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)**

Gebze Technical University
CSE 321 Introduction to Algorithm Design
Final Exam

Yusuf Can Kan
1604207

1) Inorder to find a subarray in dynamic programming approach, I follow a bottom-up approach. Lets say our input array has n. complexity. We construct an array nxn. The row (i) of this array represents our letter from left, the column (j) of this array represents our latter from right. If our arr[i] == arr[j], we will mark the arrays arr[i][j] element. Since we also need the length of the array, we will also store the current max length. We will move subarray length to the bottom in the array. At the end of the program our start index and length stored in the result array.

for example;

input; [t,e,n,e,t]

first the algorithm fills the length of 1 and length of 2.

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

⇒ After this point every time its filling arr[i][j], it will check the subsequence. So, for arr[i][j], it will look arr[i+1][j-1].

↳ one index forward from right, 1 index backward from left.
for length 2 it will look length 1.
for example, for [t e n e t] it will check [e n e] is true or not.

So if we apply this procedures to all array, the result array will be;

1	0	0	0	1
0	1	0	1	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

→ The 1 in the tables shows that, arr[i][j] is meets with our condition.
In this case arr[0:4] is same from back and from beginning. So arr[0:4] meets our condition.

Since for every substring pair we are looking for is sub array in the table, our
([i][j]) ([i+1][j-1])

recursive formula is;

- subArrTable[i][j] = 1 — if — subArrTable[i+1][j-1] == 1 and arr[i] == arr[j]
- subArrTable[i][j] = 1 — if — i == j
- subArrTable[i][j] = 1 — if — (i - j) == 2 and arr[i] == arr[j]
- subArrTable[i][j] = 0 — if — Otherwise

The pseudocode of the algorithm;

```
procedure findSubArr(arr):
    size = len(arr)
    for i ← size:                          → This loop fills the table for
        fill subArrTable[i][i] with 1          length 1 and length 2.
        fill subArrTable[i][j] with 1 if arr[i] and arr[i+1] are the same and
          set the max index.
    endfor
    for i ← (2, size):                     → This loop fills the rest of
        left = 0                               merge array.
        right = left + 1
        while right is not in the end of the arr;
            if arr[right] is equal arr[left] and subarray is true:   → subArrTable[left+1][right-1]
                fill subArrayTable[left+1][right-1] with 1
                update max length and result index.
            endif
        endwhile
    endfor
end procedure
```

As we can see from pseudocode, the subArrTable [i] [j] represents the arr [i:j] is same from start and from end or not.

The complexity of this algorithm is simple;

Filling length 1 and 2 in the table iterates string 2 time
↳ O(n)

Filling rest of the table needs 2 loop. One loop iterates the gap between left index and right, otherone moves the left and right index.
↳ O(n*n)

So total complexity is; $O(n) + O(n^2) = O(n^2)$

Note

If we didn't use dynamic programming our complexity would be $O(n^3)$ because we had to iterate 1 more iteration for the substrings.

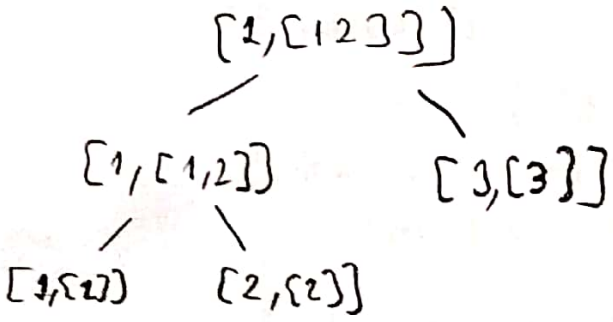
2) In this question, I implement the tree for all possible subsets and its result. The roots element of my pose tree contains all the input list and it goes to the bottom with dividing list 2 in every time. All the leaf nodes of the tree has individual element indexes of the input list.

While constructing the tree, I assign minimum values of sublist from bottom to top. Each node, finds minimum element of its list with comparing its leaf minimum elements. All the leaf elements of tree has its individual list element indexes as a minimum elements.

Node structure of leaf nodes => [minimum value in the interval ; [interval indexes]]

If we had the list of [1,2,3]

The implemented tree will be like this;



Since minimum values of lists assigning for bottom to top it does not make any additional calculation.

Note: Tree implemented as a 1D array in the code

pseudocode of construct tree algorithm;

```
procedure constructTree (Tree, inputList, currentNode, start, end):
    if interval includes 1 value:
        construct leaf node and return.
    else:
        middle = (start + end) / 2
        Construct left part of the tree
        construct right part of the tree
        Set current node min element as min(leftMin, rightMin)
end procedure
```

As we can see this construct operation contains 2 rewsive call. Each recursive call handles the hoff of the input list. At the end it constructs the current node.

