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)**

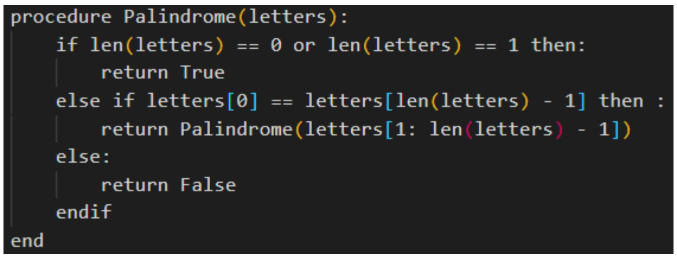
Answer :

In this problem, we need to verify each and every substring possible.This means dynamic programming bottom-up approach would work best.

Suppose we have an S string. If the length of this string is;
0 = An empty string of palindrome.
1 = All substrings of length 1 are a palindrome.Because a letter by itselfs is a palindrome.
2 = S is palindrome if the first and last letters are the same.
3 = S is palindrome if the first and last letters are the same.Middle letter doesn't effect the results.
We can implement this case by use length 1 and 2 rule.
4 and more = If the first and last letters are the same and second last letters are the same
and so on.

After that, we will reach a solution by applying the rules we have determined before.

When we write a recursive function using these rules, pseudocode is like this;



```
procedure Palindrome(letters):
    if len(letters) == 0 or len(letters) == 1 then:
        return True
    else if letters[0] == letters[len(letters) - 1] then :
        return Palindrome(letters[1: len(letters) - 1])
    else:
        return False
    endif
end
```

First if check letters is empty string or length is 1 then letters is palindrome.

Else if recursively check if first and last letters are same.

else letters are not palindrome.

There are 3 base cases. To solve this question using dynamic programming, we need to place these 3 basic initial values in dp_table.

I created NxN 2d dp_table, where n is letters size. In order to check the palindrome state, we need to keep one end of the letters and the head. I show the beginning with B and the end with E.

In the begining, the diagonal of the dp_table must be true and the bottom of the diagonal must be false.Because the bottom of diagonal consider subletters in reverse order. The reason we make diagonal true is because for length 0 and length 1 are provided at first.

If letters length is 4 then dp table start with;

	E_1	E_2	E_3	E_4
B_1	T			
B_2	F	T		
B_3	F	F	T	
B_4	F	F	F	T

After we start the table, we fill the table with T and F according to the rules we have determined above.For length 2, if letters are same in dp_Table B1,E2 will be T.If B2,E3 T, this means letters[1] = letters[2]. In this way, we fill the whole table. We keep the start and end indexes, find the longest palindrome string and return.

Time complexity of this algorithm $T(n)= O(n^2)$ because we fill the dp_table using 2 nested loops.

Auxiliary Space of this algorithm also $O(n^2)$ because we create dp_table with size n*n.

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)**

Answer : The approach is to precompute minimum of all subarrays of size **2^j** where j varies from 0 to **Log n**.

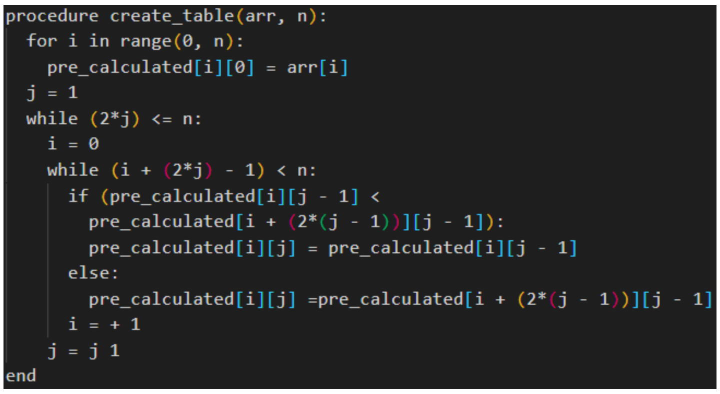
I create 2d pre_calculate table.The table [i][j] contains minimum of range starting from i and of size 2^j.

So we have to fill the table.

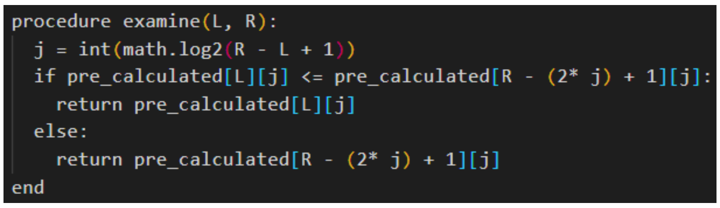
We fill the table in a bottom-up manner using previously computed values. We compute ranges with current power of 2 using values of lower power of two.

