README

Compiling and running program:

- This program was written in Java 7

- The source code can be compiled from running the following command (although I have included the compiled executable in the zip file too):

```
javac Alpharank.java
```

- Once compiled, the executable can be run using the command:

```
java Alpharank <INPUT_STRING>
```

Description of Implementation:

- My algorithm is based on my observation of the following fact: *Suppose we have a list of all permutations of an input string in alphabetical order. Starting with the first word, we can systematically "skip" through multiple words in this list(without enumerating them) until we find the input string. If we count the number of strings that we skipped and add 1, we have our "rank" for the input word. This "rank" where it falls in an alphabetically sorted list of all words made up of the same set of letters(our answer).*

- Here are the details of my algorithm:

1. We create a HashMap and iterate through the input string. Each character found in the string is mapped to its number of occurrences.
   See method `buildIndex(String s)` in my source code, where this is implemented.
2. We create a variable called "rank" which is initially set to 1.
3. For each letter '**L**' in the input string, we iterate though the letters in the HashMap in alphabetical order until we find **L**. For each letter '**C**' we iterate through in HashMap which is NOT equal to **L**, we a) decrement the occurrence count of **C** in the HashMap, b) compute the number of distinct permutations of the remainder the the letters in the HashMap by computing the multinomial coefficient(from probability & combinatorics), c) add the result to the "rank" variable, and d) re-increment the character count of **C** in the HashMap.
   See the method `computeRank()` in my source code, where this algorithm is implemented.
4. The final value of the variable "rank" is our answer.

- My implementation relies on the computation of multinomial coefficients, which I implemented in the method `computeMultinomial(int topTerm, List<Integer> bottomTerms)`

- I used the Java class `java.math.BigInteger` because some computations involve factorials which are too large to fit in a 64-bit integer.

- My implementation of this problem does not involve factorials larger then 30! , thus I included a lookup table for factorials of all numbers between 0 and 30. This way these factorials can be retrieved in constant time complexity, which is much faster then actually computing them during runtime