Comparing Types of QuickSort with Sorts Seen So Far
Lab Assignment #5

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
September 27, 2017

1. Problem Specification

The purpose of this lab was to first implement quickSort and then test it with the same values as the previous labs were tested with. This allowed a direct comparison to be made between mergeSort, insertionSort, heapSort, and quickSort as documented below. Unlike prior examples, this lab also tested the sorts on a list that was specifically already sorted, one which was sorted in reverse, and one that was completely randomized, all of which contained the same 1,024 values in them.

2. Program Design

The steps taken to develop the program were:

a) Copy over existing code for print statements and I/O from the mergeSort program and change function calls to “quickSort” instead.

b) Develop the code for the quickSort and partition methods from the pseudocode provided in class. Test and see that the sort is working properly on lists of size 100 and 1,000.

c) Develop the median of 3 version of quickSort and the median3 method and test it with the same list of 1,024 values, once already sorted, once sorted in reverse, and once randomized to compare their runtimes. Finally made a general “swap” method to further simplify code up to this point.

d) Copy over all of the print statement text generated from previous assignments and this one to show the runtimes of standard quickSort and median of 3 quickSort on values from 100 to 500,000 along with the three comparisons mentioned before as well as previous sorts.

The following methods were used in the main method:

a) readInNums (Scanner derp, int size) : receives a Scanner object, locally referred to as “derp” as well as a value for the size of the intended read in array to accommodate each input text file.

b) printArray(int [] list) : prints any given array of integers sent to it.

c) swap(int[] A, int i , int j) : swaps the values at the two positions listed in the array A.

d) quickSort(int[] A, int p, int r) : Calls partition on the middle value between boundaries p and r defining the sublist of list A and then recursively calling itself on the left portion down to the smallest sublist before recursively calling for the second portion all so long as p is less than r.

e) partition(int[] A, int p, int r) : Swaps all values iteratively with the last value of A if they are less than it and then swaps the value at index p with the last value before returning its position.

f) quickSortM3(int[] A, int p, int r) : Same as quickSort, but under the same condition, will first select a median value by calling median3 and then swapping it with the last value of the list, r.

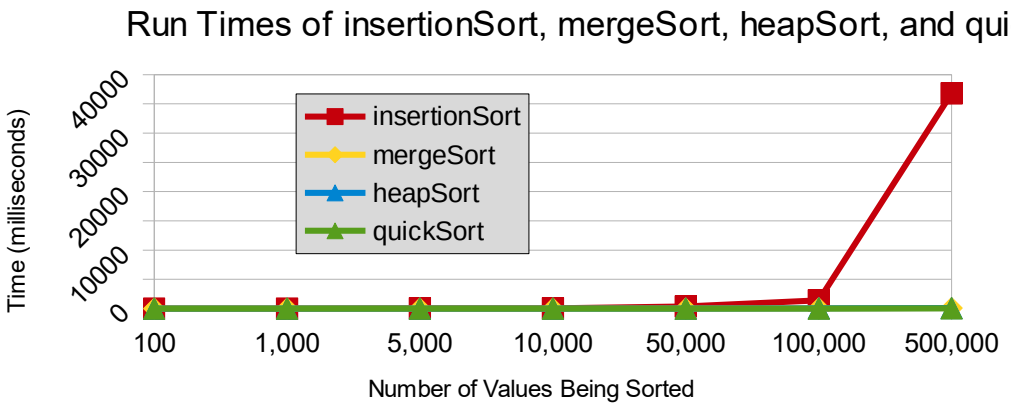
g) median3(int[] A, int left, int center, int right) : The values of the possible median values are calculated and sent from quickSortM3 and then sorted in this method with simple swaps where necessary and then the value of the next to last position among these three is returned.

