Gebze Technical University
Computer Engineering

CSE 222 - 2019 Spring

HOMEWORK 4 Q1-Q4

EMRE KAVAK
151044085

Course Assistant:

1.A

I wrote a function for find the maximum length sorted sublist in a given list. This function find maximum length list in a given list and return it. I have 2 lists for hold previous list and next list. When I find lower element, I copy backupSubList into maxSublist if size is bigger than maxSublist.

Function :

```
public static List<Integer> findMaxSortedList(LinkedList<Integer> list){
        Iterator<Integer> it =list.iterator();
        List<Integer> maxSublist = new ArrayList<>();
        List<Integer> backupSublist = new ArrayList<>();
        int size1 = 0, previous = 0, next = 0, flag = 0;
        previous = it.next();              // first element for compare with next element
        while(it.hasNext()){               // This loop will search n times. So, complexity O(n)
           next = it.next();
           if(next > previous) {           // if next element large than previous, it will add the list
              backupSublist.add(previous);          // O(1)
              size1++;                              // 1
              flag=1;                               // 1
           }
           else {                           // if not, last number will be add end of list
              if(size1 >=1)  {                      // 1
                 backupSublist.add(previous);       // O(1)
                 size1++;                           // 1
              }
               if(flag == 1 && size1 >= 1){    // for copy list if there is a ordered list // 1
                   if(backupSublist.size()>= maxSublist.size()){    // 1
                     maxSublist.clear();                           // n
                     maxSublist.addAll(backupSublist);             //n
                     backupSublist.clear();  // and backup list will be clear for add new lists.//n
               }
                 flag = 0;        //1
              }
              else backupSublist.clear(); // n
              size1=0;            // 1
           }
```

```
            previous = next;
        }
        if(backupSublist.size()>= maxSublist.size()){   // final check for max size list.
            maxSublist.clear();                        // for this , I should check backupList.
            maxSublist.addAll(backupSublist);          // copy
            backupSublist.clear();                     // clear list
            maxSublist.add(next);                      // the last element will be added.
        }
        flag = 0;
        return  maxSublist;
    }
```

Complexity : This function search all elements in a given list linear search and there is constans more than 5 and n. We ignore constansts and multiplis. So, its complexity is O(n).

1.B

I wrote a recursive function for find maximum length list in a given list. This function find maximum list in a given list with recursively and return maximum list. Paremeters is two list and an iterator. (**Note** : You should create an iterator for check this function) I thought iterator is well because of I should search a Linked list and its easy to search linear with ıterator. I add element into backuplist until found lower element and when I found lower element, I will copy it into maxList. Finally I return this maxList.

Function :

```
public static List recursiveFindMaxSortedList(List<Integer> maxList,List<Integer> backupList, Iterator<Integer> it){
        if(it == null || !it.hasNext()) {   //  check if iterator is null or not have a next element.
            if(backupList.size()> maxList.size()){
 //if the last elements ordered and its size bigger than maxList, it will be copy and clear
                maxList.clear();                       // before add, list must be removed.
                maxList.addAll(backupList);            // it will copy into maxList.
                backupList.clear();
            }
            return maxList;                            // return max list
        }
        int previous = 0 , next = 0;
        if(it.hasNext()){
            if(backupList.size()>0)
                previous = backupList.get(backupList.size()-1); // last element
            if(it.hasNext())
                next = it.next();
            if(next>previous){
                backupList.add(next);
                recursiveFindMaxSortedList(maxList,backupList,it); // recursive call
            }else{
                if(backupList.size() > maxList.size()){
                    maxList.clear();        // n
                    maxList.addAll(backupList);
                    backupList.clear();  // and backup list will be clear for add new lists.
                }else backupList.clear();
                    backupList.add(next);
                recursiveFindMaxSortedList(maxList,backupList,it); // recursive call
            }
        }
        return maxList;            // return list
    }
```

Complexity : recursiveFindMaxSortedList function traverse the list linearly. So its complexity is O(n). Lets prove it ;

Induction Prove:

Assume that ∃c is constant and lets say T(1) = c and T(n) = T(n-1) + c

We assume n>=0 and T(n)<=n.

