Searching and Sorting

This assignment introduces how to search and sort data. In the case of this assignment, you'll be searching and sorting a large array of words to count how many times different words appear in the array. More specifically, we'll be working with an array containing all the words that appear in Mary Shelley's novel *Frankenstein*. So you'll be able to answer questions like: *Which of the words "Doctor", "Frankenstein", "Monster" or "Igor" appear more often in the novel?*

You will explore two different ways to count words in a piece of text, and learn how to calculate which one takes longer.

You will implement and time three things:

- A naive counting search: given an array, count how many times a word appears in it.
- A sorting function: given an array, return a sorted version of that array.
- A binary search: given a **sorted** array, find where a word **should** be, and use that knowledge to count the number of times it appears (faster than before)

The first task is **easy**, the second two are **more challenging**.

Running this code

This program takes one command line argument: the words to search for and count.

If you don't give it any words to search for, it defaults to these words: `"doctor"`, `"frankenstein"`, `"the"`, `"monster"`, `"igor"`, `"student"`, `"college"`, `"lightning"`, `"electricity"`, `"blood"`, and `"soul"`.

You may want to run it with fewer or different words, eg:

```
$ java SearchAndSort "frankenstein,doctor,igor,monster"
```

This will only count how often the words `"frankenstein"`, `"doctor"`, `"igor"`, and `"monster"` appear in the text.

Timing things in Java

How long did your code take to run?

In this assignment, we will see some search algorithms run faster than others. To see *how much faster*, we will record how long the code takes to run.

Here is an example of timing code:

```
// Look at the clock when we start
long t0 = (new Date()).getTime();
for (var i = 0; i < 100000; i++) {
    // Do something that takes time
}
// Look at the clock when we are finished
long t1 = (new Date()).getTime();
long elapsed = t1 - t0;
System.out.println("My code took " + elapsed + "milliseconds to
run.")
```

This is already set up for you in `main` and will output how long your code took to run. Your job is to implement and call your functions, and make sure that they are working correctly. If you look in the starter code, you'll see that we're running your two different search and count methods 100 times, so as to get a good average value for how long it takes. We're only do the sort once, as sorting the array of 75289 words already takes awhile.

1. Counting words (naive approach)

- Write and implement a method `countWordsInUnsorted`
  - parameters: an array of words (`String[]`) and a query word (`String`) which is the word we are looking for
  - return: the number of times that word appears
- Call that method in `main`
- **Checking if you are right:** `"frankenstein"` should appear 26 times.

Implementing the function `countWordsInUnsorted`

Define a `countWordsInUnsorted` method that takes an unsorted array of strings and a query string, and returns an `int` of how many times that query occurs in the array. Make it a `static` method (like `main`) so that you can call it directly without having to create an instance of the class SearchAndSort first.

Create a counter variable to record how many times we have seen a word. Create a loop that iterates over every word in the array. Each time a word matches the query word, increment the counter.

Note: How can we test if two strings are equal? You have two possibilities in Java: `stringA == stringB` and `stringA.equals(stringB)`. These behave differently in Java! The first tests if they are the *same object*. The second tests if they have the same sequences of characters. We need to use the `equals` method, since we care about whether two strings have the same characters in them, not that they are the exact same object.

Calling the function in `main`

In the `main` method, within the `TODO #1` comment block, you'll want to replace the `0` in the line:

```
int count = 0;
```

with the call to `countWordsInUnsorted`. The variable `i` in the `for` loop is iterating over the query words, so you can use this to pass each query word into `countWordsInUnsorted` within the loop.

Checking if you are right

Once you've completed the two tasks above, you can run your code. Verify that `"frankenstein"` appears 26 times. You'll also see how fast this code takes to execute (averaged over 100 searches). This amount of time will vary depending on your computer and what other programs are running on it at the same time.

2. Sorting an array of words with Merge Sort

- Implement `mergeSort` (Do **not** use Java's built-in Array.sort for this assignment)
- Call your new method in `main`

- **Checking if you are right:** the words (after `SORTED` in the output) will be in alphabetical order.

