CS 0445 Spring 2022 Assignment 4

Online: Monday, March 28, 2022
Due: All files (see details below) submitted in a single .zip file to the proper directory in the submission site by **11:59PM on Friday, April 15, 2022.**
Late Due Date: 11:59PM on Monday, April 18, 2022

Purpose and Goal: In lecture we discussed QuickSort and several different variations for choosing the pivot during the partition process. We also discussed the idea of stopping the recursion with a "larger" base case array size in order to avoid the extra overhead of recursive calls for very small arrays. In this assignment you will empirically examine some Quicksort variations and will also examine the effect of changing the base case array size on the performance of the sorts. We hope to see which base cases work best to optimize our various sorts and which version of QuickSort performs the best overall. For purposes of comparison we would also like to consider the MergeSort algorithm. We'd like to compare these algorithms via a solid object-oriented approach and in a formulaic way.

General Idea: You will test 5 sorting algorithms:

1) Simple Quicksort with A[last] as the pivot (as discussed in lecture and seen in file Quick.java)
2) Median of 3 Quicksort (as discussed in lecture and seen in file TextMergeQuick.java)
3) Random Pivot Quicksort (as discussed in lecture)
4) Median of 5 Quicksort (see details below)
5) MergeSort (as discussed in lecture and seen in file TextMergeQuick.java)

Note that the first 4 algorithms are really the same recursive algorithm (QuickSort) but with different partition methods. Thus, a generic version of QuickSort can be written that accepts different versions of partition(), enabling all 4 versions above to be done via a single class (*this is really cool!*)

Let us define the interface Partitionable as follows:

```
public interface Partitionable<T extends Comparable<? super T>>
{
      public int partition(T[] a, int first, int last);
}
```

This method will partition array **a** into { items <= the pivot }, { the pivot } and { items >= the pivot } as discussed in lecture. It will return the index where the pivot is located in the partitioned array. Note that as specified this interface (naturally) is NOT stating HOW the data will be partitioned – that is left up to the implementation.

Any new Java class can implement Partitionable as long as it implements the method partition() as specified above. You will implement the Partitionable interface 4 times:

1) Class SimplePivot in file SimplePivot.java will implement the partition algorithm in the same way as specified in file Quick.java. This is using the rightmost element in the current array as the pivot. However, instead of a static method partition() will now be an instance method within class SimplePivot.
2) Class MedOfThree in file MedOfThree.java will implement the method partition() in the same way as specified in file TextMergeQuick.java. This is the median of three algorithm as discussed in lecture. However, instead of a static method partition() will now be an instance method within class MedOfThree.
3) Class RandomPivot in file RandomPivot.java will implement random pivot partition() method (as discussed in lecture) as an instance method with class RandomPivot. The code for this method was not provided so you will need to write it yourself. Carefully consider how this partition version will be correctly implemented.
4) Class MedOfFive in file MedOfFive.java will implement the method partition() by choosing the pivot as the median of five elements: A[first], A[fmid], A[mid], A[lmid] and A[last]. Index fmid should be midway between first and mid and index lmid should be midway between mid and last.

The idea of this algorithm is similar to that of median of three, but 5 items rather than 3 are being used to determine the pivot. Look at the median of three code and then carefully think about how the MedOfFive partition() method would be correctly implemented.

Note that in all of the partition() implementations, after selecting the pivot index you should move the pivot to a convenient location so that the rest of the partition can be done easily. See the median of three code in TextMergeQuick.java to see how this can be done (although it may not be exactly like this in your other versions).