For n>=0, base case : T(0) = 0 and c = 0 and for n>0 T(n) = c + T(n-1)

Lets we accept it is true, we should try it for T(n+1) <= n +1.

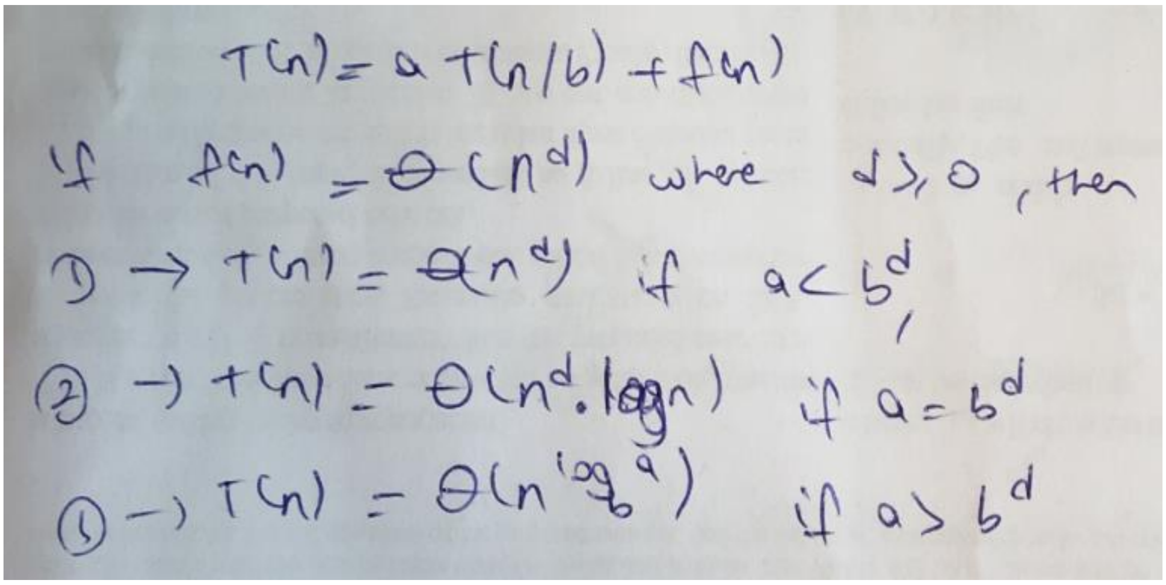
T(n +1) = 1 + T(n) (from induction hypothesis) and T(n+1) <= n+1

So, for n inputs, this function complexcity is **O(n).**

Master Teorem Prove :

$T(n) = aT(n/b) + f(n)$

• n is the size of the problem.
• a is the number of subproblems in the recursion.
• n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
• f (n) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.



$T(n) = a\,T(n/b) + f(n)$

if $f(n) = \Theta(n^d)$ where $d \geq 0$, then

① $\rightarrow T(n) = \Theta(n^d)$ if $a < b^d$,

② $\rightarrow T(n) = \Theta(n^d . \log n)$ if $a = b^d$

③ $\rightarrow T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

Table 1 (our asistant give it us)

For this function;

- a = 1
- b = 2
- f(n) = n

T(n) = T(n/2) + n
Accoring to table 1 (below); a<b^d, Complexity is **Θ (n).**

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

Algorithm :

```
int leftIndex= 0 and  rightIndex = arr.length-1, int i = 0;
While i < arr.length && flag == 0
    sum  = arr[rightIndex] + arr[leftIndex];
    İf(sum>x) rıghtIndex --
    İf(sum<x) leftIndex++
    If(sum == x)
        Print arr[rightIndex] and arr[rightIndex]
        Assign flag 1;
        increase i;
```

Analyze : I have a sorted array input and x variable refers to sum of two elements. I have 2 iterators left and right. Left refers to 0. element(first) and right refers to last element in the array. İf two index sum is greater than x, rightIndex will decrease, otherwise leftIndex will increase. If equal, two array element will be printed. So, I will search all elements in the given array.

