CEng301-Design and Analysis of Algorithms

Homework 4

Due date: January 03th

In this homework, you are going to run some algorithms successively on random input to observe and report their behavior. The algorithms that you are going to implement changes according to your student ID and your name's first letter.

If your first name starts with a **vowel** and your student id is **even**:	If your first name starts with a **vowel** and your student id is **odd**:	If your first name starts with a **consonant** and your student id is **even**:	If your first name starts with a **consonant** and your student id is **odd**:
1. Recursive Fibonacci function 2. Bubble sort 3. Insertion sort	1. Quick sort 2. Radix sort 3. Brute-force sorting	1. Merge sort 2. Bucket sort 3. Recursive Fibonacci numbers	1. Heap sort 2. Counting sort 3. Brute-force sorting

For each algorithm, create random input and save the completion times. Then you will submit a report showing **graphs** of completion times. In the innermost loops of each function add **Thread.sleep(10)** to create an artificial delay of 10 milliseconds at each iteration. Add this delay **only once** in one of the **innermost** loops of the corresponding function.

1. **Recursive Fibonacci:** Generate the first 40 Fibonacci numbers recursively and exhaustively. Save completion time of each Fibonacci number in milliseconds in a text file.
2. **Bubble Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Apply Bubble Sort to each array in ascending order. Save the completion time for each array in milliseconds in a text file.
3. **Insertion Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Apply Insertion Sort to each array in descending order. Save the completion time for each array in milliseconds in a text file.
4. **Quick Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Apply Quick Sort to each array in descending order. Save the completion time for each array in milliseconds in a text file.
5. **Brue-force Sorting:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Apply Brute-force sorting to each array in ascending order. Save the completion time for each array in milliseconds in a text file.
6. **Radix Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Full them with random integer numbers with exactly 8 digits. Apply Radix sort to each array in ascending order. Save the completion time for each array in milliseconds in a text file.
7. **Merge Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Apply Merge Sort to each array in ascending order. Save the completion time for each array in milliseconds in a text file.
8. **Bucket Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random double numbers between 0 and 1 (not including 0 and 1). Apply Bucket Sort to each array in ascending order. Save the completion time for each array in milliseconds in a text file.

9. **Heap Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers between 5000 and 1 million. Construct a Max Heap and then apply Heap Sort to each array. Apply Heap Sort to each array in descending order. Save the completion time for each array in milliseconds in a text file. Note that completion time for an array is the sum of time required to construct the heap (*heapify()*) and sorting itself (output the root and *heapify()* until the array is empty).
10. **Counting Sort:** Create arrays of size 5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000. Fill them with random integer numbers respectively between 1 and 5, 1 and 10, 1 and 30, 1 and 50, 1 and 100, 1 and 200, 1 and 500, 1 and 700, 1 and 850 and 1 and 1000. Apply Counting Sort to each array in ascending order. Save the completion time for each array in milliseconds in a text file.

You can use the following strategy to get the completion time for each array.

```
For each n in {5, 10, 30, 50, 100, 200, 500, 700, 850 and 1000}
        long startTime = System.currentTimeMillis()
        mySortingFunction(n) // This function has Thread.sleep(10) in it.
        long endTime = System.currentTimeMillis()
        Files.writeln(file, n + " " + (endTime - startTime) + " milliseconds")
Next n


For each n in {1…40}
        long startTime = System.currentTimeMillis()
        myFibonacciFunction(n) // This function has Thread.sleep(10) in it.
        long endTime = System.currentTimeMillis()
        Files.writeln(file, n + " " + (endTime - startTime) + " milliseconds")
Next n
```

Note: Do not be surprised if $O(n^2)$ algorithms take few hours to complete.

What to Submit: You will create a single zip (or rar) file with the following deliverables:

- Your source code for 3 functions (depending on your ID and first name) and 3 text files with milliseconds.
- Draw 3 separate graphs for each function (you can use Excel or MatLab for this). Create a pdf file of all graphs and put it into the same zipped file to submit. Your graph axes must look like this: