Given a single linked list of integers, we want to find the maximum length sorted sub list in this list. For example for the list L = (1,9,2,7,29,13) the returned list should be S = (2,7,20).

a. Write an iterative function which performs this task. Analyze it's complexity:

Firstly I will describe how I solved this problem iteratively. We have a linked list so we can use it's iterator in order to traverse between elements. I found the maximum sublist and it's beginning and end indexes using iterator. After finding indexes it is easy to return a new linked list from the old list.

I will write two procedures: One of them finds the sublist's head and tail. Other one takes the list as parameter and returns a new linkedlist.

Procedure findMinMax(list(), head, tail) – returns void

while(iterator has next element)
    get first element using iterator.
    get second element using iterator.
    if(first element is smaller than second element)
        increment the sublist by one.
    Else
        If(max is smaller than subCount)
            Assign subCount to max
            Assign current index – max to maxIndex
        If end
    Reset subCount to 1
    Else end
Iterate to next element
While end
head = maxIndex
tail = maxIndex + max
procedure end

procedure returnBiggestSubList(list oldList list biggestSubList int head int tail) – returns a list

```
while(oldList has next element)
        if(index is equals to head)
                while(counter < max)
                        biggestSubList.add(oldList.next)
                        counter++;
                return biggestSublist
                while end
        if end
        oldList.next
while end
procedure end
```

In order to calculate time complexity I need to show to procedures time complexities. I'm not calling these two procedures inside of each other. Therefore the total time complexity will equal to procedure which has bigger time complexity.

Calculating time complexity of findMinMax procedure:

There is only one while loop and it iterates over the list with the help of iterator therefore it's O(n). If we haven't used iterator in this situation time complexity could be O(n^2) because in order to get specified element we had to iterate over over and again. Therefore iterator used for linked list.

findMinMax time complexity = O(n)

Calculating time complexity of returnBiggestSubList procedure:

There is a while statement iterates till head of subset and there is another while loop iterates till sublist's length. Therefore we have two nested loops. **It's looks like O(n^2) but it's not O(n^2).** The reason is explained below:

If list has n elements outer while loop will execute n times in worst case scenario. Inner loop's duty is only iterate till subset's length which is always smaller than n. Since we know the locations of head and tail of biggest subset inner while only continues to execute outer while's last location.

returnBiggestSubList time complexity = O(n)

Total time complexity = O(n) + O(n) = O(n)

I'm leaving iterative version of my code here: https://paste.ubuntu.com/p/BN6HvSNVt9/

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

```
Procedure findLongestSubsetRecursive(Iterator iterator, List SubMax) returns a List

        If(iterator hasNext element)
                Return subMax
        ifend
        If(submax.size == 0  OR  iterator.next == submax's last element)
                Submax.add(iterator.next)
                Return findLongestSubsetRecursive(iterator, submax);
        ifend
        Else
                newList = new LinkedList<Integer>
                newList.add(i.next)
                retList = mls(iterator, newList);

                if(submax.size() greater than retList.size())
                        return submax;
                return retList;
        else end
procedure end
```

Calculating time complexity using master theorem:
Since we have separate two recursive calls, we have O(1) complexity for base case. On other cases we'll have 2 * T (n) recursive calls. Since we don't have any loops the complexity is O(n) for single call. If we summarize all of this, we will get:

Induction:

Base Case:

T(1) is true since function returns subset in base case.

Induction step:

For "T(i) is true for all smaller numbers, the call mls() return first second and so on.
Using this hypothesis, we need to prove T(k).

Using induction, we can prove that the call mls() returns the value for iterators single step.
Therefore it's O(n) too.

$1 + T(k– 1)$
$1 + (1 + T(k – 2)) = 2 + T(k – 2)$
$2 + (1 + T(k – 3)) = 3 + T(k – 3)$
.
.
$k – 1 + T(1)$
$k – 1 + 1 = O(n)$

Master Theorem:

According to master theorem $a(T(n/b) + cn^{k}$, $T(1) = c$;
Since $a < b ^ k$ $(1 < 2 ^ 1)$ $T(n) = O(n^1) = O(n)$.

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.

Since our array is sorted, we can check one by one from end and beginning. Let's assume that our array has size of N. After that we'll create two variables named with *beginning* and *end*.

Let *beginning* equal to 0 which is head of the list and *let* end equal to N – 1 which is end of the list.

Now only thing we have to do is iterate till we found desired x value. I wrote pseudocode below:

```
Procedure findXSum(list)


beginning = 0, end  = N -1

while(begining < end)

    if(list[beginning] + list[end] == X)

        return true; (That means we've found the sum value.)

    else if(list[beginning] + list[end] < sum)

        beginning++;

    else if(list[beginning] + list[end] > sum)

        end--;

while end

procedure end
```

Time complexity of Θ(n) algorithm:

The problem is easy because in question it says we already have sorted list. For the solution The worst case is O(n). And the average case is where the element is in middle. It's O(n) too. Therefore we can use theta notation for this case. **Finally my algorithm has Θ(n) complexity.**

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)
        {
            System.out.println("Hello");
        }
    }
}
```

In this part I will calculate the big oh notation of given code.

First let's look at most outer loop:

```
for(i = 2 * n ; i >= 1 ; i = i -1)
```

This loop starts from 2 * n and iterates till i is equal to 1. Since i decreases one by one nothing tricky here. Simple linear complexity. O(2 * n) which is also O(n)

Secondly looking at second loop:

```
for(j = 1 ; j <= i ; j = j + 1)
```

This loop is linear too, without a doubt is O(n). Let's examine it further:

The first time through the outer loop the inner loop is executed n-1, n-2 n-3 and lastly one time. We know that outer loop executed 2 * n times. So we'll get the following expression for T(n)(for both loops):

(2n – 1) + (2n – 2) + (2n – 3) + ... + 2 + 1

So the sum will be: ((2 * n) * (2 * n – 1) / 2) which is 2 n ^ 2 – n.

O(2 n ^ 2 - n) = O (n ^ 2).

Finally we calculated the complexities of first two loops. It's O(n ^ 2).

Lastly looking at the last most inner loop:

```
for(k = 1 ; k <= j ; k = k * 3)
```

k increases with logarithmic power. Also we know that outer loops have O(n^2) complexity**. Therefore all three loops complexity is:** O(logn) * O(n ^ 2) = **O((n ^ 2) * (log n)).**

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

The function is:

```
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;
}
```

What happens exactly here?

Let's check out base case:

```
if (n==1){
    return myArray[0];
}
```

If n is equal to 1 then we return the first element. This is base case of recursive call, therefore it has **O(1) complexity.**

After that we see two nested loops iterates n/2 to n/2:

```
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];
    }
}
```

Outer loop iterates n / 2 time and inner loop iterates n / 2 time. Inner loop makes necessarry changes on array so we can add 4 instructions to calculation. Since our loops are nested we can calculate the big oh notation as: O(((n/2)) *((n/2)) * 4) which is O(n^2)

Lastly checking number of recursive calls:

```
x1 = aFunc(myArray1,n/2);
x2 = aFunc(myArray2,n/2);
x3 = aFunc(myArray3,n/2);
x4 = aFunc(myArray4,n/2);
```

Here we have 4 recursive calls as T(n/2) so our expression will be: 4 * T(n / 2).

According to master theorem a = 4, b = 2 and k = 2 since a = b ^ k it's equals to n^k logn.

T(n) = 4 * T (n / 2) + n^2. Therefore complexity is **O(n^2 * logn)**