CSDS 233 Introduction to Data Structures
Assignment 5
Due date: Fri Nov 25 by 11:59 EST
TAs: Rohan Singh (rxs1182@case.edu) and Rithvik Kasarla(rxk654@case.edu)

Written Assignment (40 points)

1) Consider the following bucket sort algorithm, the pseudocode is given below:
SpecialBucketSort(arr):

```
        n = arr.length
        arrBucket[0 . . n −1] be an empty array of lists
        for i = 0 to n −1
        // hash function determines which bucket the element goes into
        insert arr[i] into list arrBucket[hash(arr[i])]
        for i = 0 to n −1
        sort list arrBucket[i] using insertion sort
        concatenate the lists arrBucket[0], arrBucket[1], . . . , arrBucket[n −1] together in order
```

   a) What is the worst-case and best-case running time of this sorting algorithm and provide the scenarios in which each of these happen. [5 pts]
   b) By replacing insertion sort with a more efficient merge sort or heap sort in line 9 of the pseudocode, bucket sort can achieve a worst-case running time of $O(nlogn)$ instead of $O(n^2)$. Argue why switching insertion sort to a more "efficient" sort is not a good idea. [5pts]
   c) If you have to sort a huge list that can't fit into memory. Would you use bucket sort or quick sort? Why?[5pts]

2) Since the worst-case running time of quick sort is $O(n^2)$, we want to improve the algorithm by setting a depth limit of the partition tree. Starting with the classic recursive quick sort algorithm, once it reaches the depth limit, the existing partitions are sorted by heap sort or merge sort. Explain why this would improve the classic quick sort algorithm. What would be a reasonable depth limit? [5pts]

3) Another upgraded version of quicksort is running the classic recursive quick sort algorithm, but stops when the subarrays have fewer than k elements. At this point, insertion sort is called once on the entire array, which finishes the sorting. Show that this sorting algorithm has an O(nk + nlog(n/k)) running time and explain why this is better than the classic quick sort algorithm. [10pts]

4) Suppose you have an array of N elements containing only two distinct keys, true and false. Give an O(N) algorithm to rearrange the list so that all false elements precede the true elements. You may use only constant extra storage space. [5pts]

5) What is the running time of quicksort (with the middle element of the input array used as the pivot), and why for:
   a) Sorted input
   b) Reverse-sorted input

   [5pts]

Programming Assignment and Benchmarking (60 points)

In this programming assignment you will implement and benchmark basic sorting algorithms.

1. **Sorting Algorithms (40 points):**
   In a file called *Sort.java* implement the following static methods:
   - **void insertionSort(int[] arr):** Takes an input array of integers. Uses the insertion sort algorithm to sort the input array in descending order.
   - **void bubbleSort(int[] arr):** Takes an input array of integers. Uses the bubble sort algorithm to sort the input array in descending order, the best-case runtime of this algorithm should be O(n).
   - **void shellSort(int[] arr):** Takes an input array of integers. Uses the shell sort algorithm using Hibbard's sequence to sort the input array in descending order.
   - **void quickSort(int[] arr):** Takes an input array of integers. Uses the quick sort algorithm to sort the input array in descending order.
   - **void mergeSort(int[] arr):** Takes an input array of integers. Uses the merge sort algorithm to sort the input array into descending order.
   - **void upgradedQuickSort(int[] input, int d, int k):** Takes an input array of integers. Uses the upgraded quick sort version from written problem 2 and 3 to sort the input array into descending order, where we switch to merge sort when it reaches the depth limit d and switch to insertion sort when the subarrays have fewer than k elements.
   - **int[] generateRandomArray(int n):** Takes an integer input. Returns an array of random integers of size n.
   - **void readCommands(String filepath):** Takes a String input corresponding to the filepath of a .txt file. Reads the commands specified in the file and then prints the corresponding result of the commands. (See attached example)

   You will be graded on compilation, correctness, comments, implementation and encapsulation.

2. **Benchmarking (20 points):**
   Time the sorting algorithms you implemented above as well as Java's in-built sorting algorithm using random inputs, sorted and reverse-sorted of sizes n = 10, 20, 50, 100, 200, 500, 1000, 2000, and 5000 each.
   You can use Java's System.nanoTime() to time your methods, you may print out the results and then use the output for benchmarking. Put your results in a table for easier comparison between the sorting algorithms (you could also include a plot).
   These "experiments" should be carried out in the main method of a separate Java class named *Demo.java*, please submit this along with your benchmarking pdf to attain any credit for this part, this is required in order to prove that you carried out your own experiments.

   After your "experiments" answer the following questions in detail:
   - What system did you run your tests on? Describe the relevant specifications. What factors might have affected the timing results?
   - Which method(s) is (are) faster for large n? How does each method scale with n for random input? Show your results either in a table or with a plot.
   - Is your answer to the previous question consistent with the theory? Explain.
   - Is insertion sort ever preferable? Your answer should include examples with benchmarks.
   - Are there input cases where one method performs poorly? Again, show examples with benchmarks.
   - Make a table to show for each of the input sizes (n = 10,20 …) and input type (random, sorted, reverse-sorted) which sorting algorithm is optimal and how much time they take.

Submission Instructions:

- Please submit your written assignment and benchmarking file as separate PDFs.
- Please submit your programming assignment source code in a zip file (with only the .java files) along with your *Demo* class.
- For the programming assignments please use the same class and method names as specified in the assignment.
- Please remove all package statements from your code.
- Failure to follow these instructions will result in a deduction of points from your score.

Tips and Suggestions:

- For sorting algorithms, using the debugger tools that most IDEs provide, will really help you find problems if you hit a dead-end ;)
- Use JUnit tests to check the correctness of your code and sorting algorithms, this is completely optional and just for you to check your work, you will NOT be penalized for not having JUnit test cases.
- Make use of helper methods to make the coding process easier.
- Meet up with the TAs Rohan Singh or Rithvik Kasarla for any questions that you may have regarding any parts of the homework.
- Try finishing the programming assignment at least 2 nights before it is due, because the benchmarking part, while not "difficult", does take some time to finish.