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
Muharrem Ozan Yeşiller 171044033						

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)**

Solutions

Question 1)

First of all, the problem asked from us in this question is that we develop an algorithm that gives the longest palindrome substring of a string.

I approached the problem as follows:
I kept the results of the sub-problems of my big problem in a table, and I thought that the recursive structure could reduce the time complexity. This approach is one of the principles of dynamic programming.

First, I created a boolean table.
If the value in the i and j th index in the table is true, then this substring is palindrome, otherwise (false) it is not.
We must first fill in cases where the lengths of substrings are 1 and 2. Because a string must have at least 2 digits to be palindrome and we make these comparisons as [i + 1] [j-1].

If this algorithm was developed recursively, its complexity would be similar to the fibonacci recursive algorithm.

And the recursive formula would look like this:
T[n] = T[n-1] + T[n-2] that means exponential time. (O(2^n))

However, we can reduce this complexity to O (n ^ 2) by using dynamic programming techniques.

PSEUDO CODE:

```
function algorithm1(str_param)
        // initialize my data
        char_arr <= list(str_param)
        // initialize dynamic programming boolean table
        dbTable_size <= length(char_arr)
        dp_table <= array [dbTable_size] x [dbTable_size]

        longestStr <= “”
        maxLen <= 1
        // first of all filling with True my table
        for i in range(0, dbTable_size)
                dp_table[i][i] <= True
        // If the two strings compared do not break the palindrome, the table continues to fill as true.
        for i in range(0, dbTable_size-1)
                if(charArr[i] equal charArr[i + 1])
                        dp_table[i][i+1] <= True
                        maxLen <= 2

        for index in range(0, dbTable_size)
                for i in range(0, dbTable_size - index)
                         j <= i + index
        // If the two strings compared do not break the palindrome, the table continues to fill as true.
                        if(charArr[i] equal charArr[j] and ( (j-i) less_equal  2
                                                                or dp_table[i+1][j-1])):
                                dp_table[i][j] <= True
                        if (j-i+1 greater maxLen):
                                maxLen <= j - i + 1
                                 longestStr <= str_param[i:j+1]

        if (len(longestStr) equal 0):
                 return "There is not palindrome substring in " + str_param

        return longestStr
```

Question 2)

In this question, we are asked to find the minimum value of the desired range (provided that it is lowerbound <= upperbound) with a divide and conquer algorithm and this algorithm must be O (n logn).

First of all, my algorithm is kept as if the first value of the interval given is virtually the smallest value.

Then the array I have continues by breaking into these intervals, namely a new sub array.

The range in question determines this fragmentation boundary.
In the technique I use while returning by sorting in merge sort, the number we assume as the minimum is updated and returns the lowest.

PSEUDO CODE:

```
function divide(arr, lowerB, upperB, returnVal)

//  if dividing the array is terminated in the range base case
    if (lowerB equal upperB)
        return min(returnVal, arr[upperB])

 //  if dividing the array is terminated in the range base case2
// until last 2 item
    if (upperB - lowerB equal 1)
        if arr[lowerB] less arr[upperB]
          // update minimum
           returnVal <= min(returnVal, arr[lowerB])
        else
          // update minimum
           returnVal <= min(returnVal, arr[upperB])

        return returnVal

// set the middle of current array
    mid <= (lowerB + upperB) // 2
// divide array with recursive call
    returnVal <= divide(arr, lowerB, mid, returnVal)
// divide array with recursive call
    returnVal <= divide(arr, mid + 1, upperB, returnVal)
 // return variable for update to minimum
    return returnVal

function query(arr, queryArr)

    for i in range(0, len(queryArr))
// set the lower and upper bound to divide array
        lowerB <= queryArr[i][0]
        upperB <= queryArr[i][1]
        x <= divide(arr, lowerB, upperB, arr[lowerB])

        print("returnVal of Intervals of [%number, %number] -> %number"->
                                                                (lowerB, upperB, x))
```

Question 3)

The algorithm requested from us in this question is similar to the highway billboard problem.
Let there be a path of length M.
It is an advertisement board placement that will maximize the desired income.
Possible locations for bulletin boards are given by the number x1 <x2 <… .. <xn-1 <xn, which indicates positions in miles measured from one end of the road. If we place a billboard at location Xi, we get ri> 0 revenue. There is a restriction that two billboards cannot be placed at t miles or less.

