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
Türker Tercan 171044032						

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:

Türker Tercan
171044032

```
Pseudocode longest_palindrome(string text)
    length = len(text)
    dp = [[false for i←0 to length] for j←0 to length]
    for i←0 to n
      dp[i][i] = true
    end for
    index = 0
    palindrome_length_2(string, dp, index)  // 2 length palindrome recursive
    k=3
    outer_recursive(k, string, dp)          // Longer palindrome recursive
    start = 0
    max_palindrome = 1
    for i←0 to length                       // Check 2D DP table and find maximum
        count = 0                           // length palindrome
        for j←0 to i+1
            if i+j+1 >= length
               break
            if !dp[i-j][i+j+1]
               break
            count += 2
        end for
        if count > max_palindrome
           max_palindrome = count
           start = i - (max_palindrome / 2) + 1
        end if
    end for
    for i←0 to length
        count = 0
        for j←0 to i+1
           if i+j >= length
              break
           end if
           if !dp[i-j][i+j]
              break
           end if
           count += 2
        end for
        if count > max_palindrome
           max_palindrome = count - 1
           start = i - (max_palindrome / 2)
        end if
    end for
    print(text[start : (start + max_palindrome)])
end
```

Tuğçe Tercan
171044032

```
pseudocode palindrome_length_2 (string, dp, index)
    if index < len(string)
        if string[index] == string[index + 1]
            dp[index][index + 1] = true
        end if
        palindrome_length_2 (string, dp, index + 1)
    end if
end

pseudocode outer_recursive (k, string, dp)
    if k == len(string) + 2
        return
    end if
    index = 0
    inner_recursive (index, k, string, dp)
    outer_recursive (k+1, string, dp)
end

pseudocode inner_recursive(index, k, string, dp)
    if index < len(string) - k + 1
        j = index + k - 1
        if dp[index + 1][j - 1] && string[index] == string[j]
            dp[index][j] = true
        end if
        inner_recursive(index + 1, k, string, dp)
    end if
end
```

Algorithm Explanation:

assume text = "dbabc"

1- Fill dp table with size [n][n]

$$dp(i,j) = \begin{cases} \text{true, if substring } S_i \dots S_j \text{ is palindrome} \\ \text{false, otherwise} \end{cases}$$

$dp(i,i) = true$ $\quad dp(i, i+1) = (S_i = S_{i+1})$ $\quad dp(i,j) = (dp(i+1, j-1) \text{ and } S_i == S_j)$

2- Fill the table with base cases given above

3 - Search the table and print the maximum length palindrome

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

Complexity Analysis:

palindrome_length_2 runs $O(n)$ times

outer_recursive ($O(n^2)$ times its iterates recursively dynamic map.

So total time complexity: $O(n^2)$

Memory space: $O(n^2)$

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)**

Q2:

Türker Tercan
171044032

```
Procedure QuickSort(L[low:high])
    if high > low
        call Rearrange(L[low:high], position)
        call QuickSort(L[low:position-1])
        call QuickSort(L[position+1:high])
    end if
end

procedure Rearrange(L[low:high], position)
    right = low
    left = high+1
    x = L[low]
    while right < left do
        repeat right ++ until L[right] >= x
        repeat left ++ until L[left] <= x
        if right < left
            interchange(L[left], L[right])
        end if
    end while
    position = left
    L[low] = L[position]
    L[position] = x
end

procedure FindIntervals(array[0:n], interval[0:m])
    QuickSort(array[0:n])
    for i in interval
        print(interval[i[0]])
```

Time Complexity:

FindIntervals function sorts the array with $O(n \log n)$ time

Iterates the n intervals that has time $O(n)$ time

Overall time complexity:
$O(n \log n) + O(n) = O(n \log n)$

As we know, we have list of n numbers and have a interval list with m intervals. ($m < n$). Each interval is a list like $[a, b]$ and every a, b $1 \le a \le b \le n$ must meet the condition.

First sort the array with QuickSort algorithm and traverse the m intervals. We want to find minimum value of each interval. $a_i \le b_i$ must meet this condition. Hence, a_i will always be the minimum value of the interval because our list is sorted.