Complexity : This algorithm complexity is **Θ (n).** Because of I should check elements both of side and it means I will check all elements in the array.The worst-case running time of linear search grows like the array size n. For this running time we can say complexity is Θ (n). When we use tetha notation, we mean that we have an asymptotically tight bound on the running time. "Asymptotically" because it matters for only large values of n*n*n. "Tight bound" because we've nailed the running time to within a constant factor above and below.

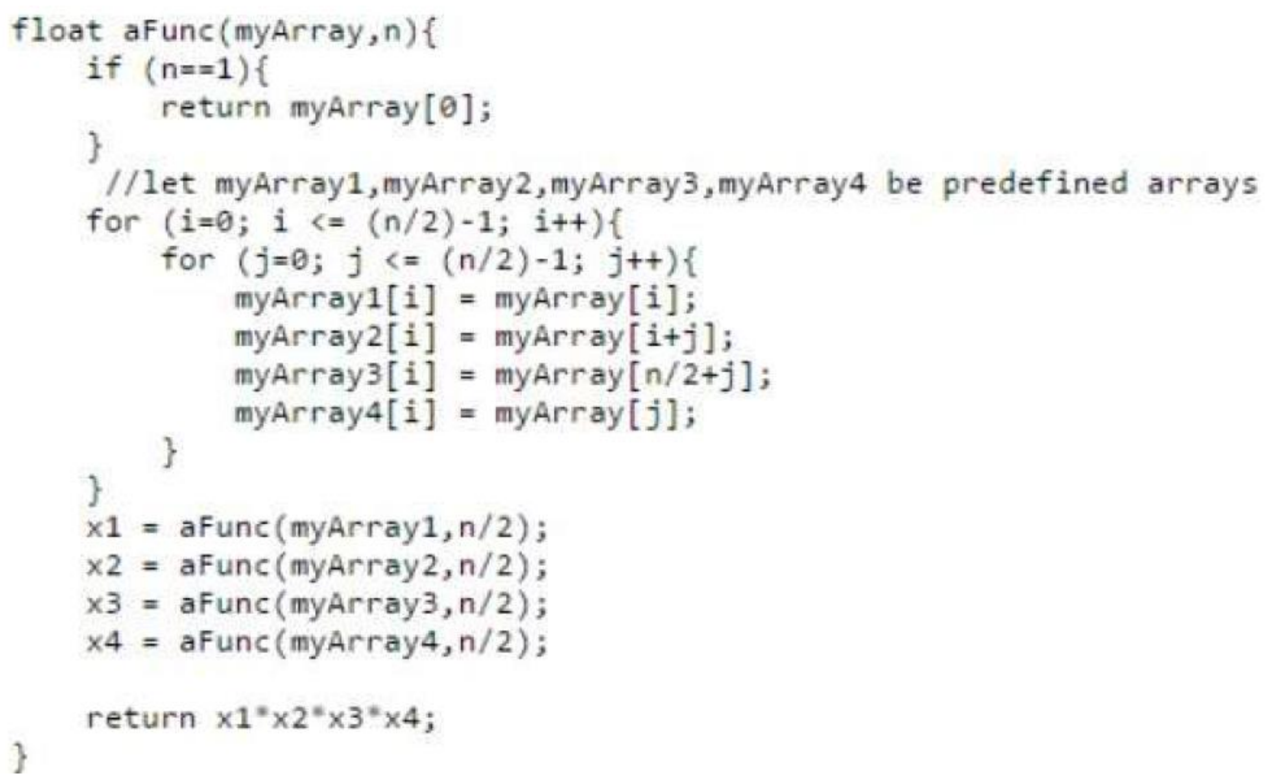
3 - Calculate the running time of the code snippet below

```
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 (k=1; k<=j; k=k*3)
          print(“hello”)
  ```
  // this for loop complexity logn. Because of k is multiple with 3 and this means logn.

-
  ```
  for (i=2*n; i>=1; i=i-1)
      for (j=1; j<=i; j=j+1)
  ```
  // this for loop and bottom loop complexity is n^2.
  Because of 1+2+3…n = n*(n+1)/2 and we have 2n.
  So, 2n*(2n +1)/2 = 2n^2 +n (we choice n^2)
  With first complexity we should multiply complexitis.Because of they are nested. Finally, This function complexity is **O(n^2logn)**.(ignored constant 2 and lower n)

