Analysis and Determination of Asymptotic Behavior Range For Popular Sorting Algorithms

Omar Khan Durrani, Shreelakshmi V, Sushma Shetty & Vinutha D C

Department of Computer Science & Engineering, Vidya Vikas Institute of Engineering & Technology, Mysore,India
E-mail : omardurrani2003@yahoo.com,v.shreelaksmi@gmail.com,sush.1312@gmail.com, vinuthadc@gmail.com

Abstract - Theories of Computer Sciences & Engineering nowadays are being only read than being designed and practiced. Hence in this paper and [8] we have featured the theories and practices relying beneath popular sorting algorithms and their performance measurement in our experiments for the realization of efficiency class. Further we have concentrated on finding the Asymptotic Behavior Range for the two classes of sorting algorithms (n^2 & nlogn). We have found shell sort and quick sort outperforming in their respective efficiency class of sorting algorithm.

Keywords- *Sorting efficiency classes, n^2class, nlogn class, Asymptotic Behavior Range (ABR), time complexity, Quick sort, Heap sort, merge sort, bubble sort, selection sort, insertion, shell sort, worst case, and random data set.*

I. INTRODUCTION

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.Although many consider it a solved problem; useful new sorting algorithms are still being invented. Theoreticians have classified two classes of popular Sorting Algorithms:

- O (n^2): Bubble Sort, Insertion Sort, Selection Sort, Shell Sort
- O (nlogn): Heap Sort, Merge Sort, Quick Sort.

The Asymptotic class O (n^2) is slower in time than O (nlogn) class of algorithms.

In [8], we have defined **Asymptotic Behavior Range (ABR)** as the range of n samples in which algorithms of same efficiency class start to show their asymptotic behavior. Every efficiency class of algorithm has its own ABR. The ABR also differs with respect to computer systems under consideration.

In the following section we have briefly explained the theoretical aspects of the above mentioned popular sorting algorithm and previous analysis made by authors mentioned in the reference. Further we have conducted the experiments on these sorting algorithms on a test bed to realize their behavior and further investigate in their performance. Finally the findings made with respect to the **Asymptotic Behavior Range (ABR)** for the different sorting methods and made with some analysis. The bird's eye section shows the programming skill used to clock the time.

II. POPULAR SORTING ALGORITHMS AND PREVIOUS ANALYSIS DONE

A. Bubble Sort

The bubble sort is the oldest and simplest sort in use. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. This causes larger values to "bubble" to the end of the list while smaller values

"sink" towards the beginning of the list.

Pros:

- Simplicity and ease of implementation.
- Auxiliary Space used is O (1).

Cons:

- Very inefficient.
- General complexity is O (n^2).
- Best case complexity is O (n).

B. Selection Sort

Selection Sort's philosophy most closely matches human intuition: It finds the largest element and puts it in its place. Then it finds the next largest and places it and so on until the array is sorted. To put an element in its place, it trades positions with the element in that location (this is called a swap). As a result, the array will have a section that is sorted growing from the end of the array and the rest of the array will remain unsorted

Pros:

- **S**pecifically an in-place comparison sort.
- Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.
- It yields a 60% performance improvement over the bubble sort.

Cons:

- It has *O (n2)* complexity, making it inefficient on large lists.
- Generally performs worse than the similar insertion sort.

C. Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Pros:

- Auxiliary space used is O (1).
- The insertion sort is a little over twice as efficient as the bubble sort.

Cons:

- General Complexity is O (n^2).
- Best Case is O (n) when the list is already sorted.

D. Shell Sort

Shell sort is a sorting algorithm, devised by Donald Shell in 1959, that is a generalization of insertion sort, which exploits the fact that insertion sort works efficiently on input that is already almost sorted. It improves on insertion sort by allowing the comparison and exchange of elements that are far apart. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted. Although Shell sort is easy to code, analyzing its performance is very difficult and depends on the choice of increment sequence. The algorithm was one of the first to break the quadratic time barrier, but this fact was not proven until some time after its discovery.

Pros:

- The algorithm is an example of an algorithm that is simple to code.
- The algorithm was one of the first to break the quadratic time barrier.
- The algorithm itself does its sorting in-place.
- Although sorting algorithms exist that are more efficient, Shell sort remains a good choice for moderately large files because it has good running time. The shell sort is by far the fastest of the n^2class of sorting algorithms. It is more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

Cons:

- Difficult to analyze theoretically.
- The initial increment sequence suggested by Donald Shell was *[1,2,4,8,16,...,2k]*, but this is a very poor choice in practice because it means that elements in odd positions are not compared with elements in even positions until the very last step.