Let us assume array = [15, 12, 13, 14, 6, 10, -2, 2]
and interval = [[1, 8], [2, 7], [3, 6], [4, 5], [8, 8], [2, 3], [5, 6]]

1- After QuickSort array → [-2, 2, 6, 10, 12, 13, 14, 15]
2- Print every interval[i[0]] (0 index means its a)
→ -2 2 6 10 15 2 12

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)**

Q3:

Türker Tercan
1710444032

```
pseudocode max_earning(miles, places, earnings, length, distance)
    dp_earning = [0] * (miles + 1)
    next_billboard_index = 0
    recursive_earning(1, miles, places, earnings, length, distance, dp_earning, next_billboard_index)
    return dp_earning[miles]
end

pseudocode recursive_earning(i, miles, places, earnings, length, distance, dp_earning, next_billboard_index)
    if i < 1 or i > miles
        return
    end if
    if next_billboard_index < length
        if places[next_billboard_index] != i
            dp_earning[i] = dp_earning[i-1]
        end if
        else
            if i <= distance
                dp_earning[i] = max(dp_earning[i-1], earnings[next_billboard_index])
            else
                dp_earning[i] = max(dp_earning[i-distance-1] + earnings[next_billboard_index],
                                    dp_earning[i-1])
            end if
            next_billboard_index ++
        end else
    end if
    else
        dp_earning[i] = dp_earning[i-1]
    end else
    recursive_earning(i+1, miles, places, earnings, length, distance, dp_earning, next_billboard_index)
end
```

dp_earning has M+1 size and dp_earning[i] represents the maximum earnings that can be earned up to i miles

$$dp\_earning[0..i..M] = \begin{cases} dp\_earning[i] = dp\_earning[i-1] & \text{// if there is no places to select} \\ dp\_earning[i] = \max(dp\_earning[i-1], earnings[next\_billboard\_index]) & \text{(if } i <= distance\text{) // We can earn from only places up to current mile is greater than distance} \\ dp\_earning[i] = \max(dp\_earning[i-distance-1] + earnings[next\_billboard\_index], dp\_earning[i-1]) & \text{// select max of removed previously place's earnings or one mile before 's earnings} \end{cases}$$

Türker Tercan
171044032

Example:

M = 12

x = [2, 4, 5, 7, 10, 11]

r = [10, 7, 12, 3, 5, 8]

distance = 4

1- Initialize dp

[0,0,0,0,0,0,0,0,0,0,0,0,0]

↳ start from index -1

2- i=2 x[0] = 2, r[0] = 10

[0,0,10,0,0,0,0,0,0,0,0,0,0]

3- i=4 x[1] = 4 r[1] = 7

[0,0,10,10,10,0,0,0,0,0,0,0,0]

dp[3] > r[1] then select dp[3]

4- i=5, x[2] = 5 r[2] = 12

[0,0,10,10,10,12 ----

dp[i-distance-1] dp[i-1]

select max(r[2] + dp[i-distance-1], dp[i-1])

5- i=7 x[3] = 7 r[3] = 3

[0,0,10,10,10,12,13,---

dp[i-distance-1] dp[i-1]

select max(r[3] + dp[i-distance-1], dp[i-1])

6- i=10 x[4] = 10 r[4] = 5

[0,0,10,10,10,12,13,13,13,17---

select max(r[3] + dp[i-distance-1], dp[i-1])

7- i=11 x[5] = 11 r[5] = 8

[0,0,10,10,10,12,13,13,13,17,20

select max([r[3] + dp[i-distance-1], dp[i-1])

8- i reaches M

[0,0,10,10,10,12,13,13,13,17,20,20,20,20,20]

dp[M] = 20 is the result

Time Complexity:

O(M), M is the total miles because we have to iterate each mile and we need O(M) memory space

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)**

Q4:

Türker Tercan
171044032

	J1	J2	J3	J4	J5
P1	550	210	1250	1235	999
P2	186	842	982	396	453
P3	666	245	852	367	1750
P4	110	962	456	385	750
P5	975	265	310	1125	870

1- Give each job-person to random non negative

2- Our goal is minimize the maximum cost among the assignments (not the total cost)

3- We need to minimize these values in order to do that lets select maximum assignment

P3 666 245 852 367 1750

J5: 999, 453, 1750, 750, 870

How to minimize these value
- Find the minimum value within its row or column
- In our case (245)
- After that assign J2 to P3 so 1750 can not be used

4- After that find maximum value (1250)

P1 550 ~~210~~ J3: 1250, 982, ~~852~~, 456, 310 1235 999

Find min value = 310
Assign J3 → P5

5- Find min value that blocks that maximum value (999)
Assign J4 → P5

6- Last value → 550
Assign P1 → J1

Final Result!

P3 → J2 (245)
P5 → J3 (310)
P4 → J4 (385)
P2 → J5 (453)
P1 → J1 (550)

Turker Tercan
1710440 32

```
Procedure minimize_maximum_assignments(assignment[0:n][0:m])
  // n is the person count, m is the job count and
  // always must be m>=n
  rows = [true] * n
  columns = [true] * m
  total = 0
  links = []
  while rows != [false * n] and columns != [false] * m do
      i=0
      j=0
      max=0
      for i=0 to n do                          // In order to find max
        for j=0 to m do                        // First, we need to find usuable
            if rows[i] and columns[j]          // first assignment
              max = assignment[i][j]
              break
            end if
        end for
      end for
      x=i
      y=j
      for i=0 to n do                          // Find the maximum cost
         for j=0 to m do                       // among the assignments
            if rows[i] and columns[j] and max < assignment[i][j]
              max = assignment[i][j]
              x=i
              y=j
            end if
         end for
      end for
      min=max
      min_x=x
      min_y=y
      for j=0 to m do                          // Check the current max cost's row
         if rows[x] and columns[j] and min > assignment[x][j]
            min = assignment[x][j]
            min_x=x
            min_y=y
         end if
      end for
      for i=0 to n do                          // Check the current max cost's column
        if rows[i] and columns[y] and min > assignment[i][y]
            min = assignment[i][y]
            min_x = i
            min_y = j
        end if
      end for
      total += assignment[min_x][min_y]
      links.append([min_x, min_y, assignment[min_x][min_y])
      rows[min_x] = false
      columns[min_y] = false
  end while
end
```

Türker Tercan
171044032

Time Complexity Analysis :

- While outer loop → Job size times $O(m)$
- 1st loop // to find start location → $B(n) = O(1)$, $W(n) = O(n.m)$, $A(n) = O(n.m)$
- 2nd loop // to find maximum cost assignment → $B(n) = O(1)$, $W(n) = O(n.m)$, $A(n) = O(n.m)$
- 3rd and 4th // to find maximum cost // assignment's neighbor // minimum assignment location → $B(n) = O(1)$, $W(n) = O(n+m)$, $A(n) = O(n+m)$

So all function takes $O(m.m.n)$ time and $\in O(n^3)$

Our function does not rely on given array. It always traverses the array. So best and average case is $O(n^3)$

If all the elements are the some worst case occurs but our expected running time will be again $O(n^3)$

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)**

Q5:

Türker Tercan
171044032

```
procedure count_of_inversions(array, length)
    temp = [0] * length
    return merge_sort(array, temp, 0, length - 1)
end

procedure merge_sort(array, temp, left, right)
    count = 0
    if right > left
        mid = (right + left) / 2
        count = merge_sort(array, temp, left, mid)
        count += merge_sort(array, temp, mid + 1, right)
        count += merge(array, temp, left, mid + 1, right)
    end if
    return count
end
procedure merge(array, temp, left, mid, right)
    i = left
    j = mid
    k = left
    count = 0
    while i <= mid - 1 and j <= right
        if array[i] <= array[j] * 2
            temp[k] = array[i]
            k += 1
            i += 1
        end if
        else
            count += mid - i
            temp[k] = array[j]
            k += 1
            j += 1
        end else
    end while
    while i <= mid - 1
        temp[k++] = array[i++]
    end while
    while j <= right
        if array[i-1] > array[j] * 2
            count -= 1
        end if
        temp[k++] = array[j++]
    end while
    for i = left to right++1
        array[i] = temp[i]
    end for
    return count
end
```

- The merge functions split arrays into two parts recursively.

array[0:mid] array[mid+1 : n]

- Calculate inversions that a[i] > a[j] * 2 when sorting.
- There are mid-i inversions so sum up with recursively and print the result

Time Complexity: Merge Sort is a divide and conquer algorithm. There is no extra traverses

So overall complexity is

O(nlogn)