Bachelor in Informatics Engineering	Algorithms and Data Structures **28th of June, 2016.** **EXAM, EXTRAORDINARY CALL**

Name: ....

Surname: ....

PLEASE, READ CAREFULLY THE FOLLOWING INSTRUCTIONS BEFORE START

1. All student data asked in this document must be filled. Please, use a pen to write in this document (not a pencil)
2. This document and all additional sheets must be delivered upon completion.
3. Please, add page numbers in all pages before delivering.
4. You cannot leave the classroom until the exam has ended.
5. Mobile phones must be disconnected during the exam.
6. **In this exam, the use of classes provided by the Java API (ArrayList, LinkedList, etc.). The use of Arrays is also forbidden.**

PROBLEM 1 (2,5 point)
In this exercise, linear ADTs must be applied to detect and delete a pattern. This pattern is described s follows: given a number, the following one must have double value. Let's suppose that a linear ADT is available, implemented as a singly linked list of integers (type "int" in JAVA).
You must implement a method that searches into the ADT for the described pattern and, if found, removes all the elements in the pattern but the first one. This method does not have parameters and returns the number of removed nodes.

The following example shows the result after executing the method three consecutive times.

- Start:
    - Initial ADT content (example): 4, 1, 2, 4, 8, 9, 3, 2, 5, 10, 20, 2, 1, 7
    - ADT content after first execution: 4, 1, 9, 3, 2, 5, 2, 1, 7
    - Method answer: 5 (that is, 5 removed nodes)

Identify and describe the worst case and compute the complexity (Big Oh!) of the implemented method. Do the same with the best case.

Note: Any auxiliary method used must be implemented after the asked method implementation.

Solution:

```
public int removeSequence(){
    SNode prev=firstNode;
    int count=0;
    if(prev!=null){
        int ref;
        ref=prev.elem;
        if(prev.next!=null){
            for (SNode nodeIt = prev.next; nodeIt != null; nodeIt = nodeIt.next) {
                if(nodeIt.elem==2*ref){
                    ref*=2;
                    prev.next=nodeIt.next;
                    count++;
                }
                else{
                    ref=nodeIt.elem;
                    prev=nodeIt;
                }
            }
        }
    }
    return count;
}
```

The worst case and the average case have linear complexity O(n) because it is always needed to traverse the whole list.
In the best case the list is empty, and complexity order is O(1)

PROBLEM (2 points)
Consider the following iterative method to check if a number is prime or not. Remember that a number is prime if it is greater than 0 and can only be divided by 1 and itself.

```
public boolean isPrimeIt(int number) {
        if (number <= 0) {
            System.out.println("The number must be greater than 0.");
            return false;
        } else {
            boolean prime= true;
            for (int x=2; x<number && prime; x++) {
                if (number % x == 0)  prime= false;
            }
            return prime;
        }
    }
```

1. (0.5 points) Compute the running time function and the complexity order (Big Oh!) for the method isPrimeIt().
2. (1.5 points) Implement a **recursive** method **public boolean** isPrime to check if a number is prime or not.

Solution:

```
a)      public static boolean isPrimeIt(int number) {
                if (number < 0) { //1
                        System.out.println("The number must be greater than 0."); //1
                        return false; //1
                } else {
                        for (int div=2; div<number; div++) // 1+ (n-2)+1 + (n-2) = 1 + 2n-2
                          if (number % div == 0)  //2(n-2)=2n-4
                                return false;          //1(n-2)=n-2
                        return true; //1
                }
        }
        //T(n)= 1+ 2n-2 +2 +2n-4+1 = 4n-2 => O(n) => Linear complexity

b)      public static boolean isPrime(int number, int div) {
                if (number < 0) {
                        System.out.println("The number must be greater than 0.");
                        return false;
                } else
                        if (number==1 || number==div) //number= 1 or the number has been
reached
                                return true;
                        else
                                if (number % div == 0)
                                        return false;
                                else
                                        return isPrime(number, div+1);    //check if the
following number is a divider
        }

        public static void main(String[] args) {
                System.out.println("Is 15 a prime number? " + isPrime(15,2)); //false
                System.out.println("Is 83 a prime number? " + isPrime(83,2)); //true
        }
```

PROBLEM 3 (3 points)

Given the following class representing a binary node:

```
public class BSTNode {
        public int elem;

        public BSTNode parent;
        public BSTNode left;
        public BSTNode right;

        public BSTNode(int e) {
                elem=e;
        }

        //returns its rightmost node of its left child
        public  BSTNode predecessor(BSTNode node) {
                if (node==null) return null;
                BSTNode pred=node.left;

                if (pred!=null) {
                        while (pred.right!=null) {
                                pred=pred.right;
                        }
                }
                return pred;
        }
```

```
    //returns its leftmost node of its right child
    public  BSTNode successor(BSTNode node) {
            if (node==null) return null;
            BSTNode suc=node.right;
            if (suc!=null) {
                    while (suc.left!=null) {
                    suc=suc.left;
                    }
            }
            return suc;
    }

    //returns the sizde of this subtree
    public  int getSize(BSTNode node) {
            if (node==null) return 0;//
            return 1 + getSize(node.left)+getSize(node.right);
    }

}
```

a) (2 points) Implement a recursive method getting an object of type BSTNode as an input and checking if it satisfies the definition for a binary search tree (BST). *successor* and *predecessor* methods can help you in the design of the solution. You also have the base cases for the recursive method as an input:

```
    public  boolean isBST(BSTNode node) {
                    //first base case: the node is null
                    if (node==null) return true;
                    //second base cases: the node is a leaf
                    if (node.left==null && node.right==null) return true;

                    BSTNode pred=predecessor(node);
                    BSTNode suc=successor(node);

                    //Include the rest of the code



            }
```