For any random spacing [left, Right] we need to use ranges that are multiples of 2. We do this using multiples of two. We always need to make at most one comparison, compare at least two intervals is the power of 2. For example, to find a minimum of range [0, 7] (Range size is a power of 3), we can use the minimum of following two ; Minimum of range [0, 3] (Range size is a power of 2) and [4, 7] (Range size is a power of 2).By doing this, we divided by 2.

For any random spacing [left, Right] we need to use ranges that are multiples of 2. We do this using multiples of two. We always need to make at most one comparison, compare at least two intervals is the power of 2.



```
procedure create_table(arr, n):
  for i in range(0, n):
    pre_calculated[i][0] = arr[i]
  j = 1
  while (2*j) <= n:
    i = 0
    while (i + (2*j) - 1) < n:
      if (pre_calculated[i][j - 1] <
        pre_calculated[i + (2*(j - 1))][j - 1]):
        pre_calculated[i][j] = pre_calculated[i][j - 1]
      else:
        pre_calculated[i][j] =pre_calculated[i + (2*(j - 1))][j - 1]
      i = + 1
    j = j 1
end
```



```
procedure examine(L, R):
  j = int(math.log2(R - L + 1))
  if pre_calculated[L][j] <= pre_calculated[R - (2* j) + 1][j]:
    return pre_calculated[L][j]
  else:
    return pre_calculated[R - (2* j) + 1][j]
end
```

Since we are dealing with the tops of 2, time complexity of algorithm **T(n) = O(nlogn)**

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)**

Answer : All possible advertisements places are in range from 0 to M.We can say;

M(i)=maximum revenue upto first i advertisements so;

We created 1d dp array called maxRev with size M+1 (0 to m). The for each mile on the highway, we need to check whether this mile has the option for any advertisement place;

Road [0....x...M] ;

If x is not the advertisement place then the maximum revenue generated till that mile would be same as maximum revenue generated till one mile before.

If x is advertisement place ; We can ignore this place **or** we will place the advertisement and ignore the advertisement in previous min_distance miles and add the revenue of the advertisement placed.

So we have to calculate maxRev[i] = max(maxRev[i- min_distance -1] + revenue[i], maxRev[i-1])

This steps repeats recursively.

```
M(i) {
        if we don't have advertisement place that mile then ; copy the previous maximum revenue
        if we have advertisement place that mile and current location is less than or equal to min_distance then calculate
                        maxRev[i] = max (maxRev[i-1], revenue[used_ads count]
        if we have advertisement place that mile and current location is bigger than min_distance then calculate
                maxRev[i] = max (maxRex[i – min_distance – 1 ]+revenue[used_ads_count], maxRev[i-1]
        Increase i
}
```

Pseudocode for algorithm ;

