



EXAMINATIONS — 2012  
**Trimester 1, MID-TERM TEST**

**COMP103**  
**Introduction to**  
**Data Structures and Algorithms**  
**SOLUTIONS**

**Time Allowed:** 45 minutes

- Instructions:**
1. Attempt **all** of the questions.
  2. *Read each question carefully before attempting it.*
  3. This test will be marked out of **45** marks (i.e. allocate one minute per mark).
  4. Write your answers in the boxes in this test paper and hand in all sheets.
  5. Documentation on some relevant Java classes, interfaces, and exceptions can be found at the end of the paper.

<b>Questions</b>	<b>Marks</b>
1. Collections	[6]
2. Implementing a Stack	[19]
3. Sorting	[10]
4. Recursion	[10]

**SPARE PAGE FOR EXTRA ANSWERS**

Cross out rough working that you do not want marked.  
Specify the question number for work that you do want marked.

**Question 1. Collections**

[6 marks]

(a) [6 marks] In the boxes below give a short (e.g. one sentence) description for each of the following collections identifying their *key feature(s)*.

Bag

Can have duplicates. No restrictions.

Set

Cannot have duplicates.

Map

Maps keys to values. Cannot have duplicate keys Can have duplicate values.

Stack

Respects the Last In First Out order of access for values.

Queue

Respects the First In First Out order of access for values.

List

Assigns a position for each value stored.

## Question 2. Implementing a Stack

[19 marks]

(a) [8 marks] Complete the following class that implements a Stack using an array. Note that pop() should through an EmptyStackException if the stack is empty.

```
public class Stack<E> {
    static int INITIAL_CAPACITY = 2;
    E[] data;

    public Stack() {

    }

    public void push(E item) {

    }

    public E pop() {

    }

}
```

(Question 2 continued on next page)

(Question 2 continued)

ANSWER CODE (2 marks for each of the first 3 methods and 2 marks for ensureCapacity):

```
public class Stack<E> {
    static int INITIAL_CAPACITY = 2;
    E[] data;
    int count;

    public Stack() {
        data = (E[]) new Object[INITIAL_CAPACITY];
        count = 0;
    }

    public void push(E item) {
        ensureCapacity();
        data[count] = item;
        count++;
    }

    public E pop() {
        if (count <= 0) throw new EmptyStackException();
        E item = data[count - 1];
        count--;
        return item;
    }

    public void ensureCapacity() {
        if (count >= data.length) {
            E[] newData = (E[]) new Object[data.length*2];
            for (int i = 0; i < count; i++) {
                newData[i] = data[i];
            }
            data = newData;
        }
    }
}
```

(Question 2 continued on next page)

**(Question 2 continued)**

**(b)** [6 marks] Write an iterator for your Stack class above that is defined as a private inner class and goes through the items in the stack *from the top to the bottom* - in other words, the *first* item should be the one that would have come off *first* if a `pop` method were to be called. For simplicity, assume that you don't need to implement the `remove` method and that you don't need to worry about anyone modifying the Stack while you are iterating through it.

```
import java.util . Iterator ;
public class Stack<E> implements Iterable<E> {
    // CODE FROM PREVIOUS SUBQUESTION IMPLEMENTING STACK
    public Iterator <E> iterator() {
        return new StackIterator<E>(this);
    }
    private class StackIterator <E> implements Iterator<E> {
        Stack<E> stack;
        int position ;

        public StackIterator (Stack<E> stack) {

        }

        public boolean hasNext() {

        }

        public E next() {

        }

        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```

(Question 2 continued on next page)

(Question 2 continued)

ANSWER CODE (2 marks each of the first 2 methods and 2 marks for the next())

```
import java.util . Iterator ;
public class Stack<E> implements Iterable<E> {
    // CODE FROM PREVIOUS SUBQUESTION IMPLEMENTING STACK
    public Iterator <E> iterator() {
        return new StackIterator<E>(this);
    }
    private class StackIterator<E> implements Iterator<E> {
        Stack<E> stack;
        int position;

        public StackIterator (Stack<E> stack) {
            this.stack = stack;
            this.position = stack.count;
        }

        public boolean hasNext() {
            return this.position > 0;
        }

        public E next() {
            this.position--;
            return this.stack.data[this.position];
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

(Question 2 continued on next page)

**(Question 2 continued)**

**(c)** [3 marks] What will the following code print when using the Stack you have just implemented?

```
public static void main(String[] a) {  
    Stack<String> ss = new Stack<String>();  
    ss.push("A"); ss.push("B"); ss.push("C");  
    System.out.println(ss.pop() + ss.pop() + ss.pop());  
    for (String s : ss)  
        System.out.println(s);  
    System.out.println("-----");  
}
```

CBA

**(d)** [2 marks] What is the best order of items when iterating through a Stack? Justify your answer.

The order is depends on the user: both "from last to first" and "from first to last" can make sense in different situations and as stack processes items in "first in first out" order, neither is a perfect fit.