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



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

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

If we use Master teorem, it says;

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1,\, b > 1$$

The master theorem concerns recurrence relations of the form: In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:
• n is the size of the problem.
• a is the number of subproblems in the recursion.
• n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
• f (n) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.
So ;

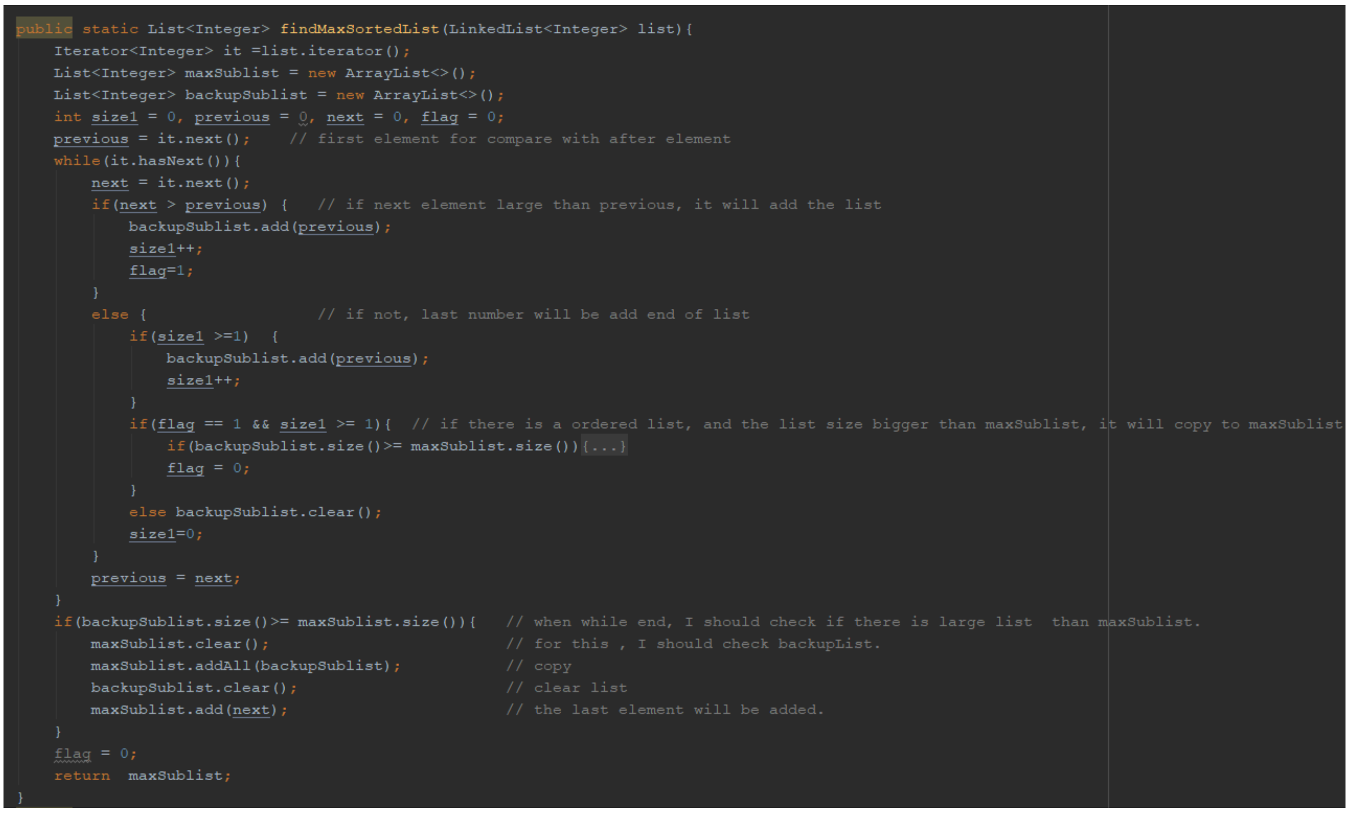
- our functions size is n.
- a is 4 because of we have four recursive call.
- f is n^2. Because of we have nested loops. Firs loop run Time is (n/2 -1) second loop Time same as loop1 one. So, we should multiply this.
  (n/2 -1) * (n/2 -1) = O(n^2) (we should choice bigger and ignore constants).
- each problem size is n/2. So, b is 2.