E. Heap Sort

All elements to be sorted are inserted into a heap, and the heap organizes the elements added to it in such a way that either the largest value (in a max-heap) or the smallest value (in a min-heap) can be quickly extracted. Moreover, because this operation preserves the heap's structure, the largest/smallest value can be repeatedly extracted until none remain. Each time we delete (extract) the maximum, we place it in the last location of the array not yet occupied, and use the remaining prefix

of the array as a heap holding the remaining unsorted elements. This gives us the elements in order.

***Pros*:**

- Time complexity of the algorithm is O (n log n).
- Auxiliary Space required for the algorithm is O (1).
- In-space and non-recursive makes it a good choice for large data sets.

Cons:

- Works slow than other such DIVIDE-AND-CONQUER sorts that also have the same O (n log n) time complexity due to cache behavior and other factors.
- Unable to work when dealing with linked lists due to non convertibility of linked lists to heap structure.

F. Merge Sort:

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of O (n log n). Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.

Pros:

- Marginally faster than the heap sort for larger sets.
- Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
- Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Cons:

- At least twice the memory requirements of the other sorts because it is recursive. This is the BIGGEST cause for concern as its space complexity is very high. It requires about a Θ (n) auxiliary space for its working.
- Function overhead calls (2n-1) are much more than those for quick sort (n). This causes it to take more time marginally to sort the input data.

G. Quick Sort

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

Pros:

- One advantage of parallel quick sort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.
- All comparisons are being done with a single pivot value, which can be stored in a register.
- The list is being traversed sequentially, which produces very good locality of reference and cache behavior for arrays.

Cons:

- Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of O (n^2) which can prove very fatal for large data sets. Competitive sorting algorithms.

III. PERFORMANCE MEASUREMENT

A. Bird's-Eye View

Performance measurement is concerned with obtaining the actual time requirements of a program. To obtain the execution time of a program, we need a clocking mechanism. We shall use the C++ function clock(), which measures time in ticks. The constant CLOCKS_PER_SEC, which is defined in the header file time.h, gives us the number of ticks in one second. This constant is used to convert from ticks to seconds. For our system, CLOCKS_PER_SEC=1000. So, one tick=1 millisecond. The following main() illustrates how we performed the experiments to clock the time with Bird's-eye view [1].

Program to obtain worst-case run time:

```
int main()
 {  int a[10000],step=1000;
   double clockspermillis=double
(CLOCK_PER_SEC)/1000;
```

```
cout<<"Worst case time, in milliseconds, are"<<endl;
cout<<"n \t Time"<<endl;
for (int n=0;n<=10000;n+=step)
{
 for (i=0;i<n;i++)
 a[i]=n-i;
 clock_t startTime=clock();
  // sort function call
 double elapsedmillis=(clock()-startTime)
/clockpermillis;
 cout<<n<<'\t'<<elaapsemiliis<<endl;
 }
return 0;
}
```

B. Experiment Results And Analysis:

In this section we have used a computer system described below as our test bed on which our experiments are conducted. We have first shown results of **nlogn sorting class** on the test bed followed by the **n^2 sorting class** other. *Test bed :*Intel ®,Pentium ® D.CPU 2.80 GHz, 2.79 GHz, 512 MB of RAM, System: Micro Windows XP, Professional, Version 2002, Service Pack 2.

TABLE 1: SHOWS THE TIME TAKEN FOR VARIOUS SAMPLES OF N

N	Time/sort (merge sort)	Time/sort (heap sort)	Time/sort (quick sort)
1000	1.663036	0.428866	1.450848
2000	1.717999	0.939569	1.685152
3000	2.25647	1.484881	1.4011731
4000	2.639559	2.046371	2.03455
5000	3.105469	2.627441	2.252955
6000	3.43471	3.219374	2.197802
7000	3.434495	3.817219	2.792917
8000	3.938714	4.422849	2.96888
9000	4.472532	5.042681	2.892149
10000	5.002736	5.673212	3.431064

From the practical analysis and by observing the table of results (time clocked) of these experiments we found the range of time clocked for various n samples for Heap sort, Merge sort and Quick sort was between those set of time clocked ranges of linear search and Selection sort which fall under the efficiency classes linear and quadratic respectively. Hence this made us clear about the efficiency class of the three Sorting algorithms Quick, Merge & Heap Sort which is **n log n**.

