1.**Write a static method named `flip` that simulates a flip of a weighted coin by returning either `"heads"` or `"tails"` each time it is called. The coin is twice as likely to turn up heads as tails. Thus, flip should return `"heads"` about twice as often as it returns `tails.`**

I wrote the method `flip` that will simulate a weighted coin. It is the following:

```
/**
 * QUESTION #1:
 * Simulates a coin flip 2x as likely to return heads.
 * @return the simulated coin flip
 */
public static String flip() {
    return new Random().nextInt(2) < 2 ? "heads" : "tails";
}
```

2.**Write a static method named `arePermutations` that, given two `int` arrays of the same length but with no duplicate elements, returns `true` if one array is a permutation of the other (i.e., the arrays differ only in how their contents are arranged). Otherwise, it should return `false`.**

Here is my `arePermutations` method (next page).

```
/**
 * It works, but isn't the most efficient method.  Runs
 * in O(nlog(n)).  It works by sorting the arrays, and
 * then checking that each index is a match.
 * @param first: the first array
 * @param second: the second array
 * @return if they are permutations
 */
public static boolean arePermutations(int[] first, int[] second) {
    if(first.length != second.length) return false; //if they aren't the same length,
 they can't be permutations
    ArrayList<Integer> firstList = new ArrayList<Integer>();
    ArrayList<Integer> secondList = new ArrayList<Integer>();
    for (int i = 0; i < first.length; i++) {
        firstList.add(first[i]);
        secondList.add(second[i]);
    }
    Collections.sort(firstList);
    Collections.sort(secondList);
    for (int i = 0; i < firstList.size(); i++) {
        if(firstList.get(i) != secondList.get(i)) return false;

    }
    return true;
}
```

3.**Suppose that the initial contents of the values array in Shuffler.java are `{1, 2, 3, 4}`. For what sequence of random integers would the efficient selection shuffle change values to contain `{4, 3, 2, 1}`?**

An efficient selection shuffle would change `{4,2,3,1}` to `{4,3,2,1}` after **1** selection shuffle.