Computer Science 331 — Winter 2016
Assignment #4 Solutions

Instructions

This assignment concerns lecture material that has been introduced in this course on or before Monday, March 28.

Please **read the entire assignment** before you begin work on any part of it.

This assignment is due by 11:59 pm on Wednesday, April 13.

There are a total of 100 marks available, plus an additional 20 bonus marks.

Questions

The purpose of this assignment is to compare the running times of various sorting algorithms, including the version of quicksort implemented in Java.

1. **(10 marks)** Implement the insertion sort algorithm. Include a function with signature

   ```
   public static void insertionSort(int [] A)
   ```

   in a Java source file `Sorting.java` that contains your implementation. The input array `A` should be rearranged so that its elements are in ascending order.

2. **(20 marks)** Implement the heap sort algorithm. Include a function with signature

   ```
   public static void heapSort(int [] A)
   ```

   in a Java source file `Sorting.java` that contains your implementation. The input array `A` should be rearranged so that its elements are in ascending order.

3. **(20 marks)** Implement the quicksort algorithm using deterministic partitioning as described in class (last element is used as the pivot). Include a function with signature

   ```
   public static void quicksort(int [] A)
   ```

   in the file `Sorting.java` that contains your implementation. The input array `A` should be rearranged so that its elements are in ascending order.

4. **(10 marks)** Two common improvements to quicksort are the following.

- Due to the recursive nature of quicksort, it is not the most efficient choice for sorting small arrays. In particular, a lot of time is wasted on recursive calls for very small subarrays that could be sorted much faster using one of the classical sorting algorithms.

  One common strategy to alleviate this problem is to make use of a threshold or cutoff value that indicates when quicksort should continue recursively as follows:

  - If the size of the input array is larger than the threshold value then the input array should be used to form a pair of smaller arrays that are recursively sorted as usual.
  - If the size of the input array is smaller than the threshold then the algorithm terminates without sorting the small array.
  - After completion of the recursive algorithm, insertion sort is called to complete the sorting process.

- As mentioned in class, the performance of quicksort depends heavily on the choice of pivot element. One common strategy is to choose the pivot to be the median of the first element in the array, the last element, and the element in the middle. These three elements are then reordered so that the smallest of the three elements is at the beginning of the array, the largest of the three is at the end, and the pivot is at the second-last position in the array, allowing the partition algorithm to start scanning the array at the second element (as opposed to the first) and finish at the second last element (as opposed to the last).

Implement a modified version of quicksort that uses the improved pivot selection strategy **as well as** the early recursion termination strategy from the previous questions. Include a function with signature

```
public static void quicksortImproved(int [] A)
```

in the file `Sorting.java` that contains your implementation. The input array `A` should be overwritten with the same elements in sorted order.

Notice that for the pivot selection strategy to work, it is necessary that the subarray being sorted have at least three elements. Thus, the cutoff value used to signify termination of the recursion should be at least three. Also, you should endeavour to choose the threshold value in such a way that your implementation is as fast as possible.

Solution: A bit of research online reveals that threshold values between 10 and 20 tend to work well. One could also do experiments using JRat or some other profiler to try to find an optimal value corresponding to your implementation, eg., using this approach revealed that a threshold of 128 tended to work well for the model solution on the web page.

5. **(15 marks)** Write a main program called `A5Q5.java` that tests your four sorting algorithms and Java's built-in `sort` function on **randomly-generated** arrays of various lengths. Using the VisualVM profiler (see the course web page for information on using VisualVM), determine the **total runtime** required to sort 1000 random arrays of lengths

$$128, 1024, 16384, 65536 \ .$$

Note that you may have to add the flag `-Xss32M` to the `java` commands for running the tests in order to allocate extra stack space when working with the larger array sizes. You may also

need to add the flag `-J-Xss32M` to the `visualvm` command for the same reason. Present your data in a table. Are the results what you would expect? Justify your answer.

Solution: Here are sample results obtained using the model solution available on the course web page. Times are listed in milliseconds. Although the raw data will vary depending on factors such as the computer used, the relative performance of the various algorithms should be similar to your findings.

	Average Runtimes				
Array Size	Insertion Sort	Heap Sort	Quicksort	Improved Quicksort	Java sort
128	5.95	15.00	13.80	8.98	6.76
1024	122.00	87.10	102.00	38.40	42.80
16384	25453.00	1388.00	1477.00	751.00	724.00
65536	402285.00	6225.00	6880.00	3378.00	3326.00

Based on the analysis presented in class, insertion sort should perform the worst of the five on random arrays for sufficiently large arrays because it requires time $O(n^2)$ whereas heap sort has worst-case time $O(n \log n)$ and and the quicksort variants both have expected time $O(n \log n)$ on random arrays. For the small array sizes, insertion sort should be faster than at least deterministic quicksort. If care is taken in the heap sort implementation, for example, by implementing the entire algorithm in one function, its performance should be not too much worse than the improved quicksort algorithm. Java's highly-optimized quicksort implementation should most likely be the fastest of all. It uses a more advanced pivot selection and partitioning strategy — details can be found in

> Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)

This paper is readily available on-line.

6. **(15 marks)** Write a main program called `A5Q6.java` that tests your four sorting algorithms and Java's built-in `sort` function on **sorted** arrays of various lengths. Using the VisualVM profiler (see the course web page for information on using VisualVM), determine the **total runtime** runtimes required to sort 1000 already-sorted arrays of lengths

$$128, 1024, 16384 \ .$$

Note that you may have to add the flag `-Xss32M` to the `java` commands for running the tests in order to allocate extra stack space when working with the larger array sizes. You may also need to add the flag `-J-Xss32M` to the `visualvm` command for the same reason. Present your data in a table. Are the results what you would expect? Justify your answer.

Solution: Here are sample results obtained using the model solution available on the course web page. Times are given in milliseconds. Although the raw data will vary depending on factors such as the computer used, the relative performance of the various algorithms should be similar to your findings.

	Average Runtimes				
Array Size	Insertion Sort	Heap Sort	Quicksort	Improved Quicksort	Java sort
128	1.30	10.00	26.80	2.20	1.87
1024	2.55	62.10	209.00	7.58	2.44
16384	7.50	962.00	31024.00	92.60	6.75

Based on the analysis presented in class, we would expect insertion sort to perform the best, because on sorted arrays it's runtime complexity is $O(n)$. Deterministic quicksort, which always selects the pivot as the last element in the array, should perform the worst, because sorted arrays represent the worst-case inputs for this algorithms resulting in a running time of $O(n^2)$. Heap sort has worst-case time $O(n \log n)$, and should perform reasonably well. Improved quicksort and Java's sort function, which have expected time $O(n \log n)$ for sorted arrays due to improved pivot selection strategies should also perform well.

Your submission for this assignment should include

- Your Java source code file named `Sorting.java` containing implementations of the required sorting algorithms.
- Your source code for the testing programs in Questions 5 and 6.
- A file containing answers to the written questions.

You are not required to submit information on your testing strategy for this assignment, however, **10 marks** will be allocated to the documentation of your source code. As usual, be sure to document your class thoroughly using `javadoc` tags and assertions as appropriate.

Bonus Questions

The following bonus questions should only be attempted after the previous questions have been completed satisfactorily. While solutions to these questions are not required for full credit on this assignment, correct solutions can result in a grade of more than 100 %.

7. **(20 marks)** Implement another version of quicksort that has *worst-case* runtime $O(n \log n)$. Describe the modifications to quicksort you use to achieve this running time, and present average, minimum, and maximum runtimes for the same tests in Questions 5 and 6. Are the results what you would expect? Justify your answer.