In our experiments which are concentrated on n-samples up to 10,000 values, where, we see the start of asymptotic behavior. The data generated for all the three **nlogn class sorting algorithms** is by using the random number generating function rand() of 'C/C++'. The table 1 shows the time clocked ranges for each of the sorting methods. We can now observe that merge Sort which is initially slower than Heap Sort getting faster during n=6000, n=8000 and later on. Also we see quick sort performing faster from n=2000 onwards. Hence we call this range where all the three sorting algorithm stabilize asymptotically as **Asymptotic Behavior Range (ABR).** The ABR for **nlogn** sorting class is 1000 to10000.

The respective values and graphs for nlogn class of sorting algorithms our test bed are as given below in fig 2.1.

In our experiments conducted for **n^2 sorting class of algorithms** which included bubble sort, selection sort, insertion sort and shell sort We have found that the algorithms are of course of $O(n^2)$ class of complexity with respect to time. We can also see the time of sorting n=10,000 random numbers in case of quick sort is 3.431064 milliseconds which is the fastest among nlogn class and that of shell sort which has clocked the time 74.451294 milliseconds for the same randomly generated set. The following are some of the observations made with respect to speed and asymptotic behavior.

TABLE 2: SHOWS THE TIME TAKEN FOR VARIOUS SAMPLES OF N

N	Time/sort (bubble sort)	Time/sort (selection sort)	Time/sort (insertion sort)	Time/sort (shell sort)
0	0.000467	0.000466	0.000472	0.000488
10	0.001024	0.000883	0.00082	0.000837
20	0.0022	0.001795	0.001354	0.001342
30	0.00391	0.003124	0.002036	0.001994
40	0.006185	0.004876	0.002882	0.002794
50	0.009049	0.007079	0.003894	0.003738
60	0.012514	0.009742	0.005047	0.00483
70	0.0166	0.012869	0.006379	0.00606
80	0.021308	0.01644	0.007857	0.007434
90	0.026637	0.020474	0.009488	0.008954
100	0.032616	0.024996	0.011284	0.010616
500	0.808836	0.593007	0.213691	0.195774
10^3	3.243893	2.370365	0.829597	0.754046
5.10^3	80.32902	59.65804	20.289902	18.64440
10^4	320.0868	238.8915	81.001473	74.45129

SPEED:

- In worst cases, Selection sort performs best amongst n^2 sorting class; it is 58.5% faster than bubble sort and 33% faster than shell sort. Bubble and insertion sort have equal speeds.
- With random data set (average cases), Shell sort performs best amongst n^2 sorting class. It is about 10% faster than insertion sort, 3 times faster than selection and about 4.5 times faster than bubble sort.

ASYMPTOTIC BEHAVIOUR:

- In worst case, initially bubble sort will be slower than insertion sort and becomes slight faster than insertion sort when the size of input reaches to 40(n=40). Selection sort gets its behavior right from n=10. Shell sort gets its behavior from n=20, because at n=10 bubble is faster and becomes slower from n=20 onwards.
- On randomly generated data set (for average cases), as shown in the table 2 and graph plot in figure 2.1, we see that shell sort getting its behavior from n=20 onwards, insertion sort and bubble sort getting from n=10 onwards, selection sort right from the start.
- Finally, we see that the ABR for n^2 sorting class is 10 to50.

The respective graphs for n^2 class of sorting algorithms are given below in fig 2.2.

IV. CONCLUSSION

From theories, as well from performance measurement it is clear that the two set of sorting algorithm classes namely the **nlogn** class of efficiency and **n^2**class of efficiency performed as intended. Merge and Heap fall always under the nlogn efficiency class for all the cases. Merge sort is slower in the ABR than the Heap sort initially and hence attains its speed in the higher instances of n samples. Quick sort is faster than the other two, right from the start of ABR for data sets generated using rand() function of 'C/C++'.Quick sort performs $O(n^2)$ for data set which is in sorted order, which is rarely seen. Designing average case data set for nlogn class without using rand function is also not noticed in the previous researches. Shell sort out performs in n^2 sorting class of algorithms. Hence it is better to use shell sort when you have moderate data set. Further improving in the speed is felt difficult in case of both the sorting algorithm classes. ABR for **nlogn** class is 1000-10000 and that for **n^2 class** is 10-50. Bubble sort is really slow to use, rather can be used in the theories. The reducing cost of internal memory and increasing processors in the core should be considered and research should be encouraged for developing a new sorting algorithm which might fall close to the linear efficiency class.