Let table [], 1 <= i <= m, be the maximum revenue generated from the beginning to i miles on the highway.

For every mile on the freeway, we need to check if that mile has any advertising options.

If not, the max income earned up to that mile is equal to the max revenue earned up to a mile. But we have 2 options for this mile bulletin board:

1) We will either think that there are no billboards in the previous t miles and add the income of the billboards placed.
2) Or we will think of it as table [i] = max (table [i-t-1] + revenues [i], table [i-1]) without considering this advertisement.

PSEUDO CODE: (algorithm is explained above.)

```
function algorithm3(advertisements, revenues, sizeB, M, t)
        table <= [0 for i in range(M+1)]

        j <= 0
        for i in range(1, M + 1)

                if(j less sizeB)
                        if (advertisements[j] not_equal i)
                                table[i] <= table[i - 1]

                        else
                                if (i less_equal t)
                                        table[i] <= max(table[i-1], revenues[j])
                                else
                                        table[i] <= max(table[i-t-1] + revenues[j], table[i-1])

                                j +<= 1

                else
                        table[i] <= table[i - 1]

        return table[M]
```

Time complexity is O(m), where m is the distance of the total road.

Question 5)

I used the merge sort algorithm in my approach in this question.

But we need a mechanism that keeps a duplicate of the extra old array because the algorithm doesn't ask us to sort the numbers.this requires a duplicate array that keeps a backup of the array.

At the time of comparison, if i <j and Left [i]> 2 * Right [J], the count variable I use in the algorithm increases as mid - i. Why mid - i, if it increased by one, merge sort is an effective algorithm, and it may not compare unnecessary numbers, adding how much inversion there will be, will bring us to the correct result. Since merge is sort divide and conquer, I took a divide and conquer approach here.

Merging Algorithm Equations: $T(n) = 2T(n/2) + \theta(n)$

In this case the algorithm provides us with **O(nlogn)** time complexity.

PSEUDO CODE: (algorithm is explained above.)

```
function merge(arr, tempArr, l, m, r)
// initi for merge l j k value and my algorithm count value
  count <= 0

    i <= l
    j <= m
    k <= l
    while (i less_equal (m - 1) and j less_equal r)

// the part requested from us in the problem.
// for the inversions
        if (arr[i] greater 2 * arr[j])
            count +<= m - i
            j +<= 1

        else
            i +<= 1

    i <= l
    j <= m
    k <= l
```

```
    while (i less_equal (m - 1) and j less_equal r)
        if (arr[i] less_equal arr[j])
            tempArr[k] <= arr[i]
            i +<= 1
            k +<= 1
        else
            tempArr[k] <= arr[j]
            k +<= 1
            j +<= 1

    while (i less_equal m - 1)
        tempArr[k] <= arr[i]
        i +<= 1
        k +<= 1

    while (j less_equal r)
        tempArr[k] <= arr[j]
        j +<= 1
        k +<= 1

    // !!! copy back original array to sorted array
   for i in range(l, r + 1)
        arr[i] <= tempArr[i]

    return count

function mergeSort(arr, tempArr, l, r)
    m, count <= 0, 0
    if (r greater l)
        m <= (r + l) // 2
        count <= mergeSort(arr, tempArr, l, m)
        count +<= mergeSort(arr, tempArr, m + 1, r)
        count +<= merge(arr, tempArr, l, m + 1, r)
    return count


function algorithm5(arr)
    tempArr <= [0 for i in range(len(arr))]
    return mergeSort(arr, tempArr, 0, len(arr) - 1)
```