Part I. Multiple Choice Questions

1. D
2. A
3. B
4. C
5. B
6. B
7. B
8. C
9. A
10. C
11. C
12. B
13. C
14. C
15. A
16. B

Part II. Written Answers
Write your answers **only** in the space provided.

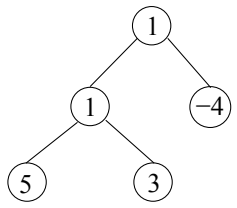
17. (3 marks) Consider the following algorithm.

```
public int traverse(BinaryTreeNode r) {
   if (r == null) return 0;
   else {
       int v = traverse(r.getLeftChild()) - traverse(r.getRightChild());
       if (v > r.getElement()) return v;
       else return r.getElement();
}
```

If the following statement is executed

```
int result = traverse(r);
```

where `r` is the root of the tree on the right.



What value will `result` have? 2

18. (5 marks) Consider the following Java code.

```
public static void main (String[] args) {
   int[] a = new int[3];
   int size = 0;
   init(a,size);
   for (int i = 1; i < size; ++i) System.out.println(a[i]);
   System.out.println(a[0]);
}
private static void init(int[] a, int size) {
   for (int i = 0; i < 3; ++i) {
       ++size;
       a[i] = (1-i) * size;
   }
}
```

Show the output produced by this algorithm. Values must be printed in the same order as the algorithm prints them.

1

19. Consider the following Java code.

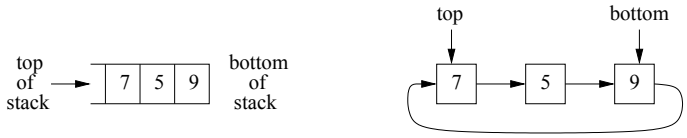
```
public static void main(String[] args) {
   int[] a = {3, 5, 2};
   int result = m(a,0,2);
}
public static int m(int[] a, int first, int last) {
   if (first > last) return 1;
   else {
       int mid = (int)Math.floor((first+last)/2);
       return m(a,first,mid-1)+m(a,mid+1,last);// Line 1
   }
}
```

where method `Math.floor(x)` rounds the value of the parameter `x` down to the nearest integer. For example, `Math.floor(3.2) = 3`, `Math.floor(3) = 3`.

(3 marks) What value does `result` have at the end? 4

(2 marks) How many recursive calls does the algorithm make (number of calls made in Line 1)? 6

20. (6 marks) Consider the implementation of a stack using a circular linked list. A circular linked list is a singly linked list where the last node of the list points to the first node of the list instead of to null. Each node of the list stores one element of the stack. There is a reference variable called `top` pointing to the node storing the value at the top of the stack and a reference variable called `bottom` pointing to the node storing the value at the bottom of the stack. For example, a stack storing values 9, 5, and 7, with 7 at the top and 9 at the bottom of the stack is represented with the circular liked list below.

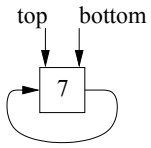


Consider the following implementation of the `pop()` operation. Assume that elements stored in the stack are of type `T`; `top` and `bottom` are instance variables. Methods `getElement`, `getNext`, and `setNext` get the data stored in a node, get a reference to the next node in the list, and change the reference to the next node in the list, respectively.

```
private T pop() {
   T element = top.getElement();
   top = top.getNext();
   bottom.setNext(top);
   return element;
}
```

Indicate whether this is a correct implementation of the pop operation. If the implementation is incorrect, explain why it is incorrect by giving an example showing that the code will not perform the pop operation correctly. If the implementation is correct, explain why it always correctly removes and returns the element at the top of the stack.

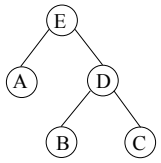
The implementation is incorrect. There are 2 situations where it will not give the desired results. First, when the stack has only one element, that element will not be removed when a `pop` operation is performed. In this case the stack before and after the `pop` operation will look like this:



Second, if the stack is empty, the above code will not execute correctly as then `top = bottom = null` and the above code will throw a null pointer exception.

21. (3 marks) Draw a binary tree in which every node stores one of the letters A, B, C, D, E and such that

- a preorder traversal visits the nodes in this order: E A D B C, and
- an inorder traversal visits the nodes in this order: A E B D C.