Let us now define interface Sorter as follows:

```
public interface Sorter<T extends Comparable<? super T>>
{
      public void sort(T[] a, int size);
      public void setMin(int minSize);
}
```

The sort() method will be the actual method that does the sorting, and the setMin() method will set the minimum size for recursive calls (after which the sort will revert to insertionSort()). For example, if a call setMin(10) is made then the Sorter should make recursive calls for arrays down to a size of 10, but for sizes 9 or smaller the sort should be completed via insertionSort() of the remaining items. See TextMergeQuick.java to see how this is done. Note that different implementations of Sorter<T> will have different lower bounds for the setMin() method. In particular, SimplePivot and RandomPivot could have a min size all the way down to 1, but MedOfThree requires a min size no smaller than 3 and MedOfFive requires a min size no smaller than 5. Think about why this is so.

Now class QuickSort can be defined in a generic way utilizing a Partitionable<T> object as its data:

```
public class QuickSort<T extends Comparable<? super T>>
                                                  implements Sorter<T>
{
      private Partitionable<T> partAlgo;
      private int MIN_SIZE;  // min size to recurse, use InsertionSort
                             // for smaller sizes to complete sort
      public QuickSort(Partitionable<T> part)
      {
            partAlgo = part;
            MIN_SIZE = 5;
      }
      // remaining code in QuickSort class not shown
      // You must complete this class - in particular the methods
      // sort() and setMin()  You will use partAlgo for partition
      // within the sort() method.
}
```

Similarly, class MergeSort can be defined to also implement Sorter<T>

```
public class MergeSort<T extends Comparable<? super T>>
                                                  implements Sorter<T>
{
      private int MIN_SIZE; // min size to recurse, use InsertionSort
                            // for smaller sizes to complete sort
      public MergeSort()
      {
            MIN_SIZE = 5;
      }
      // remaining code in MergeSort class not shown
      // You must complete this class - in particular the methods
```

```
        // sort() and setMin().
    }
```

With the QuickSort and MergeSort classes set up in this way, a variable of type Sorter<T> can be used to store any object that implements the Sorter<T> interface, and can thus be used to sort data in any of the ways described above. For example, we could have:

```
    Sorter<Integer> mySort;
    mySort = new QuickSort<Integer>(new SimplePivot<Integer>());
    mySort.setMin(10);
```

Now mySort is ready to sort an array of Integer using QuickSort with the SimplePivot partition and a base case of any size < 10. Thus, if we have an array of Integer, A, that we would like to sort, we could call

```
    mySort.sort(A);
```

These types can also be used to systematically test / compare various sorting algorithms in a straightforward, automatic fashion.

Details:
You must write the following classes:

SimplePivot in file SimplePivot.java
MedOfThree in file MedOfThree.java
RandomPivot in file RandomPivot.java
MedOfFive in file MedOfFive.java
QuickSort in file QuickSort.java
MergeSort in file MergeSort.java
Assig4 in file Assig4.java

The first 6 files should be implemented as specified above such that they work with the program SortTest.java. Given an input of 25 (on the command line), the output should match that shown in file SortTest-out.txt. All of the sorting algorithms should produce the same sorted data – make sure to double-check your output, especially with RandomPivot and MedOfFive, since you are writing these partition methods (more or less) from scratch.

The last file (Assig4) must test your sorting algorithms in a systematic way. Define a **configuration** for your program to be a given algorithm, with a given array size and a given minimum size for the recursion (set via the setMin() method). Your program will test various configurations with the various algorithms in an attempt to determine the best configuration for each algorithm and the best algorithm overall.

In particular, an execution of Assig4 should do the following:

- Accept the following **command line arguments**:
  1) Size of the array to sort (integer), **`N`**
  2) Number of runs to try for each algorithm / configuration (integer), **`Nruns`**. Multiple runs for a given configuration will increase the accuracy of your results. See more details on the runs for each configuration below.
- Create an ArrayList of Sorter<Integer> and place an instance of each sorting algorithm into this ArrayList (thus you will have 5 Sorter<Integer> objects in your ArrayList – 1 for each QuickSort version and one for MergeSort. **For help on this step, see SortTest.java.**
- Execute loops that will systematically test all of your algorithms with the given size array of initially random data. The timing will be very similar to what you have done in Recitation Exercise 8, using System.nanoTime() to determine the time elapsed. More details on the structure of your main program loops will be given below. However, another interesting /