For calculate, I used **Table 1**

a = 4, b = 2, f(n) = n^2
T(n) = 4T(n/2) + n^2
a = 2^2, so this function time complexity is **Θ(n^2* logn)**

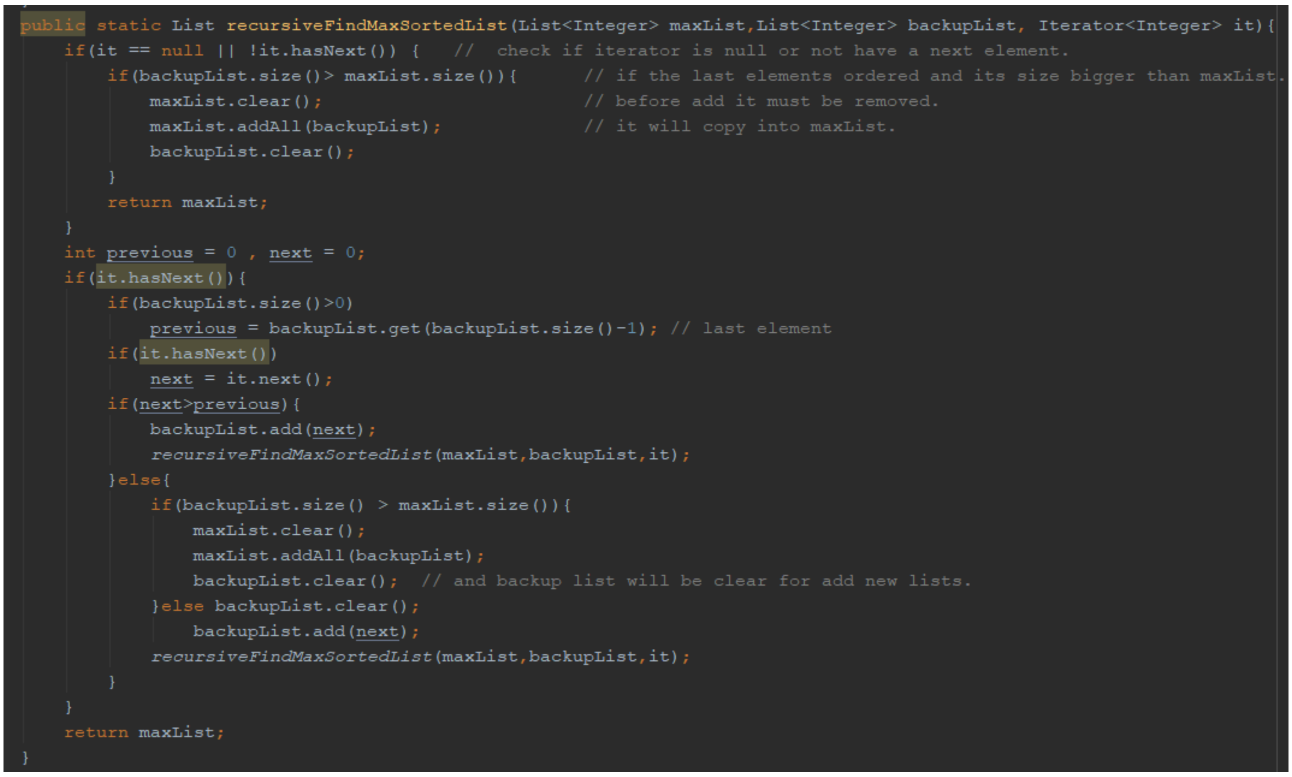
TESTS FOR Q1

1.A



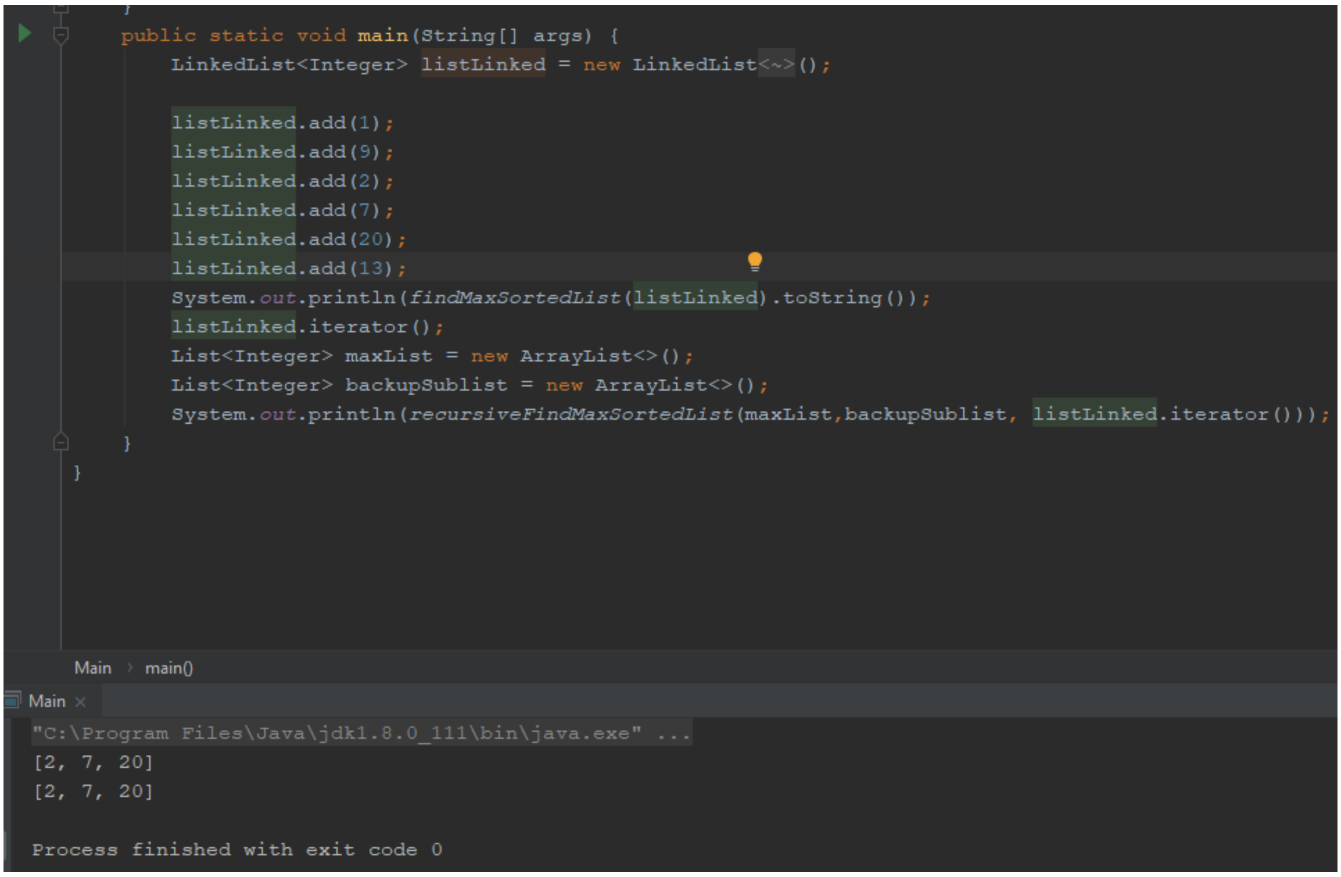
```java
public static List<Integer> findMaxSortedList(LinkedList<Integer> list){
    Iterator<Integer> it =list.iterator();
    List<Integer> maxSublist = new ArrayList<>();
    List<Integer> backupSublist = new ArrayList<>();
    int size1 = 0, previous = 0, next = 0, flag = 0;
    previous = it.next();     // first element for compare with after element
    while(it.hasNext()){
        next = it.next();
        if(next > previous) {   // if next element large than previous, it will add the list
            backupSublist.add(previous);
            size1++;
            flag=1;
        }
        else {                      // if not, last number will be add end of list
            if(size1 >=1)   {
                backupSublist.add(previous);
                size1++;
            }
            if(flag == 1 && size1 >= 1){  // if there is a ordered list, and the list size bigger than maxSublist, it will copy to maxSublist
                if(backupSublist.size()>= maxSublist.size()){...}
                flag = 0;
            }
            else backupSublist.clear();
            size1=0;
        }
        previous = next;
    }
    if(backupSublist.size()>= maxSublist.size()){   // when while end, I should check if there is large list  than maxSublist.
        maxSublist.clear();                         // for this , I should check backupList.
        maxSublist.addAll(backupSublist);           // copy
        backupSublist.clear();                      // clear list
        maxSublist.add(next);                       // the last element will be added.
    }
    flag = 0;
    return  maxSublist;
}
```

