built on 2019/04/24 at 10:11:34 ***due:*** *thu may 2 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to help you dust off your programming skills (and for some of you, learn a bit of Java) and practice new concepts that we're covering in class this week.

Special Notes: Exercises are to be graded, but problems will not be graded. Moreover, you cannot use late tokens for HackerRank problems.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a1.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw1.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Collaboration Policy: To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

Non-Task: Java Environment

In this class, we will be using Java 1.8 (the specific minor release doesn't matter). Any distribution of Java will be fine. The development kit is called the JDK for Java Development Kit. For the most part, any flavors of JDK will be fine; we're only using the most basic features. Download Java SE (compatible) JDK if you're given choices to choose from.

A good IDE can help with Java programming a great deal. Many professional programmers are happy with Eclipse, IntelliJ IDEA, and NetBeans, all of which are free for academic use. You can't go wrong with any of them. Our in-class demos will be based on IntelliJ IDEA. If you're a command-line kind of person, editors such as VSCode, Sublime, Atom, Vim, and Emacs are good candidates for Java programming as well.

Exercise 1: Min and Max (2 points)

For this task, save your code in `MinMax.java`

Consider the following problem: given an array of n numbers, we want to find both the minimum and the maximum of these numbers. For such a problem, we often measure the cost in terms of the number of comparisons made—that is, if we compare any two numbers, that's one comparison.

As an example, the following algorithm requires $n-1$ comparisons:

```
// assume a.length > 0
int maxArray(int[] a) {
    int maxSoFar = a[0];
    for (int i=1;i<a.length;i++) {
        if (a[i] > maxSoFar)
            maxSoFar = a[i];
    }
    return maxSoFar;
}
```

This is because in an array a of length n, *only* $a[1], a[2], \ldots, a[n-1]$ are compared with our `maxSoFar` in the **`if`** statement. Notice that $a[0]$ is not involved in the **`if`** statement.

You can use this algorithm to the find the maximum value and an almost-identical algorithm to find the minimum value. However, you'll need $2n-2$ comparisons ($n-1$ for max and another $n-1$ for min). Your goal in this problem is to do better!

Your Task: Implement a function

```
public static double minMaxAverage(int[] numbers) {
  // your code goes here
  int myMin = ...;
  int myMax = ...;
  return (myMin + myMax)/2.0;
}
```

that takes in an array of integer numbers, finds the minimum and the maximum among these numbers, and returns the average of the minimum and the maximum (as the code above shows). For full credit, if input contains n numbers, your function must use ***fewer*** than $3n/2$ comparisons.

(Hint: Remember the highest-number problem from class? What happens after one round in the pairing algorithm?)

Exercise 2: Hello, Definition (4 points)

This task consists of small problems involving working directly with the definition of Big-O. *(Hint: One very simple solution for the first two problems below involves just taking limits.)*

(1) Show, using either definition, that $f(n) = n$ is $O(n \log n)$.

(2) In class, we saw that Big-O multiplies naturally. You will explore this more formally here. Prove the following statement mathematically (i.e. using proof techniques learned in Discrete Math):

> **Proposition:** If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then the product $d(n)e(n)$ is $O(f(n)g(n))$.

(3) There is a function `void fnE(int i, int num)` that runs in 1000·`i` steps, regardless of what `num` is. Consider the following code snippet:

```
void fnA(int S[]) {
    int n = S.length;
    for (int i=0;i<n;i++) {
        fnE(i, S[i]);
    }
}
```

What's the running time in Big-O of `fnA` as a function of n, which is the length of the array `S`. You should assume that it takes constant time to determine the length of an array.

(4) Show that $h(n) = 16n^2 + 11n^4 + 0.1n^5$ is *not* $O(n^4)$.

Exercise 3: How Long Does This Take? (2 points)

For each of the following functions, determine the running time in terms of Θ in the variable n. **Show your work.** We're more interested in the thought process than the final answer.

```
void programA(int n) {
    long prod = 1;
    for (int c=n;c>0;c=c/2)
        prod = prod * c;
}

void programB(int n) {
    long prod = 1;
    for (int c=1;c<n;c=c*3)
        prod = prod * c;
}
```

Exercise 4: Halving Sum (6 points)

For this task, save your code in `HalvingSum.java`

Our course staff has a signature process for summing up the values in a sequence (i.e., array). Let X be an input sequence consisting of n floating-point numbers. To make life easy, we'll assume n is a power of two—that is, $n = 2^k$ for some nonnegative integer k. To sum up these numbers, we use the following process, expressed in a Python-like language:

```
def hsum(X):  # assume len(X) is a power of two
   while len(X) > 1:
      (1) allocate Y as an array of length len(X)/2
      (2) fill in Y so that Y[i] = X[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2
      (3) X = Y
   return X[0]
```

This task has *three* parts:

Part I: (2 points) First, you will implement this summing algorithm as a function

`public static double hsum(double[] X)`.