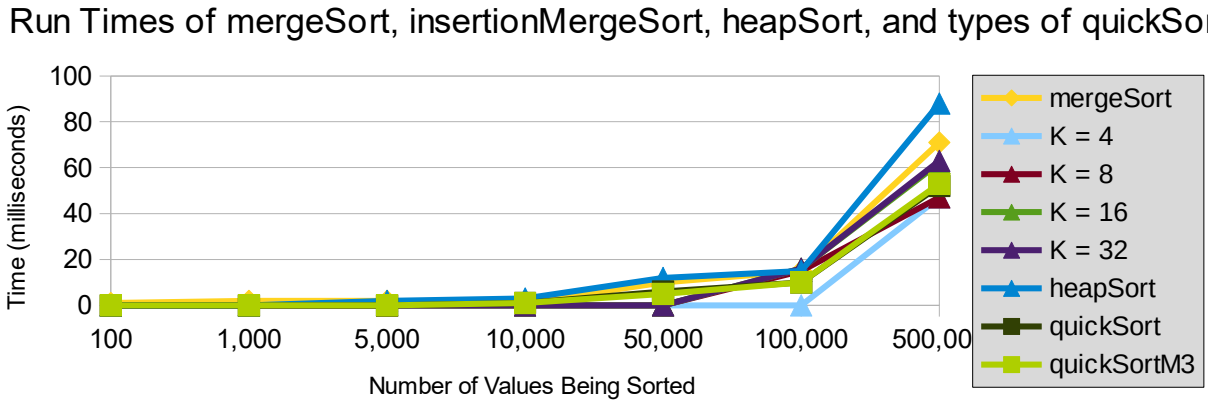
3. Testing Plan

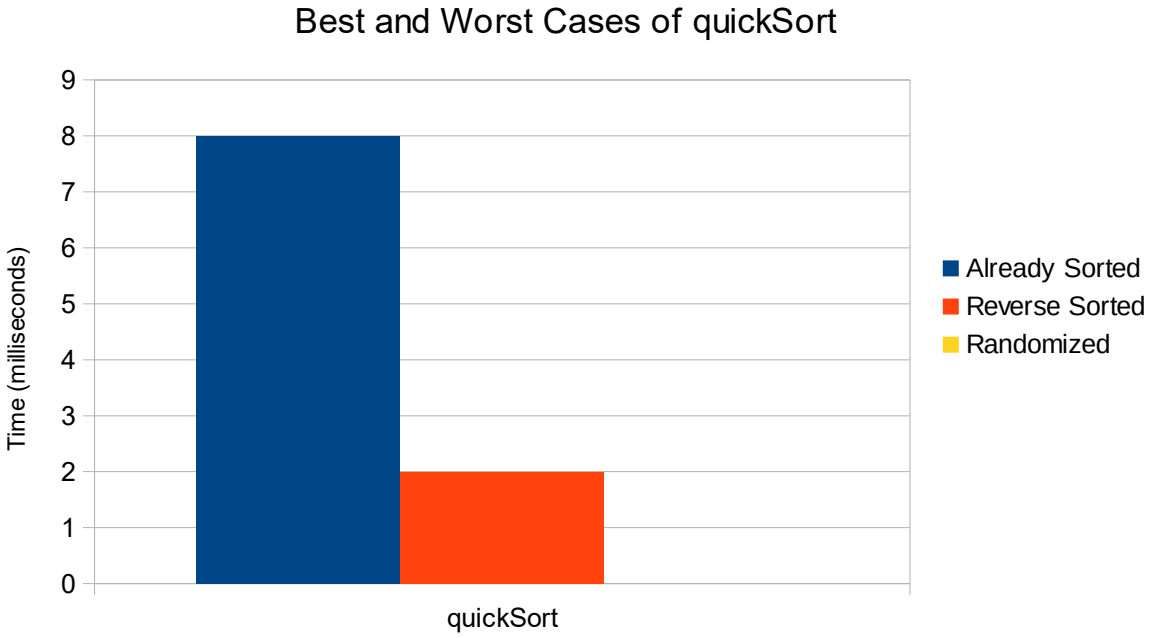
The only testing needed for this lab was to print out the corresponding values of the associated run times with each value N of the size of the arrays, which have been sorted using different algorithms, which were described and performed in previous lab assignments to include insertionSort, insertMergeSort, mergeSort, and heapSort. The time reported for the run time of each of these was already reported in milliseconds and can now be compared in a direct quantifiable means with the latest tests in run times with the same lists for heapSort. The graph and chart comparing the run times is documented below. Also documented is the difference in standard quickSort on the different versions of the list of 1,024 values mentioned prior.