$T(n) = 2T(n/2) + 1$ → constructing current node

from master theorem;

$2 > 2^0 \Rightarrow \Theta(n)$ complexity

In finding the minimum interval part, all we need to do is searching the tree that we had constructed and returning the value.

pseudocode;

```
procedure find_min (tree, node, interval)
    if interval is not match current node:
        return infinity
    if interval matches:
        return tree node.

    r1 = find min of left tree
    r2 = find min of right tree

    return min(r1, r2)
```

In this procedure as we can see again we have 2 rewsive calls that divides tree to 2.

In this case, each iteration divides array into 2;

First iteration $\Rightarrow n/2$

Second iteration $\Rightarrow n/2^2$

:

k'th iteration $\Rightarrow n/2^k$

→ After k division, the length of the array becomes 1;

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = \log_2 (2)^k$

→ $\log n = k$

Therfore complexity of search is $O(\log n)$

In final, when we look to the proved algorithm;

pseudocode;

procedure find (list, intervals):

Construct tree ⟶ O(n) time complexity

for → list ⟶ O(n) time complexity

find_min(interval) ⟶ O(lgn) time complexity

} O(nlgn)

Total time complexity is $O(n)+O(n\lg n) = O(n\lg n)$

3) In this question I implemented a program that iterates all km's of the road one by one and makes calculations with respect to advertisements location.

If I didn't implement the algorithm in dynamic programming way, the normal algorithm will calculate all the sub calculations and it will take to much time complexity.

In order to ignore to calculate all subsets more than one, I tried to store maximum income in each iteration in the problem.

```
procedure findMax(locations, incomes, M):
    incomeMemory[M] = 0 # initialize income memory
                        # it stores total income values of every km.

    for i ∈ M : # iterate all the road
        if there is no advertisement location left:
            return final incomeMemory element
        else:
            if i is adv. location:
                if previous location is in 4km:
                    compare the incomes and update the
                    the incomeMemory array. (update the max income)
                else:
                    directly add current income
                    to income array. (directly add current income to total income)
            else:
                update incomeMemory[i] with incomeMemory[i-1]
    endfor
    return current total income
end procedure
```

↳ last element of incomeMemory

In the algorithm first it takes the first adv. location and adds its income to the total income. Its updates all the indexes of incomeMemory with this max income, until it finds the another adv. location. When it finds new location, it checks if that location is in 4km. distance with old location. If it is, it finds that if it has bigger income than the older one. If new location is not in 4km. close to the old location, program directly includes this advertisement to the total income.

In this algorithm we can clearly see that program fulfills the dynamic programming conditions with looking incomeMemory.

When we iterate all the elements of income memory, we use the previous indexes maximum income value. So in this way we make the maximum value with each iteration one forward.

While doing this either we place the advertisement from the last one (if last one is in 4km and smaller than current one) or does not place the current one.

Complexity:

The findmin function iterates the road from start to end. So it makes M iteration. Inside of this statement, all the comparisons and assignments takes O(1).

So total complexity is $\underline{\underline{O(M)}}$ which M is the length of the road.

Recursive formula:

$$incomeMemory[i,\dots,n] \begin{cases} incomeMemory[i] = \max(incomeMemory[i-5]+income[i], incomeMemory[i+1]) & \text{if } i \neq j \\ incomeMemory[n] & \text{if } i == j \end{cases}$$

4) procedure assignment(arr):

```
    for i from 0 to sizeof(arr):
        i, j = find_max(arr) # finds the max element of array. Assigns indexes
                               of max element o i and j
        i, j = find_min(arr, i, j) # finds the minimum element in the given
                                     column and row and set i and j.
        add (i, j) to result.
        set -1 to all ith row and j'th column.
    endfor
    return result
end
```

Explanation of algorithm: function first finds the maximum element in all array. It looks the row and column of max element and finds the minimum element. Adds minimum element to the result (this element represents index of rowth person, index of column jobs cost). After that, since we assign the job and person for that index, program sets row of min and colum of min to -1. Repeats the same process until all the jobs are assigned.

