**Gebze Technical University**
**Computer Engineering**

**CSE 222 - 2018 Spring**

# HOMEWORK 4 REPORT

**FURKAN ÖZEV**
**161044036**

Course Assistant: Ayşe Şerbetçi TURAN

# 1 PROBLEM-1

**Given a single linked list of integers, we want to find the maximum length sorted sublist in this list. For example for the list L= {1, 9, 2, 7, 20, 13} the returned list should be S = {2, 7, 20}.**

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

Code:

```
public static LinkedList<Integer> maxList(LinkedList<Integer> list)
{
    // Create 2 list to keep sublists.
    LinkedList<Integer> list1 = new LinkedList<Integer>();
    LinkedList<Integer> list2 = null;

    // counter1 => size list1, counter2 => size list2,
    // temp = keep previous element, now = keep last element.
    Integer temp = 0, now = 0;
    int counter1 = 0, counter2 = 0;

    //Get the first element of the list until there are no elements in the list.
    while((now = list.pollFirst()) != null)
    {
        /* if the element is not smaller than the previous element, add the
                element to the list and increase the counter 1. */
        if(now >= temp)
        {
            list1.add(now);
            counter1++;
        }
        // Otherwise,
        else
        {
            /* if the number of elements of list 1 is greater than the number of
                elements of list 2, move list 1 to list2. then reset list 1. */
            if(counter1 > counter2)
            {
                list2 = list1;
                counter2 = counter1;
                list1 = new LinkedList<Integer>();
                list1.add(now);
                counter1 = 1;
            }
            // Otherwise, reset the list 1.
            else
            {
                list1 = new LinkedList<Integer>();
                list1.add(now);
                counter1 = 1;
            }
        }
```

```java
        // Keep the previous element in temp.
        temp = now;
    }
    // Returns the list with a large number of elements.
    if(counter1 > counter2) return list1;
    else return list2;
}
```

Analyze Complexity: T(n) : Time Complexity, S(n) : Space Complexity

I will do the calculations by dividing the code into certain parts.

n => size of list,  m => amount of sublist.

1.) T1 = Θ (1) , S1 = Θ (1) , Because All processes take constant time.

```java
LinkedList<Integer> list1 = new LinkedList<Integer>();
LinkedList<Integer> list2 = null;

Integer temp = 0, now = 0;
int counter1 = 0, counter2 = 0;
```

2.) T2 = Θ (n) , S2 = Θ (1) , Because loop turn n times, and condition take constant time.

```java
while((now = list.pollFirst()) != null)        // Loop turn n times.
```

3.) T3 = Θ (1), S3 = Θ (m), Because there ise no loop or recursive call,it take constant time.

```java
if(now >= temp)
{
    list1.add(now);             // Call add complexity is Θ(1).
    counter1++;
}
else
{
    if(counter1 > counter2)                    // The add method will be called
    {                                          // 1 time per round.Each round,
        list2 = list1;                         // T3 = Θ (1)
        counter2 = counter1;
        list1 = new LinkedList<Integer>();     // New space
        list1.add(now);         // Call add complexity is Θ(1).
        counter1 = 1;
    }                                          // Take new space up to the amount
    else                                       // of sublist.So, S3 = Θ (m)
    {
        list1 = new LinkedList<Integer>();     // New space
        list1.add(now);         // Call add complexity is Θ(1).
        counter1 = 1;
    }
}
temp = now;
```

4.) T4 = Θ (1), S4 = Θ (1), Just return list take 1 complexity.

```java
if(counter1 > counter2) return list1;
else return list2;
```

Conclusion:

T(n) = T1 + T2*(T3+T4)

= T(n) = T1 + T2*(max(T3,T4)) = T1 + T2*(max(Θ(1), Θ(1))) = T1 + T2* Θ(1)

= T(n) = max(T1,T2* Θ(1)) = max(Θ(1), Θ(n)* Θ(1)) = max(Θ(1), Θ(n))

Result Time complexity: T(n) = Θ(n)

S(n) = Θ(m)

m is amount of sublists, so it not directly connected n. It can be change for situation.