1.B



```java
public static List recursiveFindMaxSortedList(List<Integer> maxList,List<Integer> backupList, Iterator<Integer> it){
    if(it == null  || !it.hasNext())  {   //  check if iterator is null or not have a next element.
        if(backupList.size()> maxList.size()){       // if the last elements ordered and its size bigger than maxList.
            maxList.clear();                         // before add it must be removed.
            maxList.addAll(backupList);              // it will copy into maxList.
            backupList.clear();
        }
        return maxList;
    }
    int previous = 0 , next = 0;
    if(it.hasNext()){
        if(backupList.size()>0)
            previous = backupList.get(backupList.size()-1); // last element
        if(it.hasNext())
            next = it.next();
        if(next>previous){
            backupList.add(next);
            recursiveFindMaxSortedList(maxList,backupList,it);
        }else{
            if(backupList.size() > maxList.size()){
                maxList.clear();
                maxList.addAll(backupList);
                backupList.clear();  // and backup list will be clear for add new lists.
            }else backupList.clear();
                backupList.add(next);
            recursiveFindMaxSortedList(maxList,backupList,it);
        }
    }
    return maxList;
}
```

TESTS FOR Q1.A AND Q1.B

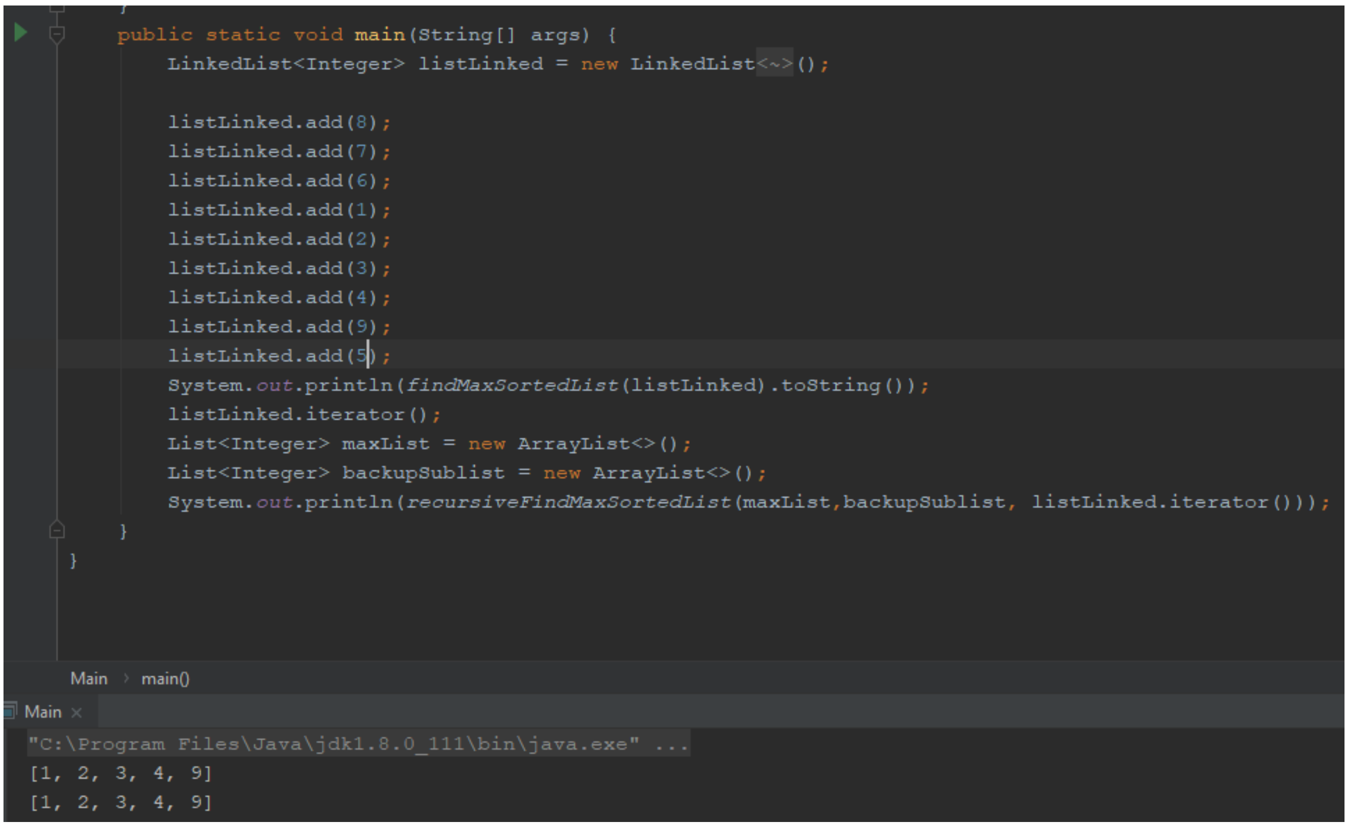


```
    public static void main(String[] args) {
