CSC220 Lab06

Sorting and Timing

The goal of this week's assignment is:

1. Practice sorting
2. Practice measuring the runtime of a program
3. Practice working with generics
4. Continue learning the significance of special cases!
5. Learning how to write tests to check your implementation!

Things you must do:

1. There are many details in this assignment. Make sure you read the whole thing carefully before writing any code and closely follow this instruction.
2. You must complete your assignment **individually**.
3. Always remember Java is case sensitive.
4. Your file names, class names, and package name must match exactly as they are specified here.

Things you must not do:

1. You must not change the file names, class names, package names.
2. You must not change the signature of any of these methods (name, parameters, …).

For this lab (and assignment), you are asked to construct a program that has the capability to determine if two words are anagrams and to find the largest group of anagrams in a list of words. Two words are anagrams if they contain the same letters with the same frequency. For example, **alert** and **later** are anagrams. There are many websites devoted to listing anagrams, such as this one (http://www.english-for-students.com/Complete-List-of-Anagrams.html) if you are unclear about the concept yet.

We have learned about how sorting algorithms work and how to measure the efficiency of a program by its time complexity. Now that we know how to implement some popular sorting algorithms, we are eager to use sorting to solve these two problems. We will implement two sorting algorithms for the task: **insertion sort** and **merge sort.** While we are at it, we will also measure the time our sorting implementations take so that we can *predict* the performance for much larger inputs, say of size 100K or even 1M!

To check if two words are anagrams, simply sort the characters in each word. If the sorted versions are the same, the words are anagrams of each other. Words with the same letters with the same frequency, but different cases are still anagrams. E.g., **Begin** and **being** are anagrams. Hint: Java's built-in String method `toLowerCase` will come in handy when sorting and determining if words are anagrams, but **you must still output them in their original case**. We will concern ourselves only with word anagrams and not anagrams that are phrases, which may contain whitespace and punctuation.

Part 0

- Create a new Java project and call it **Lab06**.
- Create a package inside your project and call it **lab06**.
- Download the Lab06-Assignment06 ZIP folder from Blackboard (google drive link). Copy and paste the following files into your new lab06 package:
    - **AnagramUtil.java**
    - **InsertionSort.java**
    - **MergeSort.java**
    - **SortedString.java**
    - **Tester.java**

Part 1 - `SortedString` implementation

The SortedString class is responsible for storing the sorted version of a String along with the (unsorted) original. We begin with finishing its implementation.

- `public SortedString(String unsorted)`
    - This constructor initializes the two member variables unsorted and sorted in the class based on the input string.
    - The member variable `sorted` can be determined by converting the input parameter `unsorted` to a char array using `toLowerCase().toCharArray()` and then converting back to a string after sorting. Remember that we do not care about upper/lower case for the sorted string.
    - To sort the characters in this array, we will use `Arrays.sort(chars)` **(don't worry, in the next steps we will implement and use our own :-)**
    - Be sure to set both member variables `unsorted` and `sorted`

- `public int compareTo(SortedString other)`
    - This method compares two SortedStrings based on the *sorted* string.
    - Note that this method will be used later for making comparisons in our sorting algorithms.

- Before moving to Part 2, **test** your SortedString implementation in `Tester.java`. You should check:
    - Instantiating a SortedString object from a string, e.g. "zebra"

- Checking the sorted and unsorted versions. Is the sorted version sorted correctly? Does the unsorted version contain the original string? Print the strings and make sure they are what you expect.
- Comparing two SortedStrings. For instance, “joy” is bigger than “ski”. Why? What about “below” and “Elbow”?

Part 2 - `AnagramUtil` implementation: phase 1

- `public static boolean areAnagrams(`
  `SortedString str1, SortedString str2)`
    - This method returns true if the two input SortedStrings are anagrams of each other, otherwise returns false.
    - Remember: to check if two words are anagrams, simply compare the sorted versions of each word. If the sorted versions are the same, the words are anagrams of each other.

- Before moving to Part 3, **test** your AnagramUtil implementation in `Tester.java`. For instance, if you pass in a string, say your first name, and your first name again but with the characters reversed, you should get true! But, if you pass in your first name and your last name, you should get false!
- You should also check:
    - empty string vs another string
    - exactly same string
    - shuffled strings
    - two different strings
    - make sure the answers are case insensitive

Part 3 - `InsertionSort` implementation

Now we will implement our first sorting algorithm using insertion sort.

- `public E[] sort(E[] array)`
    - This method takes a generic input array and returns a sorted array using an insertion sort. Since this method is generic, we cannot use > or < for

comparing values. Instead, we must make use of Comparable, a la compareTo().

    - Note: If InsertionSort is instantiated with type SortedString, i.e., InsertionSort<SortedString>, the sort() method will invoke the compareTo() we just wrote in part 1. Do you see why?

- This method should not modify the input array. Instead it should make a copy of the array using array.clone(), and sort the clone.
- Take care of special cases! What if the array is of size 1?

- Before moving to Part 4, **test** your InsertionSort implementation in `Tester.java`. You should check the following with InsertionSort<Integer> :
    - a list with one element
    - a list with two elements
    - a sorted list of numbers
    - a random list of numbers
- For the task at hand, we are more interested in an InsertionSort that handles types of SortedString, i.e. InsertionSort<SortedString>. To test this, you may find it helpful to use the convenience function `toSortedString` from `SortedString`. For instance:
    - if myArray is a String array, the following will return a SortedString[] representation: `SortedString.toSortedString(myArray)` Handy!
- Now, if we have InsertionSort<SortedString> and we pass the following array of strings to its sort function: {“joy”, “ski”, “fed”, cat”}. Here is how the output should look like {“cat”, “fed”, “ski”, “joy”}. Make sure you understand this example.

Part 4 - Predicting runtime for `InsertionSort`

For the last part of lab, we will measure the performance of InsertionSort using the time formula you learned in class. This formula is quite useful as it allows us to make *predictions* about how our function will do for much larger inputs, without actually running it.

- `public double O(int n)`

  - This method returns the order O() of the implementation.
  - Consult the lecture slides if you have doubts
  - Note: for a runtime of O(n^2), **it is better to use Math.pow(n, 2) rather than n * n.**
- `public void fit(E[] array)`
  - This method calculates the constant c using the given input array. Time measurements are measured in microseconds.
  - Remember the formula has form `time = c * O()`, where time is the time it takes to run the insertion sort for the input array and O() is the big-Oh given from the function above.
  - To test the insertion sort algorithm, make a call to sort().
  - To time your program, you can use System.nanoTime(). **Remember to convert nanoseconds to microseconds!**
- `public double predict(int n)`
  - This method predicts the running time of an insertion sort for an input size n. The estimated time return is in unit microseconds.
  - Note: this method will **NOT** run the insertion sort with the given n. Instead, you **must** use the calculated constant c from `fit()` to make a prediction for the n. You may assume that fit() has been called previously before predict().

To test your timing functions, try running fit() with a modestly sized array, say with 10 elements. Then try running predict() for an input size much larger, say 100K or even 1M. How much time would it take to sort such a large list? Would it finish by the end of lab? Think about whether your answers make sense given what you know about insertion sort. If you have doubts, consult your lab instructors.

Part 5 - Testing

Unlike previous labs/assignments you are not given any tests as a starting point. You must create your own tests to examine all and every method you implemented in the previous part. Several testing suggestions were made throughout the lab. Make sure you check the functionality of your methods very carefully.

Don't forget: The lab/assignment is due Tuesday night @ 11:59pm!