order.

And we fill this 2D array like these

dp[i][j] true if string[i] == string[j] and
dp[i+1][j-1] == 1

After filling Array one side by using this formula our 2D matrix become:

	0	1	2	3
0	1	0	0	0
1		1	1	1
2			1	1
3				1

Here 1's represent a valid substring. For example dp[1][3]=1 so str[1:3] is a valid substring. and we can hold max length one and can return it.

The recursive formula for this algorithm like this;
if substring first and last letters same
and check between this two letter string recursively.

```
Check_palindrome(string):
    if:(string length is 0 or 1) return true
    else if:(first and last letters same for given string)
        return check_palindrome(string[1:len(string)-1]
    else:
        return false
```

Recursive step

For instance, if given string "axyxa"

first and last letters same

Recursively check middle string

"xyx"

first and last letter same

recursively check middle string but it has length 1 so return true

But because of we use dynamic programing we don't need always calculate some recursive substring steps, we can just check them in our matrix.

Psuedo code:

```
def  Algorithm (string):
     db = [ ][ ]
     for(i=0; i< len(string); i++) db[i][i] = True

     first_index = 0   length = 1

     for (i=0; i<len(string)-1; i++) begin
          if(string[i] = string[i+1]) begin
               db[i][i+1] = True
               first_index = i
               length = 2
          end if
     end for

     for (i=2; i<len(string); i++) begin
          for(j=0; j<len(string) -i; j++) begin
               temp = i+j
               if ( String [j] = string[temp]  &&  db[j+1][temp-1] = 1)
                    db[j][temp] = True
                    if (i +1 > length) begin
                         first_index = j
                         length = i+1
                    end if
               end if
          end for
     end for

     return string[first_index : first_index + length]

end
```

As I mentioned before firstly I create my 2D DP matrix and fill diagonal as true. The first for loop deals with substrings of length 2. Other loop deals with substrings of length bigger than 2 by using recursive step I mentioned before but instead of recursive I use my DP, Also I hold substring first index and length information in first_index and length variable finally I return valid max length substring.

Time complexity for this algorithm $O(n^2)$

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)**

First I create a 2D array for soluing this problem in O(nlogn). My array size is n×(log(n)+1) where n is number of elements in list.

pseudocode for creating table:

```
def create_table(A)
    table ← [len(A)][log(len(A))+1]
    for i←0 to len(A) do table[i][0] ← i
    j←1
    while len(A) >= pow(2,j) do
        i←0
        while pow(2,j)+i-1 < len(A) do
            if A[table[i+pow(2,j-1)][j-1]] > A[table[i][j-1]]
                table[i][j] = table[i][j-1]
            else
                table[i][j] = table[i+pow(2,j-1)][j-1]
            i++
        end while
        j++
    end while
    return table
end
```

Let me explain algorithm from an example of created table

for example, if my $x = [4, 6, 1, 5, 7, 3]$ my table will be like this;

$\overbrace{\qquad\qquad}^{\lfloor \log(n) \rfloor + 1}$

n

0	0	2
1	2	2
2	2	2
3	3	–
4	5	–
5	–	–

This table give us min element index for between index i and $2^j + i$. for example table[2][1] = 2 because $i = 2$ and $2^j + i = 4$ so it gives [2, 4] interval smallest element index which is 2 because value 1 smaller between index 2 and 4 in array

As you can see table[i][0] = i always because 1 element interval minimum element always return that value index, so;

for i ← 0 to len(A) do table[i][0] = i for this part

The other parts of table fill by using previous table values by using the formula

table[i][j] = min(table[i][j-1], table[i+pow(2, j-1)][j-1])

for example for table[2][2]

table[2][2] = min(table[2][1], table[4][1])

table[2][2] = min(2, 5) = 2

And finding min element in interval by using this table $O(1)$

Pseudocode:

```
def interval(arr, a, b, table)
    size <- b-a+1
    k <- ⌊log2(size)⌋

    return min(arr[table[a][k]], arr[table[a+size-pow(2,k)][k]])
end
```

This algorithm works like that, for example if interval is $[0, 5]$

size $= 6$ and $k = 2$ so it will return

min (arr[table[0][2]], arr[table[2][2]])

So, I use divide and conquerer approach here.

table[0][2] give min element in $[0, 3]$ interval

table [2][2] give min element in $[2, 5]$

So now conquerer part is here. minimum of this two interval will be minimum of $[0,5]$.

