A4 Sorting!

Announcements and Clarifications

None

Brief Description

Your task in this assignment to is implement several different types of sorts and to experiment with sorting several different data sets.

Goals and Requirements

- Implement a list ADT
- Implement bubble sort
- Implement insertion sort
- Implement mergesort
- Implement quicksort
- Implement binary search

Description

Your first goal is to implement a list ADT, you may implement this data structure however you choose, but consider the running time of various operations that are necessary for sorting.

You should implement two versions of bubble sort, one that uses the list ADT you implemented and one that uses Java's ArrayList class. The reminder of your sort should all use the same data structure to store the list of things to sort. You must also implements two versions of quick sort: the first is the heuristic where you choose the first element as the pivot and the second should choose the median value as the pivot using the median finding algorithm. You also need to implement a mode that reads in the data to sort but does not perform any operations.

Your sorts should always order the elements in increasing lexicographic order.

Implementation

All of the user input will be as command line arguments:

```
java Sorts mode_number input_file output_file amount_of_data ...
```

The *mode_number* indicates which operation will be performed:

1. Bubble sort using ArrayList
2. Bubble sort using your list class
3. Insertion sort
4. Merge sort
5. Quick sort picking first element as pivot
6. Quick sort using the median method to choose pivot
7. Sort using the Arrays.sort() method
8. No-op, reads in data but does no sorting
9. Binary search

For all modes *input_file* indicates the file to read the data from. For the binary search mode, assume that the input file contains data already in sorted order.

For all modes *output_file* indicates the name of the file to output the results to. For the sorts, the only output to the file will be a list of the data in sorted order, one piece of data per line. For the binary search mode, the only output should be the pivot element in each step of the algorithm on its own line, if item is found the last line should be that item otherwise the last line should be "NOT FOUND".

For all modes *amount_of_data* indicates the number tokens to read from the data file. Use Scanner.next() to read in the tokens.

For the binary search mode there will be another command line argument that specifics the token to search for. Your implementation of the binary search should be recursive.

No-op mode will read in the data from *input_file*, place it in the ArrayList and then output the data to *output_file* in the one token per line format.

If there are any I/O problems with the input file or the output file indicate that error on standard error and exit the program gracefully. If there are fewer tokens in the input file than are requested to be read in, indicate an error and exit the program.

Data sets

We have several data sets for you to use. Consider all of the data tokens as Strings when sorting. Use the standard Scanner.next() to tokenize the data with the default separators.

- Shakespeare's Hamlet (source) 31999 tokens
- First 1 million digits of pi (source) 1000000 tokens
- First 100k numbers increasing order - 100000 tokens
- First 100k numbers decreasing order - 100000 tokens

The data sets are human readable, look through them before you start coding.

Questions to Answer

At the end of your README.txt file include answers to the following questions:

1. For each of the data sets run all modes except binary search using *amount_of_data* in increments of 1500 up to 15000 measuring the running time using the method described in assignment 3, the hints section in this assignment describes a way to automate this process. You may choose to measure with larger data sets, but be aware that for some of the algorithms this will take a long time. For each of the data sets plot the results and submit an electronic copy of the plot (.pdf, .ps or .xls). Analyze and comment on the plots. Include estimates of the what the experiment tells about the asymptotic running times of the sorts for each of the data sets.
2. Why did we ask you to implement a mode that reads in the file and outputs it but does not sort the data? (update your plots to account for this)
3. Often the known structure of data can help you improve sort performance. For the first two data sets name two distinct ways to improve sort performance based on what you know about the data.
4. Comment on the difference between the results in the third and fourth data sets.
5. Compare the performance of bubble sort on your list and ArrayList, were there differences? What do you think accounts for these differences?
6. Compare the two versions of quick sort on all test sets.

7. Which algorithm do you think Arrays.sort() uses to sort data? Why?
8. If you had used LinkedList instead of ArrayList how would your results have changed?

Commenting and Style

- Your program should be written in a style that makes it easy to read and understand.
- At the beginning of each .java file you should include a description of the class and how it interacts with the other parts of your program.
- You are not required to use javadoc style comments.
- If your code is doing something complex or non-standard please comment that portion heavily.

Handin

Please hand all necessary files into your handin directory in a subdirectory named **Sorts**. Your application class should be called **Sorts.java**. There should be exactly one class declared within each .java file. If your program does anything strange (bugs), awesome(extra features) or has a non-intuitive interface please include a file called README.txt which explain them. If there are bugs in your program but you do not describe them in your README you will lose more credit than if you had described them.

Hints

- Develop incrementally.
- Remember to design your list data structure so that operations important to sorting are fast: getting items from an index, swapping items, comparing items...
- Test your sorts on small sets of your own data so that you verify correctness by hand.
- The Linux command **diff** allows you to determine if two files differ. To use:

  ```
  % diff file1 file2
  ```

- Automating your measurements - It's a pain to run each of the test cases individually and record the results. There's a way to make it easier. First we need to use a more versatile **time** command. There is another time command that allows you to choose the format of the output:

  ```
  %/usr/bin/time -f "%U" java Sorts pi.txt pi_out.txt 15000
  ```

  This will change it so that only the value we are looking for (the time elapsed while running our program) is displayed. This is actually the first number that the **time** command we discussed in assignment 3 produces, however, it is more representative of tests which are shorter in duration. It does not match the wall clock time it takes for the test to run however.

  We can make things even easier by writing something called a shell script, it will allow us to run a batch of programs in sequence. We provide a script to automate the tests you must run, you can find it here.

  To use the script first copy it to your working directory, then run the command:

  ```
  %chmod u+x test.sh
  ```

  This will allows the shell to execute the our script (we only have to do this once). To run the script use:

  ```
  %test.sh mode_number input_file
  ```

This will produce the timing for running your program with the specified mode on the specified input file for 0, 1500, 3000, ... , 15000 tokens. Open the script in an editor. Each line of the script is run in order as though you had typed it into the command line. The $1 and $2 are like variables that represent the command line arguments to the script. In our case $1 represents the mode number and $2 represents the name of the input file. The first line of the script tells the operating system what language our script is written in.

Try writing another shell script to run all the tests for one data set (first make a new file and use the **chmod** command on that file). There is one additional command that might help you. It is called **echo**, it works exactly like println in Java. To print *Hello world* to the console you would use **echo "Hello world"** in your script.

- Running any of the sorts on any of the data sets should take no more than two minutes on a lab machine (in fact it should take much less time), if it takes longer you're probably doing something very inefficiently, look at your data structures and analyze the running time of your methods.