Practice Exam 1 (Exam 1 date: Friday 3/4)

Logistics:

- In-class, written exam.
- Closed-book, notes, and technology.
- All relevant Java API documentation will be provided for you.
- Exam is out of 100 points (10% of your overall grade).
- This practice exam mimics the layout and types of problems on the actual exam.
- There may be additional sorts of questions asked on the exam—buyer beware!
- All topics covered through Wednesday 10/5 are fair game.

Name: ________________

Solutions!__________

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1:  Cellulose Coding Fibers

Consider the following methods:

```java
public static int[] mystery1(int k) {
    int[] arr = new int[k];
    for (int i = 0; i < k; i++) {
        arr[i] = i + 1;
    }
    // Point A
    mystery2(arr)
    // Point C
    return arr;
}

public static void mystery2(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        arr[i] = arr[i] * arr[i + 1];
    }
    // Point B
}
```

Give complete stack-and-heap diagrams outlining the state of memory at each of the given points above. The style of your diagrams should be identical to that presented in class. You should reproduce the diagrams completely at each point of the program. Assuming execution starts with mystery1(k) for some integer k.

---

// POINT A

- mystery1
- k
- arr
- [ ]

// POINT B

- mystery1
- k
- arr
- [ ]

- mystery2
- arr
- [ ]

// POINT C

- mystery1
- k
- arr
- [ ]
Problem 2:  I Have a Proposition for You

Consider the following Java method:

```java
public static int calculate(int x1, int x2, Scanner console) {
    int result = 0;
    int input = console.nextInt();
    if (x1 > x2) {
        int temp = x2;
        x2 = x1;
        x1 = temp;
    }
    // Point A  ! (x1 > x2)
    while (x1 < input && input < x2) {
        // Point B
        result += input;
        if (input - x1 > x2 - input) {
            x1 = console.nextInt();
        } else {
            x2 = console.nextInt();
        }
    }
    // Point C
    return result;
}
```

For each of the propositions below, determine if the proposition never holds (×), sometimes holds (?), or always holds (✓) at the given program points.

<table>
<thead>
<tr>
<th></th>
<th>x1 &gt; x2</th>
<th>input &lt; x2</th>
<th>result &gt;= 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>×</td>
<td>?</td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>×</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>C</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

In a sentence or two, describe what the method returns as output:

The repeated additions of some randomly generated number between x1 and x2 until x1 < input < x2 no longer holds as x1 and x2 are randomly reassigned.

(Yes, this is pretty arbitrary. The actual test question won't be so arbitrary!)
Problem 3: Developing a Complex

For each of the following methods:

1. Give a mathematical function or recurrence that models the time complexity of the method. State explicitly what operations your function tracks as well as what the input to the function represents. If your model is a recurrence relation, solve that relation for an explicit mathematical function.

2. Give a tight upper-bound for your function using Big-O notation. You can simply state the upper-bound rather than formally proving it correct.

```java
public static int f1(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i][0];
        for (int j = 0; j < arr.length; j++) {
            sum += arr[i][j]
        }
    }
    return sum;
}
```

(1) \[ T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n + n^2 \]

(2) \[ T(n) \in O(n^2) \]

\[ T(n) = \text{# of array accesses as a function of the input array's length} \]

```java
public static String f2(String s, int i) {
    if (i < s.length()) {
        return s.charAt(i) + f2(s, i + 1) + s.charAt(i);
    } else {
        return "";
    }
}
```

(1) \[ T(0) = 0 \]

(2) \[ T(n) \in O(n) \]

\[ T(n) = 2 + T(n-1) = 2k + T(n-k) = 2n \]

\[ k = \#\text{ of unfoldings} \]

\[ T(n) = \text{# of string concats as a function of the length of the input string} \]
Problem 4: The Classiest Problem

Write a complete Java class, DonationTracker, that tracks donations for a charity event. The class should support the following constructor and methods:

- DonationTracker(): constructs a new, empty DonationTracker.
- void add(String name, int amount): adds the given donation to the tracker. The donation amount must be non-negative; if it is not, the method throws an IllegalArgumentException.
- int sum(): reports the total amount of money in the donation tracker.
- void printAll(): prints the donations to the console, one per line, in the format: "name : amount".

You are free to create any auxiliary classes you need to solve the problem. You may also use the ArrayList<T> class from the Java standard library in your solution. The List<T> interface defines three useful methods for this problem: void add(T v), T get(int n), and int size().

```java
public class Donation {
    private int amt;
    private String name;
    public Donation(int a, String s) {
        amt = a;
        name = s;
    }
    public int getAmt() { return amt; }
    public String getName() { return name; }
}

public class DonationTracker {
    private ArrayList<Donation> donations;
    public DonationTracker() {
        donations = new ArrayList<>();
    }
    public void add(String name, int amt) {
        if (amt < 0) {
            throw new IllegalArgumentException();
        } else {
            donations.add(new Donation(amt, name));
        }
    }
    public int sum() {
        int ret = 0;
        for (int i = 0; i < donations.size(); i++) {
            ret += donations.get(i).getAmt();
        }
        return ret;
    }
    public void printAll() {
        for (int i = 0; i < donations.size(); i++) {
            Donation d = donations.get(i);
            System.out.println(d.getName() + " : " + d.getAmt());
        }
    }
}
```
Problem 5: A Link to the Past

Consider the following implementation of a generic linked list:

```java
public class Node<T> {
    public T value;
    public Node<T> next;
    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
public class LinkedList<T> {
    private Node<T> first;
    // ...
}
```

Write a method for the LinkedList<T> class called `removeEvery(n)` that removes every n-th element from the list, starting with the first. In addition, state the worst case time complexity of the method using Big-O notation in terms of the length of the list n; you don’t need to formally prove this bound.

```java
public void removeEvery(int n) {
    if (n == 1) {
        first = null;
    } else if (first != null) {
        first = first.next;
        Node<T> cur = first;
        while (cur != null && cur.next != null) {
            for (int i = 0; i < n - 2; i++) {
                cur = cur.next;
                if (cur.next == null) return;
            }
            cur.next = cur.next.next;
            cur = cur.next;
        }
    }
    Time complexity = O(n)
}
```