For example;

persons \ jobs			
	3	5	2
	8	1	6
	(9)	7	5

(9) ← max element

=>

	(3) ← min element	5	2
	8	1	6
	(9)	7	5

=>

	-1	-1	-1
Result	-1	1	6
3	-1	(7) ← max elem.	5

=>

	-1	-1	-1
	-1	(1)	6
	-1	(7) ← min elem.	5

=>

	-1	-1	-1
	-1	-1	-1
	-1	-1	(5)

=> result 3,1

=> Result 3,1,5

person 0 assigned to job 0. cost = 3
person 1 assigned to job 1 cost = 1
person 2 assigned to job 2 cost = 5

Complexity analysis;

- The for loop iterates n times. => O(n)
- The find_max function iterates every element one by one. It looks n column and n row, and every time it runs, it iterates all the array. → $O(n^2)$
- The findmin function iterates only given row and given column. Every time it runs it iterates 1 row and 1 column. So; $O(n+n) = O(2n) = O(n)$
- Appending result to array has constant time complexity => O(1)
- Setting -1 to founded column and row every time, iterates 1 column and 1 row. => $O(n+n) = O(n)$

So;

O(n) => for loop

$O(n^2)$
$O(n)$
$O(1)$
$O(n)$

$\Big\rangle O(n^2) \Big\rangle \underline{O(n^3)}$

In best-worst and average case all the complexities are the same.

The reason for that is, different content of the array does not decreases the time complexity of any subfunction.

find_max is always has $O(n^2)$ complexity
find_min is always has O(n) complexity.
Setting -1 to row and column always has O(n) complexity.

All the complexities are $\underline{O(n^3)}$

5) In this question I implemented merge sort algorithm that also solves our problem.

In our problem we need to find a new defined inversion. In my merge sort algorithm I find left parts inversion count, right part inversion count and merge part inversion count and sum them all.

$$\text{Inversion} = \underset{\text{Inversion}}{\text{left array}} + \underset{\text{Inversion}}{\text{right array}} + \underset{\text{Inversion.}}{\text{Merge}}$$

The main calculation done in the merge part. Since we have 2 sorted (increasing order) left and right array, if one of the left element will be bigger than the other one, we can check our $2xi > 2xj$ condition. If condition is true we can add left array count to result because since both of the arrays sorted, all the left elements supplys our condition. If our invertion would be $ki > kj$ above algorithm would work. But if we look above example;

[12, 11, 13, 5, 6, 7]

↳ In this array it sorts subarrays until this part;

[11, 12, 13] [5, 6, 7]

When we look this condition, first algorithm will look 12 and 5, $11 > 5$ and $11 > (5.2)$ so it will add invertion count to length of left array. It places 5 to the merge array, and compares 11 with 6. 6 does not meet our requirements and 7 is also the same. So it does not add anything to invertion count for both this cases and it places 6-7-12-11-13 to the merge array. As we can see algorithm didn't calculate the 13-6 condition.

In order to prevent this I dabled the right array and calculate the invertion count as invertion function is $x_i > x_j$.

[11,12,13] [10,12,14]

- 11 > 10 → condition ok. invertion count +3
- place 10. [11,12,13] [12,14]
- 11 < 12 → condition is not ok.
- place 11 [12,13] [12,14]
- 12 = 12 → condition is not ok
- place left 12 [13] [12,14]
- 13 > 12 → condition ok, invertion count +1
- place 12 [13] [14]
- 13 < 14 → condition not ok.
- place 13 [14]
- place 14 [13]

AS we can see we calculate the nessesary part.

But the problem is if we double the right part of the array, the upper recursive calls will affect from this. So for solving this problem in order to update merged array with dubled right part, first I normally meged 2 array, after that I take a copy of the right array and doubled it and I only check the invertion condition with one merge iteration. Since I already construct merged array, everything is stay the same and it calculated the invertion count.

pseudocode is in next page

Pseudocode;

```
procedure countInverseWithMergeSort(input):
    inversion count = 0
    if (len(input) ≤ 1):
        return 0
    endif
    inversionCount += countInverseWithMergeSort(left half of input)      } Recursive
    inversionCount += countInverseWithMergeSort(right half of input)     } calls of
                                                                           Merge sort
    Merge the left part and right part

    Copy the right part.
    double values of right part.
    while (end of left or end of right part):
        if (left[i] ≤ doubledRight[j]):
            i = i+1
        else:
            inversion Count += (len(left[i:]))
            j = j+1
        x = x+1
    endwhile
    return inversion Count
end procedure
```

This loop is same with merge operation loop. Instead it does not constructs a merged array, it just control our inversion conditions.

Complexity analysis;

Merge sort divides problem into 2 subproblem $T(n) = 2T(n/2)$

Merging operation takes n iteration.

Controlling our inversion case takes n iterations

$T(n) = 2T(n/2) + 2n$ → so $a=2$ $b=2$ $d=1$

From master theorem;

if $a = b^d$ then $\Theta(n^k \log n)$ → $2 = 2^1$ therefore

$\Theta(n \log n)$