        LinkedList<Integer> listLinked = new LinkedList<~>();

        listLinked.add(1);
        listLinked.add(9);
        listLinked.add(2);
        listLinked.add(7);
        listLinked.add(20);
        listLinked.add(13);
        System.out.println(findMaxSortedList(listLinked).toString());
        listLinked.iterator();
        List<Integer> maxList = new ArrayList<>();
        List<Integer> backupSublist = new ArrayList<>();
        System.out.println(recursiveFindMaxSortedList(maxList,backupSublist, listLinked.iterator()));
    }
}
```

Main › main()

Main

```
"C:\Program Files\Java\jdk1.8.0_111\bin\java.exe" ...
[2, 7, 20]
[2, 7, 20]

Process finished with exit code 0
```



```
    public static void main(String[] args) {
        LinkedList<Integer> listLinked = new LinkedList<~>();

        listLinked.add(8);
        listLinked.add(7);
        listLinked.add(6);
        listLinked.add(1);
        listLinked.add(2);
        listLinked.add(3);
        listLinked.add(4);
        listLinked.add(9);
        listLinked.add(5);
        System.out.println(findMaxSortedList(listLinked).toString());
        listLinked.iterator();
        List<Integer> maxList = new ArrayList<>();
        List<Integer> backupSublist = new ArrayList<>();
        System.out.println(recursiveFindMaxSortedList(maxList,backupSublist, listLinked.iterator()));
    }
}
```

Main › main()

Main

```
"C:\Program Files\Java\jdk1.8.0_111\bin\java.exe" ...
[1, 2, 3, 4, 9]
[1, 2, 3, 4, 9]
```