Creating table time complexity is $O(n \log(n))$ and finding an intrval min $O(1)$. So finding all n intervals I will create table then use interval algorithm for finding all intervals min element. So total complexity will be $O(n \log n + n) = O(n \log n)$ for this algorithm.

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 $x_1, x_2, \ldots, x_n$. 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)**

Pseudocode:

```
def Algorithm (x, r, n, M)
    dp ← [0, ... M]
    ads ← 0
    for i ← 1 to M do
        if ads < n
            if x[ads] != i
                dp[i] ← dp[i-1]
            else
                if i >= 5
                    dp[i] ← max(dp[i-5] + r[ads], dp[i-1])
                else
                    dp[i] ← r[ads]
            ads++
        else
            dp[i] ← dp[i-1]
    end for
end
```

Let me explain my algorithm.

Firstly I create my 1D array for using in dynamic programming

The array length is M where M is the length of road

and initially first element of array 0. I traverse this arry

by starting index 1 and fill it.

first if ads<n checks if all the ads already placed or not

for example if there is 3 possible ads location (x_1, x_2, x_3) and

we already put ads those 3 places this if return false and in

else statement it just dp[i] ← dp[i-1]

if x[ads] != i check for there is a ads location in i mile.

If this if statement satisfied its mean there is no any ads location

in i. mile, in this case our dp doesn't change so dp[i] ← dp[i-1]

If this' 2 if statement satisfied, its mean we have a ads

place for i. But we have 2 option, either place ads in this place

or ignore it and don't place. so;

if i>=5 dp[i] = max(dp[i-5] + r[ads], dp[i-1])

find optimal option.

if i<5 I am in first 5 miles so there are no ads placed

prior to i mile so just place ads here dp[i] = r[ads]

I use 5 here because my restriction mile is more than 4 so

there must be at least 5 mile between 2 ads.

So recursive formula for this algorithm is;

$$dp[i] = \begin{cases} dp[i-1] & \text{if } ads >= len(x) \\ dp[i-1] & \text{if } x[ads] != i \\ r[ads] & \text{if } i < 5 \\ max(dp[i-5] + r[ads], dp[i-1]) & \text{if } i >= 5 \end{cases}$$

As I mentioned before if $ads >= len(x)$ or $x[ads] != i$ that means there is already ads in all possible positions or there is no possible position for ith mile. In this case our dp will be same so; $dp(i) = dp(i-1)$

dp(i) result will be same dp(i-1) but because we use dynamic programming we don't need calculate dp(i-1) again recursively. We already hold its solution in our dp array.

And if this 2 condition not satisfied and we are in first 5 miles than $dp[i] = r[ads]$, if our mile greater or equal than 5 than $dp[i] = max(dp[i-5] + r[ads], dp[i-1])$

Again, here find result for dp(i) we need to calculate dp(i-5) and dp(i-1) solution. But because of we use dynamic programming we already calculate their solution and store in an array. So we don't need calculate them again thanks to dynamic programming

Because of the (for $i \leftarrow 1$ to M do) part this algorithm has $\Theta(M)$ time complexity where M is the length of the road. Also we have a dp array of size $M+1$ so space complexity will be $\Theta(M)$.

My previous pseudocode return maximum possible earn money. For finding which location selected just start last element of dp and traverse to first element, whenever value changes its mean this position selected. Hold this selected position and jump 4 miles back.

Here is the pseudocode for this:

```
def print_selected_positions(dp):
    selected_miles ← []
    i ← len(dp)-2
    while i >= 0 do
        if dp[i] != dp[i+1]
            selected_miles.add(i+1)
            i ← i-4
        i ← i-1
    end while
end
```

This print selected position algorithm also has $\Theta(M)$ complexity where M is size of dp.