Comparing MergeSort and InsertionSort Run Times with Insertion-Merge Hybrid Sorting Algorithm
Lab Assignment #3

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
September 13, 2017

1. **Problem Specification**

The purpose of this lab was to first implement mergeSort and test it for the same values we tested in the previous lab. Then, we were to alter the implementation of mergeSort to call insertionSort for some "cut off" value of a small number of inputs. In other words, once mergeSort recursed down to a small enough size subarray of the list being sorted, it would call insertionSort instead of the standard mergeSort implementation. The final task was to document differences in run time based on which small cut off value was used for small subarrays used to sort with insertionSort and compare with each other and the original mergeSort running times.

2. **Program Design**

The steps taken to develop the program were:

a) Copy over the bulk of the second lab in order to have the test cases for each scanned in list of values readily available.

b) Create a loop to copy each initial array input so two of the same array can be sent to the mergeSort algorithm as depicted in the lab pseudocode.

c) Develop the mergeSort method recursively, with the singular condition that the lower bound value denoting the starting point of the array "p" is less than the upper bound variable "r", the method would call itself on either side of the middle value each time until it reached a size of 1 where it would also call the "merge" helper method that not only does the bulk of the sorting, but also merges the parts of the array back together as it goes.

d) Changed the base case of the mergeSort method from "if p < r" to "if r – p <= cutoff" where cutoff is the highest number of inputs in an array or subarray on which we call insertionSort instead of mergeSort. This value is sent in when calling this altered form of mergeSort. In this if case, the only command is to call insertionSort on the subarray from the current p value to the current r value so it only sorts the portion of the array needed, as opposed to the whole at once. The rest of the mergeSort code lies in an else statement to behave normally the rest of the time.

e) Generate the same text printouts for all the run times with the new method of mergeSort accounted for, graph the results, and fill in the associated table for all values of n and k tested as referenced later in this report.

The following methods were used in this lab:

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) insertionSort(int [] list): takes an unsorted list of integers of any size and sorts the values in Θ(n^2) time. This is done by following the algorithm previously referenced. In an outer loop using the variable "j" for indexing, from the value of the second position to the length of the given list, a "key" value is assigned the value of the value at the index "j" followed by assigning another value "i" to "j-1". Then, in an inner loop, for every value of j, while "i" >= 0 and the value at the index "i" of the list are greater than 0, the value at the position "i+1" in the list is set equal to the value before it (index "i") and "i" is then decremented by 1. Finally, the value at "i+1" of the list is set to the value of "key" which was defined at the beginning of this iteration of the outer "j" loop. This then repeats for the next index of "j".

d) mergeSort (int[] A, int [] temp, int p, int r) : takes the array of integers A as well as its identical copy and launches the standard mergeSort algorithm as implemented from the pseudocode given in the lab. The standard algorithm is that as long as the values sent to the function continue to be unequal (p < r), the program will recursively divide the list into halves, only calling the sorting/merging helper method "merge" once the length of r – p reaches 1.

e) merge (int[] A, int[] temp, int p, int r) : Starts by copying all elements in the current subarray to a temporary array of integers, which is prepared outside of the method and sent in (temp). Then depending on the conditions of whether the right or left halves are empty, it will copy from the left or right respectively. This is the main driving force behind the actual merging of mergeSort.

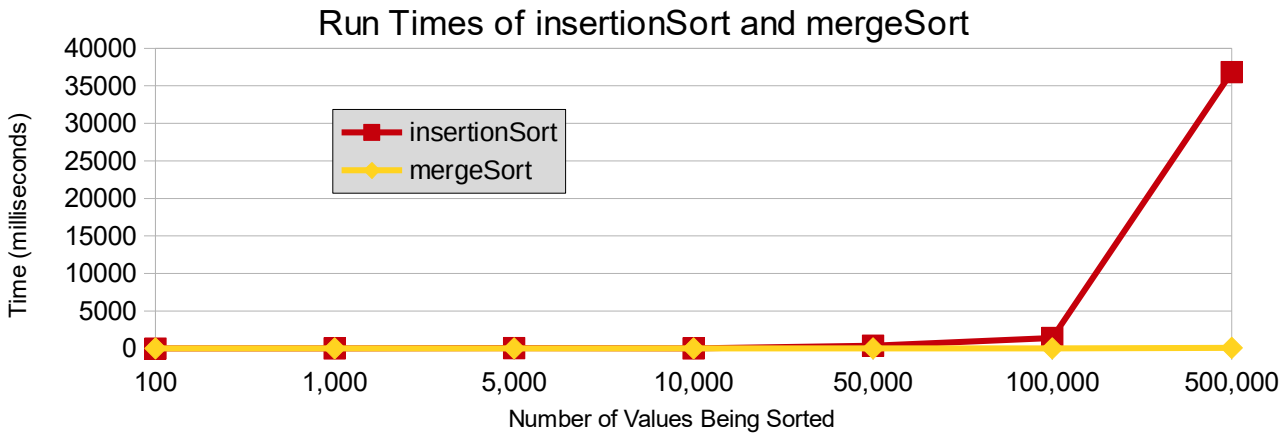
f) insertMergeSort (int[] A, int[] temp, int p, int r, int cutoff) : This is almost identical to the original mergeSort method, however, as mentioned before, it contains an altered base case that calls insertionSort on a range of sizes of inputs from 1 to the cutoff value once the program has recursed down that far. Otherwise, it behaves exactly the same as mergeSort does, calling itself on each half of the list and then merging on all sizes of inputs larger than "cutoff" before terminating.

