COMP 285: Analysis of Algorithms
North Carolina A&T State University
Dr. Allison Sullivan

Programming Assignment #1
Due: September 19, 2018 11:59pm

Programming Assignment #1

Programming assignments are to be done individually. You may discuss the problem and general concepts with other students, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. *If any code you submit is in violation of this policy, you will receive no credit for the entire assignment.*

This project has two goals:

- Ensure that you understand the structure of divide-and-conquer algorithms.
- Get a hands-on and empirical understanding of the practical benefits of designing more sophisticated algorithms, in particular, those that use the divide-and-conquer design technique.

We have provided Java code skeletons that you should fill in with the algorithmic logic for each of the three required solutions. You should not have to (and should not) implement any of the input processing and output generation. This has all been done for you.

Problem Statement. Thanks to your computer science education at NC A&T, you've graduated and landed a job at **Netflix**. Its your first week on the job, and you are ready to impress your boss. As a Netflix end-user, you've noticed that Netflix *occasionally* makes suggestions to you for movies that you have no interest in watching. This seems like something you could improve.

You decide to "crowd-source" the recommendation feature. You know Netflix maintains a database of preference lists for movies for a large set of users. You would like to examine this database in such a way that, given a user, you can determine what other user's taste is most similar. Say a client ranks a set of n movies; your system's ultimate goal is to find another user who has a similar ranking. An important step in this process is quantifying the *difference* between two such rankings. You figure this is something your boss can then hand off to the recommendation department and they can figure out how they want to leverage your implementation, i.e. **your concern is with measuring how similar two user's rankings are**, and someone else will figure out how to use that **measure** to make suggestions to similar users.

Consider comparing a new client's ranking to an existing client's ranking of the same set of n movies. A straight-forward approach would label the movies from 1 to n according to the new client's ranking, then order these labels according to the existing client's rankings. We could then examine the latter to see how many pairs of indices in the array are "out of order." Generically, consider being given a sequence of n numbers $a_1, \ldots, a_n$, where all a_i are distinct. We need to define a *measure* that quantifies how far this list is from being in ascending order.

To create our measure, we are going calculate the "edit distance" between our two rankings. An edit distance is a metric used in natural language processing to quantify how dissimilar two

strings are by counting the number of operations needed to transform one string into another. For our problem, we are going to count the number of instances where the following situation holds:

(a) $i < j$, and

(b) $a_i > a_j$

where i and j are indexes in an array. In other words, we increment our edit distance if our new ranking flipped the order in which we encounter two movies. To get a feel for this, consider the following sequence:

2, 4, 1, 3, 5

Our edit distance would be '3' for the following flipped movies: $(2, 1)$, $(4, 1)$, and $(4, 3)$. When a sequence is in complete agreement, the edit distance is 0; when a sequence is in complete disagreement, the edit distance is $\binom{n}{2}$. (**Note:** $\binom{n}{2}$ expresses a *permutation!* It is not a weird fraction.)

Part 1: Implement the Brute Force Algorithm [15 points]

Your initial thought to solve this is pretty straight-forward. You are to implement a brute force algorithm that solves the problem. You have been provided a skeleton file called `BruteForce.java`. It has a method named `measureEditDistance` which has an array called `rankings[]` as its parameter. The sequence of integers stored in `rankings[]` are initialized outside this class; *do not change the input blocks inside your* `measureEditDistance` *method.* The brute force algorithm works in the following way.

> Examine every pair of integers in the sequence and determine if the movies are flipped.
> If they are, increment the edit distance.

Your method should terminate by returning the edit distance. We have created a member variable for your `measureEditDistance` method called `editDistance` that you may use throughout the method (but you don't have to).

Part 2: A Divide and Conquer Algorithm [35 points]

Since you've taken an algorithms class, you pause before handing the program over to your boss at Netflix. You realize that you've just produced an iterative algorithm, and that means there might be a chance to *improve* on your run-time. In fact, you start to see that this problem lends itself well to a divide-and-conquer approach.