4. Test Cases

Insertion/Merge/Heap/QuickSort Run Times and Variations											
Input Size (number of values)	Time (milliseconds)										
	Insertion Sort	insertMergeSort, cutoff at values K				Merge Sort	Heap Sort (max heap)	QuickSort			
		K = 4	K = 8	K = 16	K = 32			Standard	Median of 3	Best or Worst Cases (1,024 values)	
100	0	0	0	0	0	1	0	0	0	**Already Sorted**	8
1,000	4	0	0	0	0	2	0	0	0		
5,000	33	0	0	0	0	2	2	0	0	**Reverse Sorted**	2
10,000	24	0	0	0	0	1	3	1	1		
50,000	371	0	0	0	0	10	12	6	5		
100,000	1,412	0	15	16	15	15	15	10	10	**Random**	0
500,000	36,819	47	47	62	63	71	88	52	53		


Run Times of insertionSort, mergeSort, heapSort, and quickSort
Time (milliseconds)
0
10000
20000
30000
40000
insertionSort
mergeSort
heapSort
quickSort
100
1,000
5,000
10,000
50,000
100,000
500,000
Number of Values Being Sorted


Run Times of mergeSort, insertionMergeSort, heapSort, and types of quickSort
Time (milliseconds)
0
20
40
60
80
100
100
1,000
5,000
10,000
50,000
100,000
500,00
Number of Values Being Sorted
mergeSort
K = 4
K = 8
K = 16
K = 32
heapSort
quickSort
quickSortM3


Best and Worst Cases of quickSort
Time (milliseconds)
0
1
2
3
4
5
6
7
8
9
quickSort
Already Sorted
Reverse Sorted
Randomized

5. Analysis and Conclusions

Even though the worst case of quickSort could take up to as much as $O(n^2)$ run time, this is clearly evidenced to only occur when a list is already completely sorted. The sort is still slower than optimum when sorting a list that has been sorted in reversed, but ideally it works best with a completely randomized list. In terms of its comparisons to the methods we have implemented thus far, we can see that it has the fastest run time of any algorithm yet and it is approximately the same in execution with median of 3 implementation of quickSort as well. Therefore, it is best in practice to always use quickSort (as many default sort algorithms do) when one needs to sort a non-sorted list of values.

6. References

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab.

7. Time Complexity

Determine the time complexity *T(n)* for the given pseudo code. For simplicity you may assume that n is a power of 2, *i.e.,* $n = 2^k$, for some $k > 0$.

Pseudocode:	**Cost:**	**Times:**
i = n	c_1	1
while (i >= 1) {	c_2	lg(n) +1
j = i	c_3	lg(n)
while (j <= n) {	c_4	lg(n)*(lg(n)/2+1)
// some work here		
j = j * 2	c_5	lg(n)*lg(n)/2
}		
i = i / 2	c_6	lg(n)
}		

$T(n) = c_1 + c_2*\lg(n) + c_2 + c3*\lg(n) + c_4*\lg(n)^2/2 + c_4 + c_5*\lg(n)^2/2 + c_6*\lg(n)$

$T(n) = (c_4 + c_5)*\lg(n)^2/2 + (c_2 + c_3 + c_6)*\lg(n) + (c_1 + c_2 + c_4)$

$a = c_4 + c_5$, $b = c_2 + c_3 + c_6$, $c = c_1 + c_2 + c_4$

$T(n) = a*\lg(n)^2/2 + b*\lg(n) + c$

$O(T(n)) = \lg(n)^2$