The function will return the sum of the numbers in the input list. Note: we won't directly compare floating-point numbers. To check if x and y are equal, we'll test if `Math.abs(x - y)< 1e6`, i.e., whether $|x - y| < 10^{-6}$.

Part II: (2 points) Second, it is easy to see that the amount of work done in Steps (1)–(3) is a function of the length of X in that iteration. If $z =$ `X.length` at the start of an iteration, how much work is being done in that iteration as a function z (e.g., $10z^5 + z\log z$ or $k_1 z^2 + k_2 z$ for some $k_1, k_2 \in \mathbb{R}_+$)? Don't use Big-O; answer in terms of k_1 and k_2. Let's make some assumptions here. For some $k_1, k_2 \in \mathbb{R}_+$:

- Allocating an array of length z costs you $k_1 \cdot z$.
- Arithmetic operations, as well as reading a value from an array and writing a value to an array, can be done in k_2 per operation.

Part III: (2 points) Finally, you'll analyze the running time of the algorithm (remember to explain how you get the running time you claim). To help you get started, make a table of how `X.length` changes over time if we start with X of length, say, 64. How does this work in general? *(Hint: The geometric sum formula presented in class may come in handy.)*

Exercise 5: Random Permutations (2 points)

We often need to generate a random permutation of the numbers $1, 2, \ldots, n$. This is an array of length n where each number between 1 and n (inclusive) appears exactly once. For example, all possible permutations for $n = 3$ are shown below:

```
[1, 2, 3]          [2, 3, 1]
[1, 3, 2]          [3, 1, 2]
[2, 1, 3]          [3, 2, 1]
```

A random permutation for $n = 3$ is obtained by picking one of these arrays uniformly at random.

We'll learn what all these mean later on this term. For now, we're interested in understanding the performance characteristics of an algorithm that generates a random permutation. An implementation in Java is given below:

```
import java.util.Random;

public class RandomPerm {
  static void swp(int[] arr, int i, int j) {
    int tmp=arr[i]; arr[i]=arr[j]; arr[j]=tmp;
  }

  public static int[] mkPerm(int n) {
    int[] perm = new int[n];
    Random rng = new Random();
    for (int i=0;i<n;i++) perm[i] = i+1;
    for (int i=0;i<n;i++) {
      int t = rng.nextInt(i+1);
      if (i!=t) { swp(perm, t, i); }
    }
    return perm;
  }
}
```

In your write-up:

1. Analyze the running time of `mkPerm` in $\Theta(\cdot)$ as a function of n. You should make the following assumptions:
   - The statement `new int[n]` takes $\Theta(n)$ time.
   - The statements `new Random()` and `rng.nextInt(...)` take $O(1)$ time each.
   - When you call the swap function `swp(perm, t, i)`, this takes constant time per call.

2. Write a program to execute this algorithm. For each value of n, run it $T = 10$ or more times to obtain a good average. Complete the following table and make a plot (n vs. running time):

n	running time (average of T trials)
100,000	
200,000	
400,000	
800,000	
1,600,000	
3,200,000	
6,400,000	

To measure the time of a piece of code, you may find the following snippet useful:

```
long startTime = System.nanoTime();
// insert here what you need to time
long endTime = System.nanoTime();

System.out.println("It took " + (endTime - startTime)/1e6 + "
    milliseconds");
```

3. Does the graph/data support the analysis you did earlier? Discuss your findings.

Exercise 6: HackerRank Problems (4 points)

For this task, save your code in `hackerrank.txt`

You'll begin by creating an account on `hackerrank.com` if you don't have one already, so you can solve this set of problems. There are 2 problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website.

Important: You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

`https://www.hackerrank.com/muic-data-structures-t-318-assignment-1`

Problem 7: Inside Zeros (0 points)

For this task, save your code in `FacInnerZero.java`

Trailing zeros are zeros at the end of a number, in the right-most digits. The number 102400, for example, has a total of 3 zeros—1 inside and 2 trailing.

Remember that for $N > 0$, $N! = N \times (N-1)!$ and $0! = 1$. That is why $5! = 120$.

In this problem, you are to implement a function

```
public static int zeroInsideFac(int n)
```

that takes $n \geq 0$ and returns an `int` representing the number of zeros inside the value of $n!$. Your code only has to work for $0 \leq n \leq 1024$.

Here are some examples:

- `zeroInsideFac(5)` should return 0 because $5! = 120$ contains no inside zeros.
- `zeroInsideFac(16)` should return 1 because $16! = 20922789888000$ has *one* inside zero.
- `zeroInsideFac(37)` should return 4.

(Hint: The `int` *data type in Java can only store a number up to about* 2.1×10^9*. The* `long` *data type isn't big enough either. Learn about BigInteger. Also, you don't need recursion. You will want to look at* `BigInteger`*.)*