Implementing `mergeSort`

Implement a `mergeSort` method that sorts an array of strings. The signature for this method should be:

```
public static void mergeSort(String[] arrayToSort, String[]
tempArray, int first, int last)
```

The first argument is the `String[]` to sort, the second argument is an empty temporary array that should be the same size as the array to sort, the third argument is the starting index for the portion of the array you want to sort, and the fourth argument is the ending index for the portion of the array you want to sort.

Note that the version of mergeSort we'll be implementing sorts `arrayToSort` *in place*. This means that you pass an unsorted array in to `mergeSort` and after the call that same array is now sorted.

The book provides a good description of mergeSort, on pages 527-533. We'll also be talking about how mergeSort works in class.

Note that mergeSort involves comparing to values to see which is larger. With numbers you can just use `<` or `>` to compare them. But for Strings we'll need another way to compare. Fortunately, Strings have another method defined on them for comparing called `compareTo`. You can read more about this method more here, but long story short, if you have two Strings `s1` and `s2`, then:

- `s1.compareTo(s2) == 0` if s1 and s2 contain the same string.
- `s1.compareTo(s2) < 0` if s1 would be alphabetically ordered before s2.
- `s1.compareTo(s2) > 0` if s1 would be alphabetically ordered after s2.

Calling `mergeSort` in `main`

To call `mergeSort` in `main`, you first need to create a temporary string array that is the same length as `allWords`. We want to sort the whole array, so the third argument should be the index of the first word in allWords and the fourth argument should be the index of the last word in allWords. The four arguments to call mergeSort with are then:

- The array we're sorting, which is `allWords`.
- Our new temporary array we've created that's the same length as `allWords`.
- The index of the first word in allWords (hint: what's always the index of the very first element of an array?).
- The index of the last word in allWords (hint: if you know the length of an array, you can easily compute the index of the last element).

Checking you are right

Did this sort the array of words? The program prints every 500th word. Do they look sorted?

3. Counting words (with Binary Search) and timing it

- Implement `public static int binarySearch(String[] sortedWords, String query, int startIndex, int endIndex)`
- Implement `public static int getSmallestIndex(String[] words, String query, int startIndex, int endIndex)`
- Implement `public static int getLargestIndex(String[] words, String query, int startIndex, int endIndex)`
- Call `getSmallestIndex` and `getLargestIndex` in `main` to get the smallest and largest indices at which the word you're looking for appears in the sorted array. Use these two values to compute how many times that word appears.
- **Checking if you are right:** you will get the same values as in the first search section, but **much faster.**

Implementing `binarySearch`

The arguments to binarySearch are:

- The sorted array of words to search in.
- The word to search for.
- The index in the array at which to start searching.
- The index in the array at which to stop searching.

There's a nice description of how to implement the binary search algorithm for integers on pages 168-172 of the book

Binary search returns the array index where it found the word. If the word only appears once in the array, then this index will be where it occurs. But if the word appears multiple times in the sorted array (so all the instances of the word will be next to each other in the array), then this index will be to one of the words in the middle of the group. The binary search algorithm doesn’t guarantee that this will be the first element in the group or the last element in the group. So we need to implement some other methods to do this.

Implementing `getSmallestIndex`

The method `getSmallestIndex` will be a recursive method that uses the `binarySearch` method to find the smallest index for which a word is found in the array. The outline for this method is:

- Use `binarySearch` to find an index to the word. If the index `binarySearch` returns is -1, then the word wasn’t found and `getSmallestIndex` should just return -1. *This is the base case*.
- If `binarySearch` did find the word, then recursively call `getSmallestIndex` on the portion of the array *before* where the word was found. This is from index 0 up to (but not including) the index where the word was found. If this returns -1 then we know we already had the smallest index, otherwise the recursive call to `getSmallestIndex` found the smallest index and we should return that. *This is the recursive case*.

Implementing `getLargestIndex`

The method `getLargestIndex` will be a recursive method that uses the `binarySearch` method to find the largest index for which a word is found in the array. The outline for this method is very similar to the outline above for `getSmallestIndex`, except that the recursive call should search the portion of the array starting *after* where `binarySearch` has found the word.

Using `getSmallestIndex` and `getLargestIndex` in `main` to count words

Since the array that you’re working with has been sorted by this point, all the same words appear next to each other (e.g. all 26 appearances of the word `"frankenstein"` are next to each other in the array). So if you’ve found the smallest and largest index for a word, then you can just subtract the two indices to count the words! *But it’s not quite as simple as just subtracting.* To figure out what it should be, consider the case where a word appears only once (so the first and last index are the same) and the case where the word doesn’t appear at all.

Checking if you are right

Check that you're getting the same answers as the naive approach that iterates through the whole array. Also, look to see how much faster this approach is to the naive approach!

Example arguments and output

```
$ java SearchAndSort "student,college,frankenstein"

SEARCH AND SORT

Searching and counting the words student,college,frankenstein

NAIVE SEARCH:
student:3
college:3
frankenstein:26
49 ms for 300 searches, 0.163333 ms per search

SORTING:
2393 ms to sort 75289 words

SORTED (every 500 word):  a a above after all am and and and and and and another are as astonished
authority be been beneath bosom but buy called chair closed concussion cooking covered darkness deemed
despair dire doing earnest endure even exert failure feeble fiend fondly for friend from gloom guilt had
hardly have he hellish hide his hope hurried i i i i i if in in in inspiring is it journey labours
lesson little lullaby many me me miles monotonous mould my my my my nearly no not obtain of of of of of
on one or own paused place preceded prosperity rays remained restrained s saw seems she shrieks snowy
sometimes spoke strong support teeth that that the the the the the the the the their there thirst those
timid to to to to transcendent undertaking urged views was was waves were when which who will with woman
wound you your

BINARY SEARCH:
student:3
college:3
frankenstein:26
2 ms for 300 searches, 0.006667 ms per search
```

Of course the actual timing for running the searches and sort will vary on your computer, depending on how fast your computer is and how many other processes are running on it.

Turning the code in

- Create a directory with the following name: `<student ID>_assignment1` where you replace `<student ID>` with your actual student ID. For example, if your student ID is 1234567, then the directory name is `1234567_assignment1`.

- Put a copy of your edited `SearchAndSort.java` file in the directory.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named `<student ID>_assignment1.zip` (with <student ID> replaced with your real ID of course).
- Upload the zip file through the page for Assignment 1 in canvas (https://canvas.ucsc.edu/courses/12730/assignments/38334).

Helpful tips:

Visualizations

- Visual animations of different sorting types: https://visualgo.net/bn/sorting?slide=1
- Merge sort as a dance: https://www.youtube.com/watch?v=XaqR3G_NVoo
- An explanation of Binary Search: https://www.youtube.com/watch?v=j5uXyPJ0Pew

Code things

- Using Java's printf: https://docs.oracle.com/javase/tutorial/java/data/numberformat.html
- Did you use `myString == "foo"`, or `myString.equals("foo")` In Java, these don't mean the same thing! (https://stackoverflow.com/questions/513832/how-do-i-compare-strings-in-java)

Use lots of test output to help make sure that your Binary Search and Merge sort are operating correctly at every level of recursion. For example, here is a debug statement that I made for myself to test the Binary Search.

```
System.out.println(query + " pivot:" + sortedWords[pivot] + " " + startIndex + "-"
+ endIndex + " " + sortedWords[startIndex] + "-" + sortedWords[endIndex-1]);
```

```
SEARCH: soul soul pivot:me 0-100 a-you soul pivot:task 50-100 me-you soul pivot:of
50-75 me-such soul pivot:play 62-75 of-such soul pivot:sister 68-75 play-such soul
pivot:success 71-75 sister-such soul pivot:streets 71-73 sister-streets soul not
found
```

Be sure to comment out extra debug statements when you turn in your work!