CS141

LONG PROGRAMMING ASSIGNMENT 3

DNA SAMPLING SORTER

OVERVIEW

This assignment will give you practice with arrays and producing an external output file. You are going to write a program that processes an input file of data for particular DNA sequencer test.

This particular DNA sequencer test is designed to take a sample of bacterial and generate a sequence of values for it, and then retest it at a time interval later to see if the DNA has or has not effectively changed.

YOUR INSTUCTIONS

Your assignment is to write a program that will give the user a brief introduction, then allow the user to type in the name of the file to be analyzed, the name of the data file to generate, and then process the data to match the output that is shown below.

Your goal is to match the output exactly as shown below, in both the data file structure and the program execution.

BACKGROUND

This DNA test measures the various parts of the sequence and assigns them a letter. While the letters could be anything from A to Z, the only letters that matter for this test are the letters {A,B,C,D} all other letters can be ignored completely.

A sample will be tested, given a length of time and then tested again. Each time the scientist will generate a line of data.

Here is one Example:

```
Example #1
AAAAABBBBBCCCCCDDDDD
AAEBCBAFBBCDCECDADDEFEEFFF
```

At first glance the sample looks significantly different after the second test. But if you look at the data, you will note that since we only care about A,B,C,D's that the second line does have the same number of A's , B's, C's and

D's. This means that while it looks different we would say that the sample is non-mutated. Note however that what is important is not the exact number, but the percentages. In the above case the percentages are 25% A's , 25% B's, 25% C's and 25% D's. *(Note that it is 25% of the letters we care about, not the total).* If another sample started with 10 A's, 10 B's, 10 C's, and 5 D's, and ended with 20/20/20/10. That would also be considered a non-mutation because while the numbers are different, the percentages are the same.

Contrast that with the data below.

```
Example #2
AAATAABTBBBBCCCCTCDDTDDD
AASAABBSBBCCSCCDSDDDEEEAEEFBFFFDDF
```

If we look at example #2. The initial percentages are [25, 25, 25, 25] but then in the second test they changed to [25, 25, 20, 30], which is a change and therefore counts as a mutation.

SAMPLE INPUT FILE

```
Example #1
AAAAABBBBBCCCCCDDDDD
AAEBCBAFBBCDCECDADDEFEEFFF
Example #2
AAATAABTBBBBCCCCTCDDTDDD
AASAABBSBBCCSCCDSDDDEEEAEEFBFFFDDF
```

SAMPLE OUTPUT FILE

```
Example #1
   Before:  5 - 5 - 5 - 5    After:   4 - 4 - 4 - 4
   Before: 25%/25%/25%/25%   After:  25%/25%/25%/25%
    MATCH

Example #2
   Before:  5 - 5 - 5 - 5    After:   5 - 5 - 4 - 6
   Before: 25%/25%/25%/25%   After:  25%/25%/20%/30%
    DIFFERENT
```

SAMPLE CODE EXECUTION

```
This program processes a file of DNA structures for various biological
samples to determine if a mutation of a particular nature has occurred.

input file name? The Data.txt
output file name? Results.txt
```

PROGRAM NOTES

You should round percentages to the nearest integer. You can use the Math.round method to do so, but you will have to cast the result to an int, as in:

```
int percent = (int) Math.round(percentage);
```

The output for this program will generate an output file. You do so by constructing an object of type PrintStream and writing to it in the same way you write to System.out (with print and println statements). See section 6.4 of the book for examples. It is a good idea to send your output to System.out while you are developing your program and send it to the output file only after you have thoroughly tested your program.

You can read the user's answers from the input file using a call on nextLine(). This will read an entire line of input and return it as a String.

Notice that the letters in the sample input file sometimes appear as uppercase letters and sometimes appear as lowercase letters. Your program must recognize them in either case.

The sample input and output files provide just a few simple examples of how this program works. I will be using a much more extensive file for testing your program.

I will once again be expecting you to use good programming style and to include useful comments throughout your program. I am not specifying how to decompose this problem into methods, but I will be grading on the quality of your decomposition. That means you will have to decide how to decompose the program into methods. You should keep in mind the ideas I have been stressing all quarter. You don't want to have redundant code. You don't want to have any one method be overly long. You want to break the problem down into logical subproblems so that someone reading your code can see the sequence of steps it is performing. You want main to be a concise summary of the program. You want to avoid method chaining by using parameters and return values. You should restrict yourself to the programming constructs included in chapters 1 through 7 of the textbook in solving this problem.