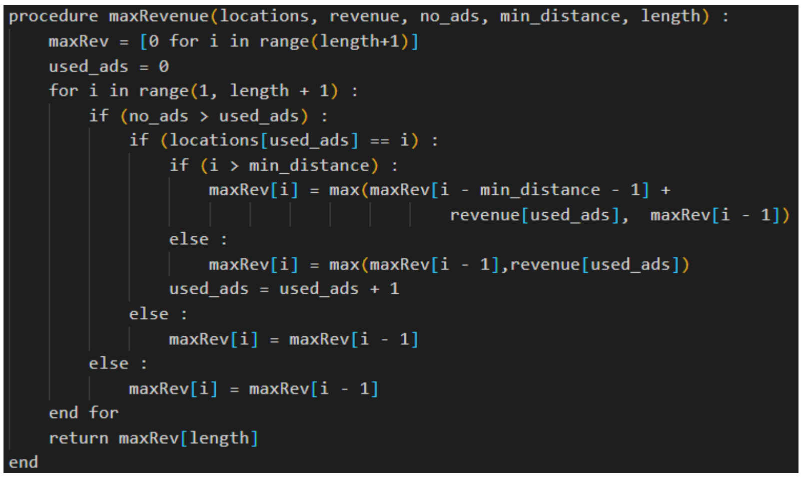
no_ads = number of advertisiments

locations = the locations of the ads that can be placed

revenue = the income of the places in the location

min distance = at least how many miles will be between ads

length = the length of the road



```
procedure maxRevenue(locations, revenue, no_ads, min_distance, length) :
    maxRev = [0 for i in range(length+1)]
    used_ads = 0
    for i in range(1, length + 1) :
        if (no_ads > used_ads) :
            if (locations[used_ads] == i) :
                if (i > min_distance) :
                    maxRev[i] = max(maxRev[i - min_distance - 1] +
                                            revenue[used_ads],  maxRev[i - 1])
                else :
                    maxRev[i] = max(maxRev[i - 1],revenue[used_ads])
                used_ads = used_ads + 1
            else :
                maxRev[i] = maxRev[i - 1]
        else :
            maxRev[i] = maxRev[i - 1]
    end for
    return maxRev[length]
end
```

The time complexity of algorithm is **T(n)= O(n)** because we iterate array only once.But algorithm also has extra O(n) 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)**

Answer :

Since we are trying to minimize the total cost in this part;

My solution approach for this problem;

There are n persons and jobs, I keep it in a 2d array.

Step 1 = Find the largest cost in array and get location.

Step 2 = If there is another cost in the row and column of the cost we find, we eliminate the cost we find.

Step 3 = If there is 1 cost left in the row or column of the cost we have eliminated, we eliminate the neighbors of that cost. Because there can be no other cost in the column and row with that cost.

Step 4 = Repeat from step 1 for the next largest suitable cost.

Continuing this way, I find my minimized costs for each person.

For instance ; There is 4 person and 4 job then initial table is;

	J1	J2	J3	J4
P1	40	27	75	50
P2	62	20	8	45
P3	37	98	60	94
P4	94	33	6	64

Then Start calculating

Largest cost is 98 then eliminate

40	27	75	50
62	20	8	45
37	**98**	60	94
94	33	6	64

Largest cost is 94 then eliminate

40	27	75	50
62	20	8	45
37	98	60	94
94	33	6	64

Largest cost is respectively 94,75,64,60 then eliminate respectively

40	27	**75**	50
62	20	8	45
37	98	**60**	**94**
94	33	6	**64**

Since there is only one cost left in row 3, we eliminate the other elements in the column with the cost.Eliminate 62 and 40.

40	27	75	50
62	20	8	45
37	98	60	94
94	33	6	64

After largest cost is then 50 eliminate.

40	27	75	**50**
62	20	8	45
37	98	60	94
94	33	6	64

Since there is only one cost left in column 4, we eliminate the other elements in the row with the cost.Eliminate 20 and 8.

40	27	75	50
62	**20**	**8**	45
37	98	60	94
94	33	6	64

Next largest suitable cost is 33 then eliminate.

40	27	75	50
62	20	8	45
37	98	60	94
94	**33**	6	64

The remaining 4 numbers are our result.