For each one of the following 4 questions, compute the time complexity of the given code. You **must explain how you computed the time complexity** and you must give the order (big-Oh) of the time complexity. (**Hint.** Your answer might be like this: "Number of operations performed outside the loops: x; number of operations performed in one iteration of the inner loop: y; number of iterations of the inner loop when i = 0 is z_1, when i = 1 is $z_2, \ldots$; total number of operations performed by the loops: w. Total number of operations performed by the algorithm: x + w. The order of the time complexity is O(v).") The following fact might be useful to you: $\sum_{k=1}^{m} = \frac{m(m+1)}{2}$.

22. (7 marks)

```
int x = 0;
for (int i = 0; i < n; i = i+1)
   for (int j = 0; j < n; j = j+1) {
       if (j < i) j = j + n;
       else x = x+1;
   }
```

In each iteration of the inner loop a constant number c_1 of operations are performed. When $i = 0$ the inner loop performs n iterations: one for each $j = 0, 1, \ldots, n-1$. However, for each value $i = 1, 2, \ldots, n-1$ the inner loop performs only one iteration; this is because for these values of i when $j = 0$ the condition of the if statement is true so the value of j is set to n ending the inner loop.

Hence, the total number of iterations of the inner loop is $2n-1$ and the total number of operations performed by the loops is $(2n-1)c_1$. Outside the loop a constant number c_2 of operations are performed, so the total number of operations performed by the algorithm is $c_2 + c_1(2n-1)$; thus the time complexity is $O(n)$.

23. (7 marks)

```
int x = 1;
for (int i = 0; i <= n * n; i = i + n)
   for (int j = 0; j < i; j = j + n)
       x = x + 1;
```

Each iteration of the inner loop performs a constant number c_1 of operations. The number of iterations of the inner loop depends on the value of i as shown in the table below.

	number of iterations of inner loop	number of operations in i-th iteration of outer loop
$i = 0$	0	0
$i = n$	1 ($j = 0$)	c_1
$i = 2n$	2 ($j = 0, n$)	$2c_1$
$\vdots$	$\vdots$	$\vdots$
$i = n^2$	n ($j = 0, n, 2n, \ldots, n(n-1)$)	nc_1

Outside the loops a constant number c_2 of operations are performed. Adding the number of operations in the last column of the table we get that the total number of operations performed by the algorithm is $c_2 + c_1 + 2c_1 + 3c_1 + \ldots + nc_1 = c_2 + c_1 \sum_{i=1}^{n} i = c_2 + c_1 \frac{n(n+1)}{2}$. Therefore, the time complexity is $O(n^2)$.

24. (7 marks)

```
int j = 1;
int i = 1;
while (i <= n) {
   if (i == n)
       if (j < n) {
           i = 1;
           j = j+1;
       }
       else i = n+1;
   else i = i+1;
}
```

Each iteration of the while loop performs a constant number c of operations. The following table shows the number of iterations and operations performed by the loop.

	number of iterations	number of operations
$j = 1$	n $(i = 1, 2, \ldots, n)$	cn
$j = 2$	n $(i = 1, 2, \ldots, n)$	cn
$j = 3$	n $(i = 1, 2, \ldots, n)$	cn
$\vdots$	$\vdots$	$\vdots$
$j = n - 1$	n $(i = 1, 2, \ldots, n)$	cn
$j = n$	n $(i = 1, 2, \ldots, n)$	cn

Outside the while loop there is a constant number c_1 of operations. Hence, adding the number of operations in the last column of the table we get that the total number of operations performed by the algorithm is $c_1 + (cn)n = c_1 + cn^2$. Therefore, the time complexity is $O(n^2)$.

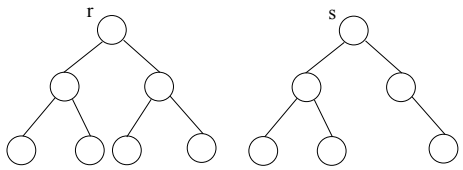
25. (7 marks) In the following algorithm node `r` is the root of a binary tree with $n > 0$ nodes.

```
public int traverse(BinaryTreeNode r) {
   int res = 2;
   if (r.getLeftChild() != null) res = res + traverse(r.getLeftChild());
   if (r.getRightChild() != null) res = res + traverse(r.getRightChild());
   return res;
}
```

First, ignoring the recursive calls, the algorithm performs a constant number c of operations. The algorithm performs one recursive call for each non-null child of every internal node of the tree; hence the recursive calls make the algorithm visit or process once each one of the nodes of the tree. Therefore, the number of recursive calls is n. Since each call performs c operations and the number of calls is n, the total number of operations performed by the algorithm is cn, so the time complexity is $O(n)$.

