Chae-Ryeong(Chloe) Yeo 2458600Y

How to run my code:

Run testSortingAlgorithms.java
However, due to the pathological input array, it won't run from int500k and larger files as it gives stack over flow error when running the quicksort using the median of 3 partitioning scheme with the pathological input array, as the runtime is too long.
So, **to run and find out the running times of all four quicksort algorithms for larger files as well, comment out just the:**
/ part 3 - pathological input array sorted via quicksort medianOf3 */**
part and the code below, all of them in testSortingAlgorithms.java (and the part to comment out is also shown in the green comment), and run testSortingAlgorithms.java again.

Part 2 - Compare all four versions of the algorithm you implemented in Part 1 (try experimenting with different cutoff values in 1b). Also compare with the running times of InsertionSort and MergeSort. Report and explain your findings.

For each sorting algorithms using each given input file (except for when using pathological input - this is commented out to give the following results, as this isn't in Part 1 but in Part 3), the running time in nanoseconds(ns) are:

	int1000.txt	**int20k.txt**	**int500k.txt**	**intBig.txt**	**dutch.txt**
Standard QuickSort	389800 ns	1191700 ns	30467200 ns	67425800 ns	410260700 ns
QuickSort Using InsertionSort (cutoff value (k) = 30)	1824000 ns	61525300 ns	5453561000 ns	83022257400 ns	11972197200 ns
QuickSort Using InsertionSort (cutoff value (k) = 5)	3700 ns	15500 ns	748200 ns	3760800 ns	410700 ns
QuickSort Using Median-Of-Three Partitioning Scheme	146400 ns	1457800 ns	34395300 ns	131106900 ns	388644600 ns

3-Way-QuickSort	492600 ns	3869400 ns	35823300 ns	128308200 ns	37628100 ns
Insertion Sort	586200 ns	25123200 ns	13730300200 ns	97580188100 ns	24847562900 ns
Merge Sort	866300 ns	3013100 ns	64350300 ns	207052500 ns	111460900 ns

When comparing all four variations of quicksort with insertion sort and merge sort, quicksort using insertion sort with a small cutoff value, of 5, closer to zero has the shortest runtime for all file sizes. For smaller text files like int1000 and int20k, quicksort using a higher cutoff value of 30 has the longest runtime, on the other hand for larger files like int500k, intBit and dutch, insertion sort has the longest runtime.

Comparing 3 way quicksort and quick sort using median of 3 partitioning scheme: 3 way quicksort has a longer runtime for the 3 smaller files - int1000, int20k and int500k, but has a shorter runtime, than quicksort that uses median of 3 partition, for the 2 larger files - intBig and dutch.

Comparing 3 way quicksort and standard quicksort: standard quicksort has a shorter runtime for all files except dutch.txt, on the other hand 3 way quicksort has a shorter runtime for dutch.txt.

Comparing 4 versions of the quicksort algorithm when considering cutoff value=30 for quicksort using insertion sort: with a relatively large cutoff value, quicksort using insertion sort has the longest runtime out of the four variations of quicksort for all files.

Comparing 4 versions of the quicksort algorithm when considering cutoff value=5 for quicksort using insertion sort: on the other hand, with a relatively small cutoff value that is closer to zero, quicksort that uses insertion sort has the shorted runtime out of the four, for all files.

Looking at insertion sort and merge sort, merge sort only runs faster for the smallest file, int1000, and for the rest of the larger files, insertion sort runs faster.

Part 3 - Explain in the report how your algorithm operates.

To generate the pathological input array, i.e. the worst case input array, for quicksort algorithm that uses median of three partition, we generate an input array where the **second smallest element in the array is at the middle of the array**, i.e. the value of the middle element in the

array is the second smallest in the array, **for each iteration of the array when sort function gets called recursively on the input array** when the generated array is passed into the quicksort algorithm as an input parameter.

How the algorithm works is that first we take an input parameter n, that is the length of the pathological input array that is to be generated. If n, i.e. the length of the array we want to generate, is odd, then we set the last element's value equal to n-1, and decrement n by 1. We set the index of the middle element of the array, that is n/2, so that we can divide the array into the first half of the array and the second half of the array, and assign values of the elements accordingly depending on the index of the element in the array. Finally, we return the array that is to be used as input for quicksort that uses the median of 3 partitioning scheme.

For int1000.txt, quicksort algorithm using median of 3 partitioning scheme that takes in the array with given numbers in its order in the text file, takes a running time of 139000 ns, whereas the quicksort algorithm using median of 3 partitioning scheme that takes in the pathological input array generated using our algorithm above takes a running time of 1074700 ns. We can also see for int20k.txt that the former takes 1511700 ns to run whereas the latter takes 48740200 ns to run. From this, we can indeed see that the quicksort algorithm with the median of 3 partition takes much longer to run when using the worst case input array. For larger files however, including int500k, dutch and intBig, we are given a stackoverflow error when sorting running the quicksort algorithm using median of 3 partition with the pathological input array, as the runtime becomes too large.