Best case is m=1 So, S(n) = Θ(1)          Worst case is m=n So, S(n) = Θ(n)

Result Space complexity: S(n) = O(n) , S(n) = Ω(1)

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

Code:

```
// Firstly take main list, and free 2 list, counter1 = 0 and counter2 = 0
// counter1 => size list1, counter => size list2
public static LinkedList<Integer> maxList_recursive(LinkedList<Integer> list,
LinkedList<Integer> list1, LinkedList<Integer> list2, int counter1, int
counter2)
{
    // now = keep current element, next = show next element.
    Integer now = list.pollFirst();
    Integer next = list.peekFirst();

    /* If there is no element in the main list, the last element is added to
       list1. Then, if the number of elements of list 1 is greater than 2,
       returns list1, otherwise returns list 2.      */
    if(next == null)
    {
        list1.add(now);
        if(counter1 > counter2) return list1;
        else return list2;
    }

    /* if the next element is not smaller than the current element, add the
                next element to the list and increase the counter 1. */
    else if(next >= now)
    {
        list1.add(now);
        counter1++;
    }
    // Otherwise,
    else
    {
       // Add current element to list1.
       list1.add(now);

       /* if the number of elements of list 1 is greater than the number of
                elements of list 2, move list 1 to list2. then reset list 1. */
        if(counter1 > counter2)
```

```java
        {
            list2 = list1;
            counter2 = counter1;
            list1 = new LinkedList<Integer>();
            counter1 = 0;
        }

        // Otherwise, reset the list 1.
        else
        {
            list1 = new LinkedList<Integer>();
            counter1 = 0;
        }
    }
    // Recursive call method. Then, repeat same process.
    return maxList_recursive(list,list1,list2,counter1,counter2);

}
```

Analyze Complexity: T(n) : Time Complexity, S(n) : Space Complexity

1.) T1 = Θ (1) , S1 = Θ (1) , Because All processes take constant time.

```java
Integer now = list.pollFirst();        //poolFirst and peekFirst 's time
Integer next = list.peekFirst();       //complexity is Θ (1)
```

2.) T2 = Θ (1) , S2 = Θ (1) , Because All processes take constant time.

```java
if(next == null)                                // Base Condition of recursion
{
    list1.add(now);                             //complexity is Θ (1)
    if(counter1 > counter2) return list1;
    else return list2;
}
```

3.) T3 = Θ (1) , S3 = Θ (1) , Processes take constant time but each sublist need new space.

```java
else if(next >= now)
{
    list1.add(now);                        // add method takes constant time.
    counter1++;
}
else
{
    list1.add(now);
    if(counter1 > counter2)                     // Other process takes constant
    {
        list2 = list1;
        counter2 = counter1;
        list1 = new LinkedList<Integer>();      // Reserve new space affect
        counter1 = 0;                           // space complexity.
    }                                           // m = amount of sublists
    else
    {
        list1 = new LinkedList<Integer>();      // Affect space complexity
        counter1 = 0;
    }
}
```

4.) T4 = Θ (T(n)-1) , S4 = Θ (S(n)-1) , Recursive calling.

```java
return maxList_recursive(list,list1,list2,counter1,counter2);
```

Conclusion:

T(n) = T1 + T2 + T3 + T4

T(n) = max(T1,T2,T3) + T4

T(n) = max( Θ (1), Θ (1), Θ (1) ) + T4

T(n) = T4 + Θ (1)

T(n) = Θ (T(n-1)) + Θ (1)

a = 1, b = 1, k = 0

Induction Method:

T(n) = Θ (T(n-1)) + Θ (1)

T(n) = Θ (T(n-1)) + c

T(n-1) = Θ (T(n-2)) + c

T(n) = (Θ (T(n-2)) + c) + c

T(1) = Θ (T(0)) + c => T(1) = c

T(1) = Θ T((n-(n-1))