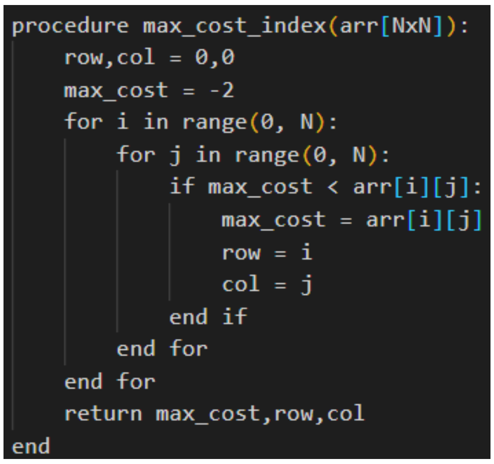
	27		
			45
37			
		6	

P1 = J2 ;; P2 = J4

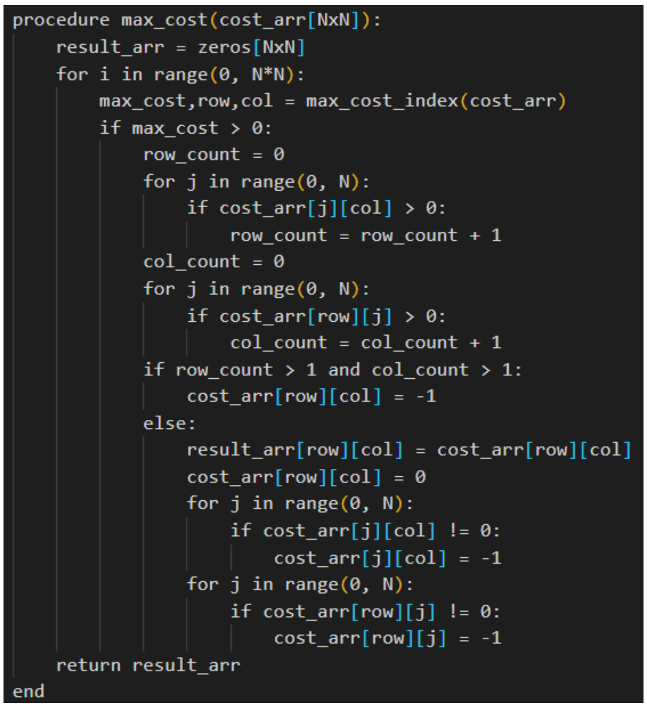
P3 = J1 ;; P4 = J3

	J1	J2	J3	J4
P1		27		
P2				45
P3	37			
P4			6	

Pseudocode for this algorithm;



```
procedure max_cost_index(arr[NxN]):
    row,col = 0,0
    max_cost = -2
    for i in range(0, N):
        for j in range(0, N):
            if max_cost < arr[i][j]:
                max_cost = arr[i][j]
                row = i
                col = j
            end if
        end for
    end for
    return max_cost,row,col
end
```



```
procedure max_cost(cost_arr[NxN]):
    result_arr = zeros[NxN]
    for i in range(0, N*N):
        max_cost,row,col = max_cost_index(cost_arr)
        if max_cost > 0:
            row_count = 0
            for j in range(0, N):
                if cost_arr[j][col] > 0:
                    row_count = row_count + 1
            col_count = 0
            for j in range(0, N):
                if cost_arr[row][j] > 0:
                    col_count = col_count + 1
            if row_count > 1 and col_count > 1:
                cost_arr[row][col] = -1
            else:
                result_arr[row][col] = cost_arr[row][col]
                cost_arr[row][col] = 0
                for j in range(0, N):
                    if cost_arr[j][col] != 0:
                        cost_arr[j][col] = -1
                for j in range(0, N):
                    if cost_arr[row][j] != 0:
                        cost_arr[row][j] = -1
    return result_arr
end
```

It takes O(n) amount of time each time to find the largest cost. The cost of iterating in the 2d array is O(n).Since the algorithm will do the same steps all situations, best case and worst case is same.

Worst case, Best case: O (n*n) = **$O(n^2)$**

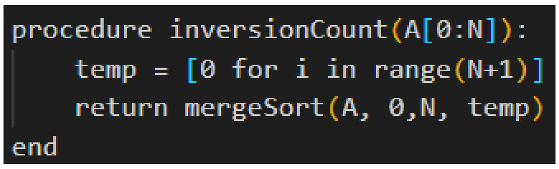
Time complexity of algorithm = **$T(n) = \theta(n^2)$,** where n is number of costs.

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)**

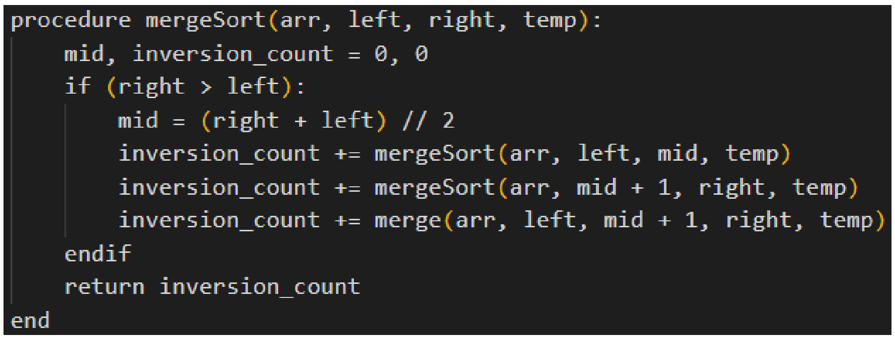
Answer :

I use merge sort to solve this algorithm by using divide and conquer approach.