Solution

```
    public  boolean isBST(BSTNode node) {
                    if (node==null) return true;
                    else if (node.left==null && node.right==null) return true;

                    BSTNode pred=predecessor(node);
                    BSTNode suc=successor(node);

                    if (pred==null)
                            return suc.elem>node.elem && isBST(node.right);
                    else if (suc==null)
                            return pred.elem<node.elem && isBST(node.left);
                    else
                            return pred.elem<node.elem && suc.elem >node.elem &&
                                    isBST(node.left) && isBST(node.right);
```

```
            }
```

b) (0.5 points) Implement a method to compute the size balance factor for a given node.

Solution:

```
private int balanceFactorSize(BSTNode node) {
        if (node==null) {
                return 0;
        }
        return Math.abs(getSize(node.right)-getSize(node.left));
}
```

c) (0.5 points) Implement a method to chek if the tree is size balanced (that is, all the nodes have a size balance factor less or equal than 1).

Solution:

```
public boolean isBalanceSize () {
                return isBalanceSize(root);
}

public boolean isBalanceSize(BSTNodeContact node) {
                if (node==null) return true;
            return (balanceFactorSize(node)<=1 && isBalanceSize(node.left) &&
            isBalanceSize(node.right));
}
```

PROBLEM 4 (2.5 points).

The Social Network Mynet wants to store information about users and relations among them. For this purpose, a graph structure is used. There are four types of relations: first-degree family relations (for parents, brothers and sisters), other-degree family relations (for uncles, cousins, grandparents and so on), work relations and friends. There cannot be people involved in two types of relations at the same time (if someone is linked to me through a work relation, that person cannot be linked to me through a friend relation).

The following code is used to define the graph:

```
public class GraphMA {

        boolean matrix[][];
        //maximum number of vertices
        int maxVertices;
        //current number of vertices
        int numVertices;
        //true if the graph is directed, false in other case
        boolean directed;

        //constructor
        public GraphMA(int max, boolean d) {
              matrix=new boolean[max][max];
              maxVertices=max;
              numVertices=0;
        directed=d;
    }
}
```

a) (0.75 point) The given implementation is valid to represent the social network described? If the answer is no, why? And, how would you adapt it?

Solution

No, there is no room for the type of relation.

```
public class GraphMA {

    public enum relationType {FAM_FIRST_DEGREE,
                FAM_SECOND_DEGREE, WORK, FRIEND};

String matrix[][];
//maximum number of vertices
int maxVertices;
//current number of vertices
int numVertices;
//true if the graph is directed, false eoc
boolean directed;

//constructor
public GraphMA(int max, boolean d) {
    matrix=new String[max][max];
    maxVertices=max;
    numVertices=0;
    directed=d;
    }
}
```

b) (0.75 point) Is this a directed or undirected graph? Please, give the implementation for a method to connect two users through any type of relation.

Solution
The common interpretation would be that this is an undirected graph (you could also do the interpretation that you could see somebody as your friend but that person could consider you as his/her work colleague, not a friend. In this solution we are not considering this circumstances).

```
public void addEdge(int i, int j, relationType rel) {
    if (checkIndex(i) && checkIndex(j)) {
        matrix[i][j]=rel.toString();
        if (!directed)  matrix[j][i]=rel.toString();
    }
}
```

```
public boolean checkIndex(int i){
	if (i<0 || i>=numVertices) return false;
	return true;
}
```

c) (1 point) Provide an implementation for the ‘show()’ method for the social network graph. Which is the complexity for this method? Would an implementation based on adjacency lists reduce this complexity?

Solution

```
public void show(){
	if (numVertices==0) {
		System.out.println("The graph is empty!!");
		return;
	}
	for (int i=0;i<numVertices;i++) {
		for (int j=0;j<numVertices;j++) {
			System.out.print(matrix[i][j]+"\t");
		}
		System.out.println();
	}
}
```

The running time function for this method would be: $T(N) = 1 + (1 + n + n + 1) + n*(1 + n + n + 1) + n*n = 2n + 3 + 2n^2 + 2n + n^2 = 3n^2 + 4n + 3$

So using Big-Oh! notation, complexity is $O(n^2)$.

If an implementation based on an adjacency list is used complexity is not reduced as the vertices array must be traversed and, for each vertice, the list of adjacent vertices must also be traversed. We have again nested loops so complexity would be again $O(n^2)$. The answer to the question is no.