So, T(n) = Θ (T(0)) + n*c => T(0) = 0 => T(n) = Θ (n*c) (c is constant)

Result : T(n) = Θ(n)

Master Theorem:

T(n)={ $O(n^k)$, if a<1,

$O(n^{(k+1)})$, if a=1 ,

$O(n^k a^{\frac{n}{b}})$, if a >1 }

So a = 1, result is : $O(n^{(0+1)})$= $O(n)$

Result : T(n) = $O(n)$ Also, T(n) = $\theta(n)$

# 2 PROBLEM-2

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

1.) Let's create a function that takes sorted array where the numbers are kept, the size of the array, and the sum to be searched as a parameter. We accept that the array that is a parameter is ordered from small to big.

2.) Create variables that hold left and right indexes. Left index start with 0, the right index start with (size - 1) . In this way, the left index starts from the left of the array , the right index starts from the right of the array.

3.) Until the left and right indexs show the same item, we will compare the sum of the values shown by the left and right indexes and the sum we are looking for.

4.) If the sum of the values shown by the left and right indexes and the sum we are looking for are equal, It will print values shown by the left and right indexes. We have found a pair that provides the sum, we must increase the index to 1 to find other pairs.

5.) If the sum of the values shown by the left and right indexes smaller than the sum we are looking for. It moves from the left to a big element to increase the sum of the values shown by the indexes. To makes this process, it will increase 1 the left index. This will not cause trouble because the array is sequential.

6.) If the sum of the values shown by the left and right indexes bigger than the sum we are looking for. It moves from the right to a small element to reduce the sum of the values shown by the indexes. To makes this process, it will reduce 1 the right index.

7.) The program will continue until the indexes show the same element, ie there are no pairs that can provide the sum we are searching for.

```
static void SearchSum(int[] arr, int size, int x)
{
    int left = 0, right = size - 1;

    while (l < r)
    {
        if(arr[l] + arr[r] == x){
        System.out.printf("\n%d , %d",arr[l], arr[r]);
            l++;}
        else if(arr[l] + arr[r] < x)
            l++;
        else
            r--;
    }
}
```

TIME COMPLEXITY:

1.) The distances of the left index and the right index are equal to the size of the array.

2.) In each cycle, either the left index 1 is increasing or the right index 1 is decreasing.

3.) This means that the distance between them in each round is decreasing 1.

4.) We know the loop will end when the distance between them is 0.

5.) As a result we know that the loop will repeat as much as the size of the array.

6.) Assume that n = size of aray and Time complexity is T(n).

7.) Result : T(n) = Θ(n)

# 3 PROBLEM-3

## 3.1 Calculate the running time of the code snippet below:

```
for (i=2*n; i>=1; i=i-1)
    for (j=1; j<=i; j=j+1)
        for (k=1; k<=j; k=k*3)
            print(“hello”)
```

```
for (i=2*n; i>=1; i=i-1)
// Let T1 (n) be the time complexity of this loop.
    ...............
    ...............
```

```
    for (j=1; j<=i; j=j+1)
    // Let T2 (n) be the time complexity of this loop.
    ...............
    ...............
```

```
        for (k=1; k<=j; k=k*3)
        // Let T3 (n) be the time complexity of this loop.
        ...............
        ...............
```

```
            print(“hello”)
            // Time complexity of print is Θ(1)
```

T1(n) = $\sum_{i=1}^{2n} \sum_{j=1}^{i} T3(n)$

T2(n) = $\sum_{j=1}^{i} T3(n)$

T3(n) = $1 * \log_3 j$ = $\log_3 j$   Because k *= 3 so, Increase amount of logarithmic

T2(n) = $\sum_{j=1}^{i} T3(n)$ = $\frac{T3(i)*T3(i+1)}{2}$ = $\frac{(\log_3 i)*(\log_3 i+1)}{2} \cong \frac{(\log_3 i)*(\log_3 i)}{2} \cong \frac{(\log_3 i)^2}{2} \cong \log_3 i$