For the following 4 questions write algorithms in Java or in detailed Java-like pseudocode like the ones used in the lecture notes.

26. (12 marks) A binary tree is *symmetric* if for every internal node the number of nodes in its left subtree is the same as the number of nodes in its right subtree. Given a node `p`, let `p.size()` return the number of nodes in the subtree with root `p`, and `p.getLeftChild()` and `p.getRightChild()` return the left and right children of `p`, respectively. Write a recursive algorithm `isSymmetric(r)` that receives as parameter the root `r` of a binary tree and it returns `true` if the tree is symmetric and it returns `false` otherwise. For example for the tree below with root `r` the algorithm must return true, but for the tree with root `s` it must return false.




```
Algorithm isSymmetric(r)
Input: Root r of a binary tree.
Output: True if the tree is symmetric; false otherwise

if r = null then return true
else if r.getLeftChild() = null and r.getRightChild() = null then
        return true
     else if r.getLeftchild() = null or r.getRightchild() = null then
             return false
          else if r.getLeftChild().size() ≠ r.getRightChild().size() then
                  return false
               else if isSymmetric(r.getLeftChild() = false then
                       return false
                    else return isSymmetric(r.getRightChild()
```

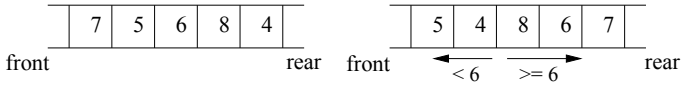
27. (8 marks) Let `q` be a queue storing `n` data items. Write an algorithm `reverse (q)` to reverse the queue so that the first element in `q` (the element at the front of the queue) becomes last (the element at the rear of the queue), the second element becomes the second last and so on. The **only** methods that you can use to manipulate the queue are `q.enqueue(element)` that adds an element to the rear of the queue, `q.dequeue()` that removes the element at the front of the queue, and `q.isEmpty()` that returns true if the queue is empty and it returns false if the queue is not empty. You **cannot** use any auxiliary data structures. (Hint. Design a recursive algorithm similar to the one in one of the questions of the midterm.)

```
Algorithm reverse(q)
Input; queue q
Output: Reverse queue

if q.isEmpty() then return q
else {
   elem = q.dequeue()
   q1 = reverse(q)
   q1.enqueue(elem)
   return q1
}
```

28. (11 marks) Let `q` be a queue storing `n` integer values. Write an algorithm `partition(q,target)` that receives as parameter `q` and an integer value `target` and it re-arranges the values in the queue so that all the values smaller than `target` appear closer to the front of the queue than any values larger than or equal to `target`. For this algorithm you can use **one** auxiliary stack `s`. You **cannot** use any other auxiliary data structures. You can use the following queue methods: `q.dequeue()`, `q.enqueue(element)`, `q.isEmpty` and `q.size()` (that returns the number of elements in the queue). You can also use the following stack methods: `s.push(element)`, `s.pop()`, and `s.isEmpty()`.

For example, for the following queue on the left side and `target` = 6, your algorithm should produce a queue like the one on the right.



Algorithm `partition(q,target)`
Input: queue `q` storing `n` integer values and value `target`
Output: queue re-arranged so al values smaller than `target` appear closer to the front than values larger than `target`

```
s = new empty stack
n = q.size()
for i = 1 to n do {
   elem = q.dequeue()
   if elem < target then q.enqueue(elem)
   else s.push(elem)
}
while s.isEmpty() = false do
   q.enqueue(s.pop())
```

29. (11 marks) Write an algorithm `remove(target)` to remove the node storing a given value `target` from a sorted singly linked list. If no node in the list stores value `target` then the algorithm must return `null` otherwise it must return `target`. Instance variable `first` is a reference variable pointing to the first node in the list. If `current` is a reference to a node, `current.next()` returns a reference to the next node in the list (or `null` if `current` points to the last node in the list), `current.setNext(succ)` makes the node referenced by `current` point to node `succ`, and `current.getElement()` returns the data item stored in the node pointed by `current`.

Algorithm `remove(target)`
Input: `target` value
Output: `target` if a node storing `target` was removed; `null` otherwise

```
prev = null
curr = first
while (curr ≠ null) and (curr.getElement() ≠ target) do
   prev = curr
   curr = curr.next()
}
if curr = null then return null
else {
        if prev = null then first = curr.next()
        else prev.setNext(curr.next())
        return target
}
```