First i create temp array to use after. I'm sending the input array and the temp array to my merge sort algorithm. First we recursively split the input list in half into 2 part. Once we have recursed $\log_2 n$ times and get to single element lists and hit our base case. Then the algorithm tries to return in the recursion tree. When returning, it returns by ordering the elements it has at each tree level. While doing the merging process, I calculate an inversion count at each step.I calculate the number of x[i] > 2 * x[j] inversion count in the merged array. For any index i in left array **if** x[i] > 2 * x[j], then increment j, all the elements to the left of ith index in left array will also contribute to x[i] > 2 * x[j] inversion count.**Else** increment i. I continue the merge process for the remaining elements. When all the elements are sorted and all inversions are found, the algorithm is finished.



```
procedure inversionCount(A[0:N]):
    temp = [0 for i in range(N+1)]
    return mergeSort(A, 0,N, temp)
end
```

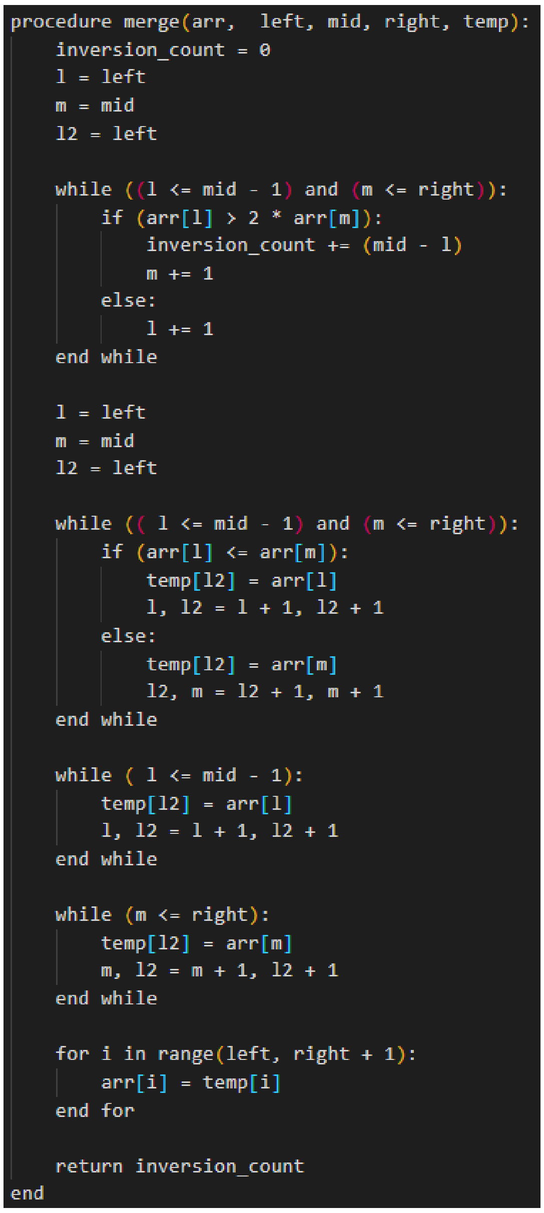


```
procedure mergeSort(arr, left, right, temp):
    mid, inversion_count = 0, 0
    if (right > left):
        mid = (right + left) // 2
        inversion_count += mergeSort(arr, left, mid, temp)
        inversion_count += mergeSort(arr, mid + 1, right, temp)
        inversion_count += merge(arr, left, mid + 1, right, temp)
    endif
    return inversion_count
end
```

Continue on the following page...

First while calculate inversion count in that step.Other whiles for merging process.

Return inversion count.



```
procedure merge(arr,  left, mid, right, temp):
    inversion_count = 0
    l = left
    m = mid
    l2 = left

    while ((l <= mid - 1) and (m <= right)):
        if (arr[l] > 2 * arr[m]):
            inversion_count += (mid - l)
            m += 1
        else:
            l += 1
    end while

    l = left
    m = mid
    l2 = left

    while (( l <= mid - 1) and (m <= right)):
        if (arr[l] <= arr[m]):
            temp[l2] = arr[l]
            l, l2 = l + 1, l2 + 1
        else:
            temp[l2] = arr[m]
            l2, m = l2 + 1, m + 1
    end while

    while ( l <= mid - 1):
        temp[l2] = arr[l]
        l, l2 = l + 1, l2 + 1
    end while

    while (m <= right):
        temp[l2] = arr[m]
        m, l2 = m + 1, l2 + 1
    end while

    for i in range(left, right + 1):
        arr[i] = temp[i]
    end for

    return inversion_count
end
```

Time complexity analysis ;

We already knew that the time complexity of MergeSort is O(nlogn).

The algorithm we wrote is the same as before with merge sort. Merge Sort was already counting inversions for us.We just needed to track it. Merge Sort with inversion counting, just like regular Merge Sort, is

O(n log(n)) time.