**Question 3. Sorting**

[10 marks]

(a) [3 marks] Consider the *insertion sort* as discussed in the lectures. Give an example of an array of 10 elements that would produce the *best* cost behaviour for insertion sort and explain briefly why.

A, B, C, D, E, F, G, H, I, J Because insertion step will always pick the best letter that will be greater than the end of the "sorted" part and thus there will be no need to shift items.

(b) [3 marks] Consider the *insertion sort* as discussed in the lectures. Give an example of an array of 10 elements that would produce the *worst* cost behaviour for insertion sort and explain briefly why.

J, I, H, G, F, E, D, C, B, A Because insertion step will always have to take the next item all the way down to the start of the "sorted" part thus maximising the number of shifts required.

(c) [2 marks] What does it mean for a sorting algorithm to be *stable*?

If two items are the same (equal) their order will be the both before and after the sort is performed.

(d) [2 marks] Imagine you are given two sorting algorithms: one that has worst cost  $O(n \log(n))$  and one that has worst cost  $O(n^2)$ . Which of these two algorithms is going to sort 1 trillion integers faster? Justify your answer.

We do not know as the cost is given for the "almost all" values of  $n$  in the "long term". There can be exceptions (e.g. when a specific case of  $n = 1000000000000$  is given) as to which one is faster, rather the important part is that "overall" the  $O(n \log(n))$  algorithm will be faster.

#### Question 4. Recursion

[10 marks]

(a) [4 marks] Fix the recursive code below that tries to calculate a factorial. A reminder:  $n! = 1 * 2 * \dots * (n - 1) * n$ .

```
public int factorial ( int n) {
    if (n > 1) {
        return n * factorial (n);
    } else {
        return 1;
    }
}
```

**ANSWER TO ABOVE: Replace recursive call to factorial(n) with factorial(n-1)!**

(b) [4 marks] Write a recursive function that returns the  $n$ th Fibonacci number. A reminder that Fibonacci sequence is as follows:  $1, 1, 2, 3, 5, 8, \dots, f_{n-2}, f_{n-1}, f_n, \dots$  where  $f_n = f_{n-1} + f_{n-2}$ .

```
public int fib ( int n) {
}
}
```

**ANSWER:**

```
public int fib ( int n) {
    if (n < 1) return 0;
    if (n == 1) return 1;
    if (n == 2) return 1;
    return fib (n-2) + fib (n-1);
}
```

(c) [2 marks] Explain why this recursive version of fib is particularly inefficient. Justify your answer.

The two calls mean that the same number is recalculated at least twice. If one could cache these intermediate results somehow, the calculation would work much faster!

\*\*\*\*\*

## Appendix (may be removed)

**Brief (and simplified) specifications of some relevant interfaces and classes.**

**public class** *Scanner*

```
public boolean hasNext(); // there is more to read
public String next(); // return the next token (word)
public String nextLine(); // return the next line
public int nextInt (); // return the next integer
```

**public class** *Random*

```
public int nextInt(int n); // return a random integer between 0 and n-1
public double nextDouble(); // return a random double between 0.0 and 1.0
```

**public interface** *Iterator* <*E*>

```
public boolean hasNext();
public E next();
public void remove();
```

**public interface** *Iterable* <*E*>

```
public Iterator <E> iterator();
```

// Can use in the "for each" loop

**public interface** *Comparable* <*E*>

```
public int compareTo(E o);
```

// Can compare this to another *E*

**public interface** *Comparator* <*E*>

```
public int compare(E o1, E o2);
```

// Can use this to compare two *E*'s

```
public class Collections
    public void sort(List<E>, Comparator<E>)
```

```
public interface Collection<E>
    public boolean isEmpty();
    public int size ();
    public boolean add();
    public Iterator <E> iterator();
```

```
public interface List<E> extends Collection<E>
    // Implementations: ArrayList
    public E get(int index);
    public void set(int index, E element);
    public void add(E element);
    public void add(int index, E element);
    public void remove(int index);
    public void remove(Object element);
```

```
public interface Set extends Collection<E>
    // Implementations: HashSet, SortedSet, TreeSet
    public boolean contains(Object element);
    public boolean add(E element);
    public boolean remove(Object element);
```

```
public interface Queue<E> extends Collection<E>
    // Implementations: LinkedList, PriorityQueue
    public E peek (); // returns null if queue is empty
    public E poll (); // returns null if queue is empty
    public boolean offer (E element);
```

```
public class Stack<E> implements Collection<E>
    public E peek (); // returns null if stack is empty
    public E pop (); // returns null if stack is empty
    public E push (E element);
```

```
public interface Map<K, V>
    // Implementations: HashMap, TreeMap, LinkedHashMap
    public V get(K key); // returns null if no such key
    public void put(K key, V value);
    public void remove(K key);
    public Set<Map.Entry<K, V>> entrySet();
```