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
171044046 Esra Eryılmaz						

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)**

"Palindrome"

Designing the algorithm:

	s	e	k	a	b	a	k	n
s	1	0	0	0	0	0	0	0
e	0	1	0	0	0	0	0	0
k	0	0	1	0	0	0	1	0
a	0	0	0	1	0	1	0	0
b	0	0	0	0	1	0	0	0
a	0	0	0	0	0	1	0	0
k	0	0	0	0	0	0	1	0
n	0	0	0	0	0	0	0	1

Explanation:

- Every single character makes palindrome. (length 1)
- Same ajacent characters makes palindrome. (length 2)
- For palindrome of size 3 upto the length of the string, Mark substring from i till J as palindrome.

table[i][J] = True, if string[i+1][J-1] is palindrome and char at the begining i matches char at the end J.

Pseudo code :

```
ALGORITHM maxLengthSubarray (string)
    n ← length(string)
    begins ← 0
    ends ← 1
    create table[][] = [False] * n * n
    for i ← 0 to n do                          length 1
        for J ← 0 to n do
            if i == J
                table[i][J] = True
        end for
    end for
    for i ← 0 to n-1 do                        length 2
        if string[i] == string[i+1]
            table[i][i+1] = True
        end if
    end for
    for length ← 3 to n+1 do                   length 3 and above
        for J ← 0 to n-length+1 do
            J = i + length - 1
            if string[i] == string[J] AND table[i+1][J-1] == 1
                table[i][J] = True
                if length > ends
                    begins = i
                    ends = length
                end if
            end if
        end for
    end for
    return string[begins : begins + ends]
end
```

Providing the recursive formula:

$$F(i, j) = \max \begin{cases} table[i][i+1] = True & \text{if } string[i] = string[i+1] \\ table[i][j] = True & \text{if } string[i] = string[j] \;\&\&\; table[i+1][j-1] = 1 \\ False & \text{Otherwise} \end{cases}$$

- The recursive formula comes from the above algorithm, I take the conditions inside the loops and put it into my recursive formula.
- This dynamic programming algorithm takes a bottom up approach, I find out the palindromes from length 1 till the length of the given string.

Computational complexity:

- Time complexity: $O(n^2)$ (where n is the length of the given string)

  Because we need two nested traversals.
  In other words, we have nested loops. so $O(n^2)$

- Space complexity: $O(n^2)$

  Because we need Matrix of size $n*n$ to store the table array.

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)**

Since I have to use divide-and-conquer algorithm and in the question; the intervals are always given ascending order or equal order, I decide to use Quick Sort algorithm which complexity is O(nlogn)
After calling the QuickSort algorithm, I make array of intervals and traverse and print the element in the list according to the minimum value in each intervals.

Pseudocode:

```
QuickSort (A, low, high)
  if (low < high)
    pivot = partition (A, low, high, position)
    Quick Sort (A, low, pivot-1)
    Quick Sort (A, pivot+1, high)
  end if
end
```

```
main ()          # Driver code
    arr = [list of n numbers]
    n = length (arr)
O(nlogn)  Quick Sort (arr, 0, n-1)
    intervals = [list of n intervals]
O(n)  for i in intervals
          print ( arr [ i [0] ] )
                      m_i
```

```
partition (A, low, high, position)
  pivot = A [low]
  right = low
  left = high + 1
  while (right < left)
    repeat right = right + 1 until
                    A [right] >= pivot
    repeat left = left - 1   until
                    A [left] <= pivot
    if (right < left)
      swap (A [left], A [right])
    end if
  end while
  position = left
  A [low] = A [position]
  A [position] = pivot
end
```

- QuickSort algorithm divides the function into two parts on the basis of the partition value and makes two recursive calls.
- For the different low and high value it divides the array into subarrays and partition() function is used to sort the array within the range of low and high.
- My partition function takes low index of the array as the pivot element, and sorts subarrays
- After sorting the list, with using one loop ; I traverse the intervals and print the element in the given list according to the minimum value of the each intervals.

Computational Complexity:

→ Time Complexity : For the Quick Sort part

Each partitioning operation takes $O(n)$ operations. In average, each partitioning divides the array into two parts which sum up to $\log n$ operations. In total we have $O(n\log n)$ operations

For the main part

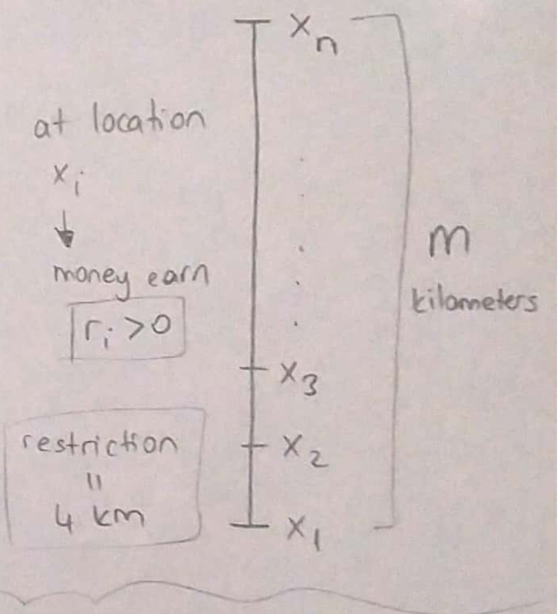
Traverse through the intervals have $O(n)$ operations.