You decide to implement a new algorithm employing this design paradigm. The basic idea follows the strategy of MergeSort discussed in class. We define an index m and divide the list of rankings into two pieces $a_1, \ldots, a_m$ and $a_{m+1}, \ldots, a_n$. We recursively find the edit distance for these two halves. Then we increment the edit distance for situations that *cross* the two halves. In this latter situation, we increment the edit distance if a_i is in the first half, a_j is in the second half, and $a_i > a_j$. The easiest way to find this out is to actually sort the list as you go.

The pseudocode for the top-level procedure is:

```
1  measureEditDistance(A){
2     if (length(A) = 1) { return (0, A) }
3     B = A[1 ... m]
4     C = A[(m + 1) ... n]
5     r_B = measureEditDistance(B)
6     r_C = measureEditDistance(C)
7     r_A = measureCrossEditDistance(B,C)
8     return ((r_A + r_B + r_C), A)
9  }
```

The more complicated part is defining the pseudocode for MEASURECROSSEDITDISTANCE(B,C) It turns out you can do this with a fairly straightforward modification to the MERGE procedure we discussed in class. When in the merging loop, if the element selected as the smallest of the left and the right arrays happens to be in the right array, then you've identified flipped movies and should increment the edit distance. You have been provided a second skeleton file called `DivideAndConquer.java`, which also contains a method called `measureEditDistance`. Your method should terminate by returning the total number of pairs you explored that were out of order. We have created a member variable for your `measureEditDistance` method called `editDistance` that you may use throughout the method (but you don't have to).

Part 3: A Less Sensitive Counter [35 points]
Perhaps the exact number of flipped movies is not the best measure of similarity between two rankings of movie preferences. It is perhaps too sensitive. Let's define a *less sensitive edit distance* in which $i < j$ and $a_i > 2a_j$. Your task is to design and implement an algorithm to compute the new edit distance using this definition. (This implementation should obviously find *smaller* edit distances compared to the previous case.) You have been provided a second skeleton file called `LessSensitiveDivideAndConquer.java`, which contains a method called `measureEditDistance`. Your method should terminate by returning the total count of pairs you explored that were out of order. We have created a member variable for your `measureEditDistance` method called `editDistance` that you may use throughout the method (but you don't have to).

Part 4: Analyzing Your Algorithms [15 points]
First, for each of the algorithms you implemented above, determine the theoretical running time of the algorithm in terms of N (the number of ranked movies). Second, you will empirically measure the execution times of your various approaches. Class `Lab1.java` is provided to test your program, and it is also instrumented with timing instructions to measure the time your algorithm takes to run. Run each of your algorithms on a variety of input sizes, and record the time each takes to run. To change the input size, change the value of `n` in `Lab1.java`'s `main` method. (We advise you to change NOTHING else inside of `Lab1.java`.) Generate a plot (with your favorite plotting software) with input size as the X axis and running time as the Y axis. Create a file `readme.pdf` that contains a brief description of your algorithms and their running times (and anything you think we need to know about your implementation to grade it fairly) and this graph.

Instructions

- Download and import the code into your favorite environment. There are 4 `.java` files, but you only need to make substantial modifications to `BruteForce.java`, `DivideAndConquer.java`, and `LessSensitiveDivideAndConquer.java`. Carefully look through the code provided to you before you start. The set of provided files should compile and run successfully before you touch them at all.
- You must implement the method `measureEditDistance` in the three classes `BruteForce.java`, `DivideAndConquer.java`, and `LessSensitiveDivideAndConquer.java`. You may add methods to these three classes if you feel it necessary or useful.
- Class `Lab1.java` is provided to test your program. You may change the value of n to generate rankings of different sizes. Do NOT change anything else in `Lab1.java`.

What To Submit

You should submit a single file titled `LastName_FirstName_Lab1.tar.gz` or `LastName_FirstName_Lab1.zip` that contains the three java files with your solutions (i.e., `BruteForce.java`, `DivideAndConquer.java`, and `LessSensitiveDivideAndConquer.java`) and your `readme.pdf` file. Do NOT include anything else (e.g., we do NOT need the `Lab1.java` you used). Your solution must be submitted via Blackboard BEFORE 11:59 pm on September 19, 2018.