Efficiency Analysis of Quicksort in Combination with Insertion Sort

Dillon Mabry

University of North Carolina at Charlotte Undergraduate Computer Science

Background

The essence of any efficient sorting algorithm is the combination of running time efficiency as well as the use of appropriate memory allocation for performance. By analyzing how an algorithm grows over time it is useful to construct a growth function to determine how to best use the algorithm. Quicksort is known to be an efficient algorithm used in Computer Science to sort data in many given situations. Insertion sort is an algorithm usually used on smaller sub sets of data. This report will combine quicksort with insertion sort to determine the effectiveness of using the given algorithms together.

Efficiency Analysis of Quicksort in Combination with Insertion Sort

There are many algorithms in the Computer Science field that are efficient at sorting various types of data in any given situation. One of those algorithms is quicksort which is a useful algorithm that can be applicable to many instances in the real world. Quicksort is a sorting algorithm based on partitioning sub sections of a set of elements into sorting based on comparable values. By using recursion quicksort can perform very fast sorts on data in an average time of *n log n* and a worst case time of n^2. By "over-sorting" certain sections of elements quicksort can actually decrease sorting time if the elements have already been sorted more specifically partitioned end-pieces that are present in the sorted elements.

Insertion sort is an algorithm that works well in sorting small sections of elements (usually between 9-10 elements) of data. By inserting the lesser value before the greater one insertion sort is efficient at sorting periodical sections of partitions that have a size of less than 10 in quicksort.

By sorting partitions with sizes of less than 10 using insertion sort the programmer can sort data with allegedly more efficiency than simply using quicksort as a standalone sorting algorithm. This report will analyze the effectiveness of using quicksort with insertion sort as opposed to simply using quicksort as the main sorting process.

Experimentation with Quicksort using Insertion Sort

In testing the various phases of quicksort with insertion sort the main objective was to determine the efficiency of this algorithm compared with using normal quicksort. Given a limit of size *n* the user can then set a cut-off limit of when to use insertion sort.

Whenever the size of *n* is increased the efficiency of the algorithm directly correlates to the input size of the program (input size being the number of elements to be sorted). The following chart below demonstrates how the cut-off limit affects the sorting time of the algorithm.

Limit of n=10	*Limit of n=100*	*Limit of n=1000*
Time(in ms)= 0	Time(in ms)=73	Time(in ms)=110

**For input size of ten thousand elements tested 20 times*

Limit of n=10	*Limit of n=100*	*Limit of n=1000*
Time(in ms)= 7	Time(in ms)=6	Time(in ms)=10

**For input size of one hundred thousand elements tested 20 times*

Limit of n=10	*Limit of n=100*	*Limit of n=1000*
Time(in ms)= 76	Time(in ms)=73	Time(in ms)=110

**For input size of one million elements tested 20 times*

The table above shows how modifying the cut-off limit for insertion sort the program running time is affected slightly in a manner fitting the input size. By allowing the partition sizes of the sub arrays of quicksort to be sorted by insertion sort whenever the appropriate input size is given there can be a determination on the best estimate for the size of *n*.

In summary the size of *n* or the partition size in which sub arrays should be sorted by insertion sort should be based on the given input size. Whenever a smaller input size is given the results varied little (usually consisting of very fast sorts of 0 milliseconds) however whenever the input size is increased the running time of the algorithm is affected. Whenever an input size of 100000 or larger is made the partitioned sections should be sorted based on a value of 100 or in the closest proximity. This allows the programmer to make the best judgment based on the given input he is working with at the time of running the program in a real time setting.

Comparing Quicksort with Using the Quick/Insertion Sorting Combination

Since the values of when to use insertion sort depend on the input size at higher values the use of quicksort with insertion sort depends on general knowledge of the input size beforehand. In many real world scenarios the luxury of knowing a given input size is not given at times and the programmer must design an algorithm to fit the needs of other types of situations. Testing quicksort with the quick/insertion sort combination proved that quicksort is still a viable sorting method for years to come. Although the algorithms compared differ in actual code the process remains the same: separate the given array into sub array partitions in order to allow fast sorting. Test results between optimal *n* values for the insertion sort went similar with the standard quicksort test between 100000 elements. The graph below demonstrates the test results between quicksort and using the quicksort insertion sort modified algorithm.

Quicksort/Insertion	*Quicksort*
Time(in ms)= 6	Time(in ms)=6

**For input size of one hundred thousand elements tested 20 times*

Other tests performed using the standard quicksort algorithm vs. the quicksort/insertion sort algorithm remain similar in nature. Quicksort and the modified algorithm go evenly against values up to one million elements when using optimal values for the modified sorting code. If optimal values are not used there can be consequences when preferring the insertion code over standard quicksort. Running times are in fact reduced when n values of smaller size are used with large sets of data. Test results below should larger data used with n values of less than 10 (which is normally the agreed upon optimal value for standard insertion sort).

Quicksort/Insertion	*Quicksort*
Time(in ms)= 7	Time(in ms)=6

**For input size of one hundred thousand elements tested 20 times with insertion sort cutoff limit of less than 10*

Quicksort/Insertion	*Quicksort*
Time(in ms)= 85	Time(in ms)=71

**For input size of one million elements tested 20 times with insertion sort cutoff limit of less than 10*

Evidence suggests that when using quicksort/insertion sort without optimal n values the time values are increased compared with standard quicksort. Whenever the input size is increased the chance of error when generating a likely cutoff limit for insertion sort is great if the programmer does not already have a designated algorithm for this.

Summarization of Points

In conclusion the algorithm of using insertion sort with quicksort is a practical concept in nature with a more in-depth background of knowing certain features about the program at hand. Without prior knowledge of a general input size the program can be thrown off without having a specific algorithm to calculate optimal *n* values for using insertion sort. *N* values being the cutoff limit for insertion sort to be used on the given partition of the sub array in the array being sorted. If the optimal cutoff limit is used for insertion sort the algorithm is generally faster but on an insignificant scale for input sizes of less than one million elements. This is not to say that the algorithm is to be not used or would not improve greatly in higher input values.

Extended Notes

**Machine used for testing included an i7-4770k processor @ 3.8 GHz as well as 16 GB DDR3-1866mhz RAM on an ASUS motherboard.*

**Quicksort methods included recursive method calls as well as appropriate comparisons with other codes to ensure stability*