Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 4

Ahmet DENİZLİ
161044020

Course Assistant:

1. Part

1.1 A) Write an iterative function which performs this task. Analyze its complexity.

```
LinkedList<Integer> countIncreasingElements(Node head)
{
    // Traverse the list and keep track of max increasing
    // and current increasing lengths
    int curr_len = 1, max_len = 1;
    int total_count = 1, res_index = 0;
    Node curr=head;
    while ( curr.next!=null)                        → O(n)
    {
        // Compare curr.data with curr.next.data
        if (curr.data < curr.next.data)
            curr_len++;
        else {
            // compare maximum length with curr_len.
            if (max_len < curr_len)
            {
                max_len = curr_len;
                res_index = total_count - curr_len;
            }

            curr_len = 1;
        }
        total_count++;
        curr=curr.next
    }

    // Checks if curr_len greater then max_len
    if (max_len < curr_len) {
        max_len = curr_len;
        res_index = total_count - max_len;
    }

    // Traverse the list again to get longest increasing sublist
    int i = 0;
    LinkedList<Integer> new_list = new LinkedList<Integer>();
    for (Node curr=head; curr!=null; curr=curr.next)      → O(n)
    {                                                      (n=res_index+max_len)
        // go to first index of sublist
        if (i == res_index)
        {
            // loop until max greater then 0.
            while (max_len > 0)
            {
                new_list.add(curr.data)
                curr = curr.next;
                max_len--;
            }
            break;
        }

        i++;
    }
    // Returns longest increasing sublist
    return new_list;
}
```

Complexity equation:

Best Case : O(n) time complexity for this algorithm which there is no sorted sublist in this list.

Worst Case : O(2n) ∈ O(n) time complexity for this algorithm which this list is sorted.

1.2 B) Write a recursive function for the same purpose. Analyze its complexity by using both the Master theorem and induction.

```
int[] find_sublist_rec(Node head, int total_count){
    int curr_len = 1, res_index = 0;
    if(head == null){
        res_index = total_count - curr_len;
        return new int[] {res_index, 1};
    }

    for (Node curr=head; curr.next!=null; curr=curr.next)  → O(n/2)    (Average)
    {
        // Compare curr.data with curr.next.data
        if (curr.data < curr.next.data)
            curr_len++;
        else {
            total_count++;
            int[] news = find_sublist_rec(curr.next, total_count);    → T(n/2)
            // compare maximum length with len.                       (Average)
            if (news[1] > curr_len)
                return news;
            else{
                res_index = total_count - curr_len;
                return new int[] {res_index, curr_len};
            }
        }
        total_count++;
    }
    res_index = total_count - curr_len;
    return new int[] {res_index, curr_len};

}
```

```
LinkedList<Integer> getSubList(Node head){
    int[] sublist_info = find_sublist_rec(head, 1);       → T(n)=T(n/2)+ O(n/2)

    int index = sublist_info[0], len = sublist_info[1], i= 0;

    LinkedList<Integer> new_list = new LinkedList<Integer>();

    for (Node curr=head; curr!=null; curr=curr.next)       → O(n)      (Average)
    {
        if (i == index)
        {
            // loop until sublist length smaller then 0.
            while (len > 0)
            {
                new_list.add(curr.data);
                curr = curr.next;
                len--;
            }
            break;
        }

        i++;
    }
    return new_list;
}
```

Complexity of getSubList method : O(n) + T(n)

$T(n) = T(n/2) + O(n/2)$

Using master theorem for complexity equation:

The parameters are:

a = 1, b = 2, f (n) = n ∈ Ω(n^(log2(1)+1))

It is the case 3 of Master theorem. The resulting complexity is thus:

$T(n) = \Theta(n)$

Then Complexity of getSubList method : O(n) + T(n) = O(n) + Θ(n) → O(n)

Using induction:

Suppose, we come (somehow) to a guess: T (n) = O(n)

Using the definition of upper bound "O" we want to prove:

$T(n) \leq cn$ for some suitable $c > 0$

State an induction hypothesis (i.e. let the guess be true for n/2): $T(n/2) \leq c(n/2)$

$T(n) \leq 2(c\,(n/2)) + n/2$

$T(n) \leq cn + n/2$

$= (2c + 1) * n/2$

$T(n) \leq cn$

$n \geq n0=1$ and $c \geq 2$ then $T(n) = O(n)$

Then Complexity of getSubList method : $O(n) + T(n) = O(n) + O(n) \rightarrow O(n)$

2. Part - Describe and analyze a Θ (n) time algorithm that given a sorted array searches two numbers in the array whose sum is exactly x.

```
int l = 0, r = size -1;
while (l < r){
    if (Arr[l] + Arr[r] == sum)
        return true;
    else if( Arr[l] + Arr[r] <  sum )
        l++;
    else
        r--;
}
return false
```

1) Initialize leftmost and rightmost index variables to find the candidate elements in the sorted array.
2) Loop while l < r.
3) No candidates in whole array - return 0

Best Case : O(1) time complexity for this algorithm which sum of first and last number of array is equal to x which is searched.

Worst Case : O(n) time complexity for this algorithm which sum of middle numbers of array is equal to x which is searched or there is no candidates in whole array.

3. Part - Calculate the running time of the code snippet below

```
for (i=2*n; i>=1; i=i-1)              → O(2n)
   for (j=1; j<=i; j=j+1)             → O(n)
      for (k=1; k<=j; k=k*3)          → O(log n)
         print("hello")
```

Runnig time :

→ O(2n * n * log n) = O(2n^2 * log n) = O(n^2 * log n)

4. Part - Write a recurrence relation for the following function and analyze its time complexity T(n).

```
float aFunc(myArray,n){
    if (n==1){                           -> O(1)
        return myArray[0];
    }
    //let myArray1,myArray2,myArray3,myArray4 be predefined arrays
    for (i=0; i <= (n/2)-1; i++){                    -> O(n/2)
        for (j=0; j <= (n/2)-1; j++){                -> O(n/2)
            myArray1[i] = myArray[i];
            myArray2[i] = myArray[i+j];
            myArray3[i] = myArray[n/2+j];
            myArray4[i] = myArray[j];
        }
    }
    x1 = aFunc(myArray1,n/2);            -> T(n/2)
    x2 = aFunc(myArray2,n/2);            -> T(n/2)
    x3 = aFunc(myArray3,n/2);            -> T(n/2)
    x4 = aFunc(myArray4,n/2);            -> T(n/2)

    return x1*x2*x3*x4;
}
```

Recurrence relation for the following function :

T(n) = Θ (1)   if n=1;

T(n) = 4*T(n/2) + Θ (n^2 / 4)   if n > 1;

Using master theorem for complexity equation:

The parameters are:

a = 4, b = 2, f (n) = n^2/4 ∈ Θ (n^log2(4))

It is the case 2 of Master theorem. The resulting complexity is thus:

T(n) = Θ(n^log2(4) * log(n))