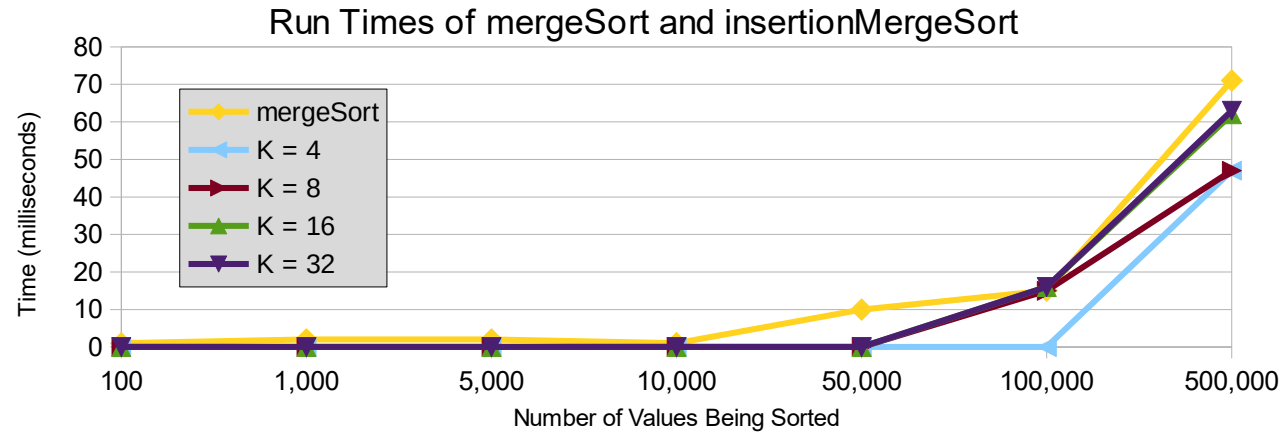
3. **Testing Plan**

Once all text prompts and associated sorting methods had been set up, all results needed to be compared could be observed almost simultaneously. The values presented in the test cases do not come from a simultaneous output of course, as the value of K (cutoff) had to be changed by sending in different values every time insertMergeSort was called. These K values were picked because they were all powers of 2 greater than 2 itself (Subsets of only 2 values would only allow one comparison and not yield a noticeable change algorithmically.), which still seemed small enough compared to all sizes of n. In all cases of K, the run times for insertMergeSort were smaller than mergeSort on average.

4. **Test Cases**

Insertion/MergeSort Run Times and Variations						
Input Size (number of values)	**Time (milliseconds)**					
	insertionSort	**insertMergeSort, cutoff at values K**				**mergeSort**
		K = 4	**K = 8**	**K = 16**	**K = 32**	
100	0	0	0	0	0	1
1,000	4	0	0	0	0	2
5,000	33	0	0	0	0	2
10,000	24	0	0	0	0	1
50,000	371	0	0	0	0	10
100,000	1,412	0	15	16	15	15
500,000	36,819	47	47	62	63	71





5. **Analysis and Conclusions**

From these graphs and tables, we can conclude that insertionSort very clearly works better on smaller values and quickly expands in run time in stark contrast to mergeSort, whose growth is completely overshadowed in the first graph. However, even mergeSort is just barely bested by the growth pattern of insertMergeSort, whose best small cutoff value is only using insertionSort for subarrays of size 4 or less at a time and delegating all greater sizes to be merged as in the standard mergeSort algorithm. With all forms of mergeSort presented (to include insertMergeSort), the values varied and sometimes showed lesser run times for greater inputs before expanding again as the graph and table also show. As mentioned before, insertionSort grows exponentially (O(n^2) time complexity) as evidenced by the incredible run time expansion in the first graph. This is mirrored slightly even in input sizes as small as the K values which were tested in the insertMergeSort method. From 4 to 32, it is visible that allowing even slightly larger inputs to be sorted by insertionSort will noticeably increase the run time. Therefore, the fastest and most efficient algorithm presented from this lab experiment was insertMergeSort which called insertionSort on subarray sizes of up to 4 and otherwise acted as standard mergeSort would.

6. **References**

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