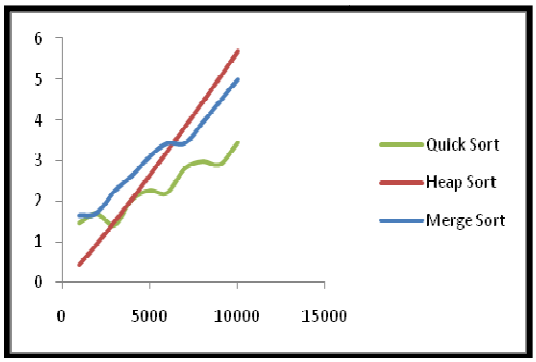

Figure 2.1: graph plot for random data set

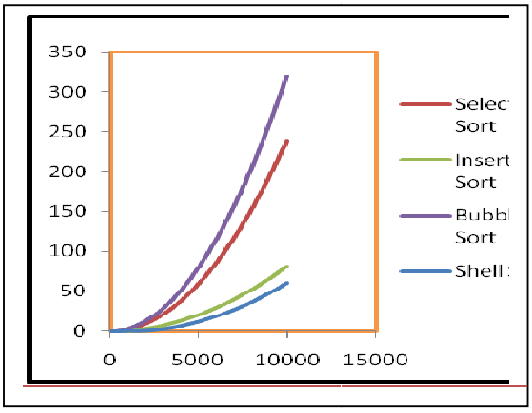

Figure 2.2: graph plot for random data set

ACKNOWLEDGEMENT

We cannot forget Mrs. Meenakshi H N for her full support and encouragement which is inexpressible. Also, we want to thanks colleagues of our department who had directly or indirectly supported. At last we cannot forget the students of fifth semester, 2009 batch, who helped in making this practically implemented and get published. Also we want to thanks the staff of E.I.T chamarajanagar district that co-operated to execute the n^2 sorting class at their laboratory, especially H A Sharath Lecturer in computer Science Department, EIT chamarajanagara.

REFERENCES

[1] Sartaj Sahni, "Data structures and Algorithms in C++", university press ublication 2004, chapter 4: performance measurement, pages 123-136.

[2] Levitin, A. Introduction to the Design and Analysis of Algorithms. Addison-Wesley, Boston MA, 2007.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest,Clifford Stein, "Introduction to Algorithms", Second Edition, Prentice-Hall New Delhi,2004.

[4] Vandana Sharma, Parvinder S. Sandhu, Satwinder Singh, and Baljit Saini," Analysis of Modified Heap Sort Algorithm on Different Environment", World Academy of Science, Engineering and Technology 42 2008.

[5] Yediyah Langsam, Moshe J Augenstein,Aaron M Tenenbaum, "An introduction to data structures with c++" Prentice hall India Learning private limited,2e 2008..

[6] "Sorting Algorithm Analysis", Gina Soileau, Muhammad Younus, Suresh Nandlall,Tamiko Jenkins, Thierry Ngouolali, Tom Rivers,Data Structures and Algorithms (SMT-274304-01-08FA1), Professor James Iannibelli, December 21, 2008.

[7] Pooja Adhikari, "Review On Sorting Algorithms-A comparative study on two sorting algorithms", A Term Paper Submitted to the Faculty of Dr. Gene Boggess, Mississippi State University, In the Department of Computer Science & Engineering, Mississippi State, Mississippi.

[8] Omar Khan Durrani, Shreelakshmi V, Sushma Shetty," Performance Measurement and analysis of sorting Algorithms", National Conference on Convergent innovative technologies and management (CITAM-11),held on Dec 2-3,2011 at Cambridge Institute of Technology and Management, Bangaluru.

[9] C.Canaan, M.S Garai, M Daya,"Popular Sorting Algorithms",World Applied Programming, Vol 1,No. 1,April 2011,pages 42-50.

[10] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging.

◈◈◈