important issue should first be discussed. Due to the very large array sizes that will be used in your program, **you will need to rethink the way your random data is generated**. In particular, because the arrays will contain Integer objects rather than int values, if we generate new random data for each run of each algorithm we will be generating an extremely large number of objects, which will consume a lot of memory in the heap of the computer. For example, consider the following code:

```
for (int i = 0; i < A.length; i++)
{
      A[i] = Integer.valueOf(R.nextInt());
}
```

where A is an array of Integer and R is a Random object in Java. If A.length is very large and if we are doing this many times (multiple runs for each of our algorithms) we will clearly generate a very high number of Integer objects throughout our program execution. This will likely lead to the need for garbage collection during the program execution, perhaps more than once. Garbage collection is a compute-intensive process and will thus greatly affect the run-time of the algorithms, skewing the results. Thus, we would like to design our program in such a way as to limit / reduce the number of Integer objects that must be generated. With some thought this can be done with the following approach:

- In an array of size N, initially generate an array of Integer objects in sequence, from 1 to N. Clearly this initial configuration will contain sorted data.
- Prior to **testing each configuration**, make sure your array is in this same initial state and seed your random number generator (some Random object) using the setSeed() method.
- Prior to **each run** for a given configuration, shuffle the data in your array using your Random object. This will involve moving the data around randomly within the array but it will NOT involve creating any new objects (so you should also not use an extra array for this process – just shuffle the data in the single existing array). The idea here is similar to that of your shuffle() method from Assignment 1. By randomizing the data in this way you are able to get different random configurations without generating new Integer objects, thereby limiting the amount of required memory. To get your overall timing result from multiple runs you simply add the times for the runs together and divide by the number of runs – this will give the average time per run. Make sure for each run to time ONLY the sort itself – don't time shuffling the array items prior to the sort.
- **Note 1:** Do NOT reseed the random number generator between runs in a given configuration. We want the different runs for the same configuration to have different random data.
- **Note 2:** DO reseed the random number generator (with the same seed) when switching to a different configuration (ex: a new algorithm or a new min size for the current algorithm). This will allow all of your configurations to be tested on the same, random data, which will make the results more accurate.

Overall Main Program Approach:

- Your main program should iterate over all of the algorithms. **For each algorithm** you should do the following:
  - Set min size = 5 using the setMin() method
  - Time `Nruns` runs for this configuration, in the fashion described above, and store the average time per run
  - Double min size and repeat up to (and including) a min size of 160. Thus, min size will be 5, 10, 20, ... , 160 for each of your algorithms
- During the process described above, keep track of the following:
  - Best overall configuration (algorithm + min size)
  - Worst overall configuration (algorithm + min size)
  - For each algorithm:
    - Best min size

        - Worst min size

- After all of the configurations have run, output your results (to standard output) in a nicely formatted way. These results should include all of the information specified above. For an example of how your output should look, see file A4-1.txt. This is the output with an array of size 1000000 and 5 runs per configuration.
- Run your program for each of the array sizes, N = 1000000, 2000000, 4000000, 8000000, 16000000. For each execution use the same Nruns value of 5.
- Save the output of each of your program executions by redirecting the standard output to a file. Use the file names A4-X.txt where X = 1, 2, 4, 8, 16. In other words the X in each file name will be the non-zero leading digits of the array size used for that execution. For example, for the first array size (N = 1000000) and with 5 runs per configuration your execution would be as shown below:

  ```
  java Assig4 1000000 5 > A4-1.txt
  ```

  Redirecting standard output during a program execution is a useful way to have all of the "println" statements in your program send output to a file without having to explicitly create / use File objects in your program. It is done (as shown above) using the ">" sign followed by the name of the file to which the output will be sent.

