

Note: This practice test is longer with more questions than the actual test.

## 1. Write the output from mystery functions

1a) Write the output generated by the method call `mystery("X", 6)`;

```
public void mystery(String s, int digit) {
    if(digit <= 1)
        System.out.println(digit);
    else {
        s = s + "<";
        mystery(s, digit - 2);
        System.out.println(s + digit);
    }
}
```

1b) Write the return values from each call to the method named `mystery1`.

\_\_\_\_mystery1(0)    \_\_\_\_mystery1(1)    \_\_\_\_mystery1(2)    \_\_\_\_mystery1(3)    \_\_\_\_mystery1(4)

```
public int mystery1(int n) {
    if (n <= 0)
        return 1;
    else
        return 3 + mystery1(n - 1);
}
```

1c) Write the return values from each call to the method named `mysteryTwo`.

\_\_\_\_\_ mysteryTwo("T")    \_\_\_\_\_ mysteryTwo("ab")    \_\_\_\_\_ mysteryTwo("123")

```
public String mysteryTwo(String s) {
    if (s.length() == 0)
        return "";
    else
        return mysteryTwo( s.substring(1, s.length()) ) + "/" + s.charAt(0);
}
```

## 2) Use a recursive definition to write a recursive method

2a) Implement the `trib` function recursively. Do not use a loop.

$$\text{trib}(n) \begin{cases} 1 & \text{if } n \leq 3 \\ \text{trib}(n-1) + \text{trib}(n-2) + \text{trib}(n-3) & \text{if } n > 3 \end{cases}$$

2b) Implement the **power** function recursively. Do not use a loop.

$$\text{power}(x, n) \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \text{power}(x, n - 1) & \text{if } x \geq 1 \end{cases}$$

2c) Implement the **factorial** function recursively. Do not use a loop.

$$\text{factorial}(n) \begin{cases} 1 & \text{if } n \leq 1 \\ x \cdot \text{factorial}(n - 1) & \text{if } x > 1 \end{cases}$$

### 3) Write methods to do some abstract thing

3a) Given a string, let recursive method **changeXY** compute recursively (no loops) a new string where all the lowercase 'x' chars have been changed to 'y' chars. Do not use a loop.

```
changeXY("codex") -> "codey"  
changeXY("xxhixx") -> "yyhiyy"  
changeXY("xhixhix") -> "yhiyhiy"  
  
public String changeXY(String str) {
```

3b) Complete recursive method `printInt` to print an integer with commas in the correct places. Do not use a loop. `printInt(12004)` should print 12,004 and `printInt(123456780)` should print 123,456,780

```
public void printInt(int n) {
```

3c) Write recursive method `isPalindrome` that returns true if the string argument is a palindrome. A palindrome is a string that reads exactly the same backward as forward such as "racecar", but not "RaceCar". Empty strings are palindromes. Do not use a loop.

```
public boolean isPalindrome(String str) {
```

#### 4) Recursive solutions to array processing

4a) Complete recursive method **getRange** that returns the range in a filled array of `int` referenced by `x`. The range is defined as the largest minus the smallest. Do not use a loop. You may use a helper method.

```
public int getRange(int[] x) {
```

4b) Complete recursive method **sum** that returns the sum of all the `int` elements in a filled array referenced by `y`. Do not use a loop. You must have a recursive call somewhere in your answer.

```
public int sum(int[] x) {
```

4c) Complete recursive method **printArrayInReverse** that prints all array elements in a filled array of ints referenced by `x` in reverse order. Do not change the array. Do not use a loop. You may use a helper method.

```
public void printArrayInReverse(int[] x)
```

## 5) Recursive solutions to linked structure processing

Add methods to this collection class that uses a singly linked data structure. Note: There is no size method or instance variable allowed.

```
public class LinkedList {  
  
    private class Node {  
        private int data;  
        private Node next;  
  
        public Node(int element, Node nextRef) {  
            data = element;  
            next = nextRef;  
        }  
    }  
  
    private Node front;  
  
    public LinkedList() {  
        front = null;  
    }  
  
    public void addFirst(int el) {  
        front = new Node(el, front);  
    }  
}
```

5a) Write recursive method **numberOfOdds** to return the number of odd integers in a `LinkedList` object. Do not use a loop.

```
public int numberOfOdds() {
```

5b) Complete method **occurrencesOf** that returns the number of times an int element occurs in a `LinkedList` object.

```
public int occurrencesOf(int el) {
```

5c). Write method **addLast** that recursively adds the int argument to the end of the singly linked structure. Do not use a loop.

```
public int addLast(int el) {
```

```
} // End class LinkedIntList
```

**Use the following interfaces for stack and queue methods to use.**

```
public interface OurQueue<E> {  
    // Return true if this queue has 0 elements  
    public boolean isEmpty();  
  
    // Store a reference to any object at the end  
    public void enqueue(E newEl);  
  
    // Return a reference to the object at the  
    // front of this queue (you may need to cast)  
    public E peek() throws NoSuchElementException;  
  
    // Remove a reference to the element at the front of this Queue and return that reference  
    public E dequeue() throws NoSuchElementException;  
}
```

```
public interface OurStack<E> {  
    // Check if the stack is empty to help avoid popping an empty stack.  
    public boolean isEmpty();  
  
    // Put element on "top" of this Stack object.  
    public void push(E element);  
  
    // Return reference to the element at the top of this stack.  
    public E peek() throws EmptyStackException;  
  
    // Remove element at top of stack and return a reference to it.  
    public E pop() throws EmptyStackException;  
}
```

6. Write the output generated by the following code.

```
OurStack<Character> s = new LinkedStack<Character>();
s.push('3');
s.push('2');
System.out.println(s.peek());
System.out.println(s.pop());
System.out.println(s.isEmpty());
```

7. Write the output generated by the following code.

```
OurQueue<String> q = new LinkedQueue<String>();
q.enqueue("a");
q.enqueue("b");
while(! q.isEmpty())
    System.out.println(q.isEmpty() + " " + q.dequeue());
```

8. Write the output generated by the following code

```
OurStack<Character> s = new LinkedStack<Character>();
try {
    s.push('A');
    s.pop();
    System.out.println("before");
    s.pop();
} catch (EmptyStackException ese) {
    System.out.println("middle");
}
System.out.println("after ");
```

9. Write method **doubleQueue** that will cause every element in the `OurQueue` argument to be duplicated. If the `OurQueue` object contains the string objects "a", "b", "c", "d", "e" (with "a" at the front of the queue and "e" at the end) the `LinkedQueue` should be made to contain "a", "a", "b", "b", "c", "c", "d", "d", "e", "e" (with "e" still at the end). The `OurQueue` referenced by `q` could have any number of elements in it.

```
// Operations include dequeue(), peek(), and isEmpty()
OurQueue<String> q = new LinkedQueue<String>();
q.enqueue("a");
q.enqueue("b");
q.enqueue("c");
// any number of enqueues could be added here before the call to you method.
doubleQueue(q);

public void doubleQueue(OurQueue<E> q) {
```