Overall complexity → $O(n\log n) + O(n) = O(n\log n)$ (take bigger)

→ Space Complexity : $O(\log n)$

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)**

Using dynamic programming algorithm:



- Let maxEarned[i], where $1 \le i \le M$, be the maximum earned money generated from beginning to i km on the road.
- For each km in the road, we need to check whether this km (x_i) has option for any advertisement, if not then maximum earned money generated till that km would be same as maximum earned money generated till one km before
- If that km (x_i) has option for advertisement, then we have two options:
  - we can place the ad and add the earned money of ad placed
  - we can ignore the ad.
- So maximum earned money can generated by:

maxEarned[i] = max(maxEarned[i-restriction-1] + r[line], maxEarned[i-1])

Pseudocode :

```
                                x_i    r_i    length(x)
                                                      4 km for this problem
ALGORITHM maxEarnedMoney(M, x, r, n, restriction)
    create maxEarned[] ← [0] * (M+1)
    line ← 0
    for i ← 1 to M+1 do
        if line < n
            if (x[line] != i)
                maxEarned[i] ← maxEarned[i-1]
            else
                if i ≤ restriction
                    maxEarned[i] ← max(maxEarned[i-1], r[line])
                else
                    maxEarned[i] ← max(maxEarned[i-restriction-1] +
                                        r[line], maxEarned[i-1])
                end if
                line ← line + 1
            end if
        else
            maxEarned[i] ← maxEarned[i-1]
        end if
    end for
    return maxEarned[M]
end
```

Providing the recursive formula : Let maxEarned[] => M[]

$$F(i) = \max \begin{cases} M[i-1] & \text{if } x[line] \,!= i \text{ or } i \leq restriction \\ M[i-restriction-1] + r[line] & \text{if } i \geq restriction \end{cases}$$

- The recursive formula comes from the above algorithm, I take the conditions inside the loop and put it into my recursive formula.

Computational complexity :

→ Time complexity : O(n) → where n is length of road (M)

Because inside the algorithm, we have one loop, which traverses M times

→ Space complexity : O(n)

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)**

$$
\begin{array}{c c}
 & \text{persons} \\
\text{jobs} & \begin{bmatrix} - & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & C_{ij} & - & - \\ - & - & - & - \end{bmatrix}
\end{array}
$$

Cost of assigning job i to person J.

→ Our goal is to assign each job to a person such that the maximum cost among the assignments (not total cost) is minimized.

Person	Jobs	costs
P_1	x	100
P_2	y	60
P_3	z	120
P_4	y	50
P_5	z	200

$\xrightarrow[\text{cost}]{\text{Sort}}$

Person	Job	cost
P_4	y	50
P_2	y	60
P_1	x	100
P_3	z	120
P_5	z	200

We should take min cost among the jobs:

3 jobs:

x	y	z
P_1	P_4	P_3

Explanation :

- Let's say we have n person and we have m jobs. Such that $\boxed{m < n}$

And then ;

- Let's sort cost array in increasing order. In that way we can take the jobs with min cost.
- After taking the cost we have to check the job is available or not.
  - If available then assign that job to that person
  - If not we are gonna find other available.
- With this approach, we are gonna assign one person to each job such that the maximum cost among the assignments is minimized.

Best case complexity : $O(n)$ If we find the all jobs available in first iteration.

worst case complexity : $O(n^2)$ If we can not find the all jobs available. Then we should check $n-1$ again.

Average case complexity : same as worst case

Actually I could not implement it.

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)**

- I used Merge Sort which uses divide and conquer algorithm

- The count of inverson with index i, J such that i is in the left half and J is in the right half and at the end add all three to get the total inversion count.

- Using divide and conquer approach, we can perform two passes of the left and right half during merge step.
  → In the first pass, calculate the number of inversions in the merged array
  (Special case left[i] > 2 * right[J])
  → In the second pass, castruct the merged array.

Computational complexity:

→ Time complexity: Since I use Merge Sort, and we know that merge sort time complexity is $O(n\log n)$

Best
worst
Avg

$$\boxed{T(n) = 2T\left(\frac{n}{2}\right) + n} \qquad a = 2 \quad b = 2 \quad d = 1$$

Using Master Theorem $\rightarrow a = b^d \rightarrow O(n^d \log n)$
$= O(n^1 \log n)$
$= O(n \log n)$

→ Space complexity = $O(n)$

Pseudocode :

```
NumberOfInversions (arr)
    if length(arr) ≤ 1
        return arr, 0
    else
        mid ← length(arr)/2                         # Divide array into two parts.
        left, a ← NumberOfInversions (arr[:mid])    # Recursive call for
        right, b ← NumberOfInversions (arr[mid:])     left and right part
        res, c ← UsingMerge (left, right)           # Merge two parts.

        return res, a+b+c
```

```
UsingMerge (left, right)  ───→  O(n)
    result ← [ ]
    count ← 0
    i, J ← 0, 0
    while (i < length(left) AND J < length(right))  # First pass to count
                                                        number of inversions.
        if left[i] > 2 * right[J]                   # special case.
            count += length (left) - i
            J ← J+1
        else
            i ← i+1
        end if
    end while
    i, J = 0, 0
    while ( i < length(left) AND J < length(right))  # Merge two
                                                         sorted arrays.
        if left[i] < right[J]
            result.append (left[i])
            i ← i+1
        else
            result.append (right[J])
            J ← J+1
        end if
    end while
```

```
while (left[i:] or right[j:])
    if left[i:]
        result.append(left[i])
        i ← i+1
    if right[j:]
        result.append(right[j])
        j ← j+1
    end if
end while
return result, count
end
```