- Run your program for each of the array sizes shown, saving the output in the files as specified above. Thus, you should produce and submit files: A4-1.txt, A4-2.txt, A4-4.txt, A4-8.txt and A4-16.txt.

Results:
After you have finished all of your runs, **write a brief summary / discussion of your results**. Based on all of your results for all your configurations, which algorithm / configuration do you think is best overall and why? Are your results consistent as the array sizes increase or do they change? Did your empirical time increases match with our asymptotic run-times discussed in lecture? Did any of your results surprise you? Support the assertions in your paper by referring to your results. Include a table of your results (for at least one of the array sizes) in the paper so that the reader can see them without having to look at all of your output files. Your write-up should be 1-2 pages (double-spaced, counting your table) and it should be submitted as a separate document (ex: a Word or .pdf document), and must be included with your other submission files.

Submission: There are a LOT of files to submit in your .zip submission for this project, so make sure you have everything. In particular, you need to submit:

Java files provided for you:
   Partitionable.java, Sorter.java, SortTest.java

Java files that you wrote:
   SimplePivot.java, MedOfThree.java, RandomPivot.java, MedOfFive.java, QuickSort.java, MergeSort.java and Assig4.java

Output text files
   Five files as specified above, one for each array size.

Writeup document:
   Word or .pdf file

Assignment Information Sheet
   As always

Additional Requirements, Hints and Help:

- For help with generating random integers, see the Random class in the Java API and specifically the nextInt() method. Also refer to your shuffle() method in Assignment 1. SortTest.java will also be helpful in this regard.
- SortTest.java will also be helpful in showing you how to set up your collection of Sorter<T> objects and of filling them with data. Read over that file carefully, noting all of the comments.
- Recitation Exercise 8 will also be helpful with this assignment – read over that solution for help with the timing procedure for your main program.
- Thinking about your Assig4 main program, note that **you need 3 nested loops to do all of your configurations and runs**. You need a loop to iterate through the algorithms, a loop to iterate through the minimum recurse values for each algorithm and a loop to iterate through the runs for each configuration.
- To make your results more accurate, **do not run anything else on your machine while you are doing your runs.** Don't worry about system processes that are running – just make sure you don't run any other applications.
- To make your results consistent, **do all of your runs on the same machine under the same (if possible) circumstances.**
- Due to fluctuations in your computer's background processes during execution of this program, you may see different results if you run the program more than once even with identical configurations. This is especially true for the smaller array sizes. **This is ok and in fact likely to happen.** Your results may also not exactly agree with mine. This is also ok as long as your results are reasonable.
- Your program may take a while to execute – especially for the largest array sizes. Make sure you test it thoroughly before doing your runs so that you are confident that it does not have any infinite loops.
- Be sure to time only the actual sorting procedure – do not time loading the data into the array or any I/O (especially not I/O – this is very slow and will skew the timing greatly).

Extra Credit:
(Either of these options, if done well, can earn the full 10 extra credit points, but no more than 10 points total will be awarded).

- Write a 2nd version of your main program that attempts to precisely (somewhat) determine the best min size to use for your algorithms. In the assignment you try some different values and output which was best, but you don't refine your choices or attempt to zero in on an actual optimal value for min size. Think about how you could do this effectively / efficiently. Note: Starting min size at 5 and incrementing by 1, testing every possible value, is NOT an effective way of doing this. You should think about how to do it without doing too much work.
- Add DualPivot Quicksort as another algorithm and compare to the others. To learn more about DualPivot Quicksort and to see an implementation, see the paper below:
  https://people.cs.pitt.edu/~ramirez/cs445/assigs/assig4/DualPivotQuickSort.pdf
  Note, however, that you will need to do some work in converting it into the format required for this assignment (i.e. implementing Sorter<T> -- don't try to use Partitionable<T> with DualPivot QuickSort since it does not partition in the same way as the other QuickSort algorithms).