T2(n) = $\log_3 i$

T1(n) = $\sum_{i=1}^{2n} \sum_{j=1}^{i} T3(n) = \sum_{i=1}^{2n} \log_3 i = \frac{(2n)*(2n+1)}{2} * \frac{(\log_3 n)*(\log_3 n+1)}{2}$

T1(n) $\cong \frac{(2n)*(2n+1)}{2} * \frac{(\log_3 n)*(\log_3 n)}{2} = \frac{4n^2+2n}{2} * \frac{(\log_3 n)^2}{2} = (2n^2+n) * \log_3 n$

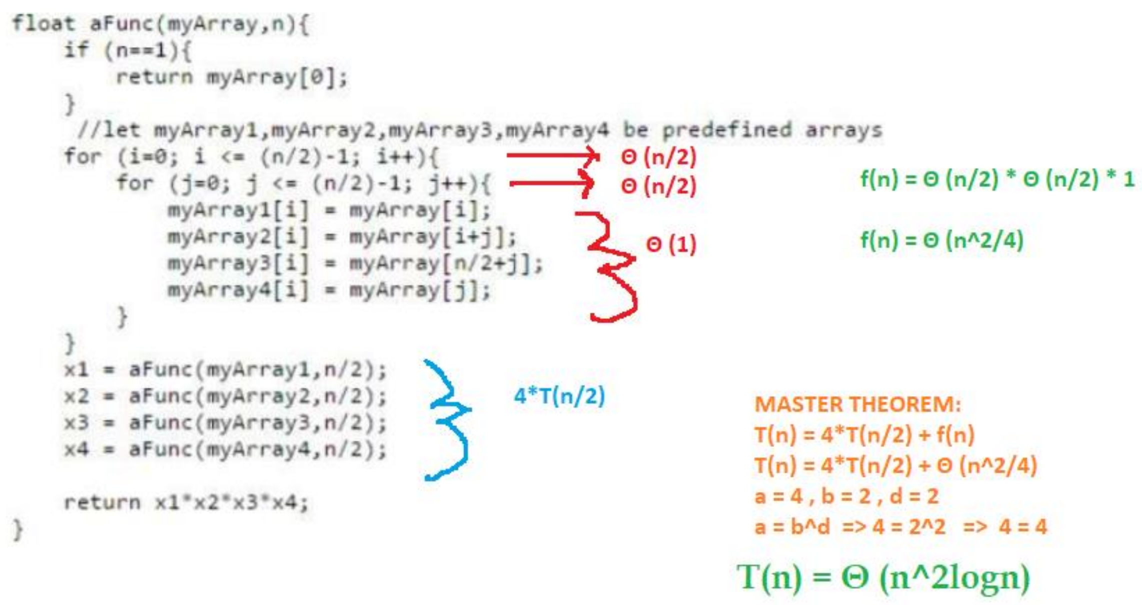
T1(n) = $(2n^2+n) * \log_3 n = \theta(n^2 logn)$

T1(n) = $\theta(n^2 logn)$

RESULT: T(n) = $\theta(n^2 logn)$

# 4 PROBLEM-4

## 4.1 Write a recurrence relation for the following function and analyze its time complexity T(n).




1.) First loop turns (n/2) times and second loop turns (n/2). Each round it will do 7 processes that are taken constant time. So, $f(n) = \theta\left(\frac{n}{2}\right) * \theta\left(\frac{n}{2}\right) * \theta(1) = \theta\left(\frac{n^2}{4}\right) = \theta(n^2)$

2.) 4 times the function calls itself again. Each call invokes the size as n / 2.

So, Recursive complexity is 4T$\left(\frac{n}{2}\right)$

3.) T(n) = 4*T$\left(\frac{n}{2}\right)$) + f(n) = 4*T$\left(\frac{n}{2}\right)$) + $\theta(n^2)$

4.) a = 4 , b = 2 , d = 2 a = $b^d$ 4 = 4

5.) $T(n) = \theta(n^d \log n) = \theta(n^2 \log n)$