<u>Recitation Worksheet: Algorithm Analysis, Lower Bound Sorting, Counting/Radix Sorts</u>

1) In class we've looked at some sorts. To verify that our code has worked, we've written a utility function, isSorted, shown below:

```
public static boolean isSorted(int[] arr) {
     for (int i=0; i<arr.length-1; i++)
          if (arr[i] > arr[i+1])
               return false;
     return true;
}
```

In our particular use case, unless our sort was incorrect, the run time of this method ends up being O(n), when run in an array of length n because the for loop runs to completion when the input array is sorted.

Now consider the case of running this method on a randomly arranged array of n unique items, where each of the possible n! permutations is equally likely.

In this question, we'll set up the summation equal to the ***<u>average case</u>*** run-time of the method under these particular assumptions.

(a) What is the probability that the first two items are in order? What is the probability that the first two items are out of order? (Note that in this case, the method does one single comparison before returning.)

(b) What is the probability that the second item is larger than the first, but the third item is smaller than the second? (This corresponds to the situation where the method does 2 comparisons before returning a result in the negative.)

(c) In general, what is the probability that the first k-1 items are in sorted order but the k^{th} item is less than the $(k-1)^{th}$ item? (Hint: Consider the possible k! permutations of any of the first k chosen items and how many of them fit this mold, then divide that value by k!.)

(d) Using the information in part (c), set up a summation equal to the average case run-time of the isSorted method when run on a random array where we expect each permutation of items to be equally likely. You may leave one term (corresponding to sorted input) out of the sum.

You are given n k-sided fair dice, each labeled 1, 2, 3, …, k. You roll all of them. Then, separate out the ones that show k. Take the rest and roll them all again. Then, separate out the ones of these that show k, and roll the rest again. Repeat this process until you've separated all the dice out (ie, they all show k). The question we want to analyze is the number of times we expect to roll before completing the game.

2) One way to analyze this is to write a simulation, run it many times and take the average. The latter portion of this is fairly trivial, so for this question, you will simply write a single Java method that takes in n, the number of dice, and k, the number of sides on each dice, simulates this process, and returns the number of turns it took to complete a single simulation of the game. Fill in the method signature given below. Assume that you have access to a static class variable r, that is of type Random, which you can use to generate random integers.

```
public static Random r;

public static int numTurnsSim(int n, int k) {
    // Fill in code here.
}
```

3) Let the dice have k sides each and let T(n) equal the expected number of turns to complete the simulation described previous. A recurrence relation that T(n) satisfies is as follows:

$$T(0) = 0$$

$$T(n) = 1 + \sum_{i=0}^{n}[\binom{n}{i}\left(\frac{1}{k}\right)^{i}\left(\frac{k-1}{k}\right)^{n-i} T(n-i)]$$

In words, explain why this formula is correct. In your explanation, please explain what 1 represents, what the summation index i, represents and what T(n-i) represents. (Of course, explain what each part represents and why their interaction is the way that it is.) Note: One issue with this formula is that a T(n) term appears on the right hand side. But, if one were to solve this recurrence, we can easily take care of this problem by subtracting that term to the left hand side and factoring out T(n). Then, we would have a factor by which we could divide both sides of the resulting equation.

4) You are given a composite number n, but are asked to give proof that it is composite. You have a test that you can run such that, given a composite number, it proves that it is composite 50% of the time. Thus, to achieve your goal, you'll simply keep on repeating your test until you obtain the proof that the given number is composite. Fill out the chart below indicating the probability your algorithm will run the test various number of times.

Number of Times Test is Run	Probability
1	½
2	
3	
4	
k	

Using the chart above, write an infinite summation, in summation notation equaling the expect number of times your test will have to be run to prove that a given composite number n is composite.

Solve the summation on the following page. You should get a constant value as your answer. (Use the scratch page if you need more room to complete the summation.) Also, no need to use summation notation here. Feel free to write out the pattern of the sum so that it's easier to see your work.

5) What is the fewest number of comparisons necessary to sort an array of size 9? You may use the fact that 9! = 362880 and that 2^{16} = 65536.

6) Show each phase of radix sort on the following integers where we sort by digit:

Original List	Phase 1	Phase 2	Phase 3	Sorted
2987				1287
2381				1391
3967				2381
1391				2567
2567				2987
3394				3394
1287				3967