ISSN 2229-5518

Evaluation of Sorting Algorithms, Mathematical and Empirical Analysis of sorting Algorithms

Sapram Choudaiah P Chandu Chowdary M Kavitha

ABSTRACT:Sorting is an important data structure in many real life applications. A number of sorting algorithms are in existence till date. This paper continues the earlier thought of evolutionary study of sorting problem and sorting algorithms concluded with the chronological list of early pioneers of sorting problem or algorithms. Latter in the study graphical method has been used to present an evolution of sorting problem and sorting algorithm on the time line.

An extensive analysis has been done compared with the traditional mathematical methods of —Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort. Observations have been obtained on comparing with the existing approaches of All Sorts.

An "Empirical Analysis" consists of rigorous complexity analysis by various sorting algorithms, in which comparison and real swapping of all the variables are calculatedAll algorithms were tested on random data of various ranges from small to large. It is an attempt to compare the performance of various sorting algorithm, with the aim of comparing their speed when sorting an integer inputs.The empirical data obtained by using the program reveals that Quick sort algorithm is fastest and Bubble sort is slowest**.**

Keywords: Bubble Sort, Insertion sort, Quick Sort, Merge Sort, Selection Sort, Heap Sort,CPU Time.

Introduction

In spite of plentiful literature and research in sorting algorithmic domain there is mess found in documentation as far as credential concern2. Perhaps this problem found due to lack of coordination and unavailability of common platform or knowledge base in the same domain. Evolutionary study of sorting algorithm or sorting problem is foundation of futuristic knowledge base for sorting problem domain1. Since sorting activity is known as pre-requisition or supportive activity (searching, Matching etc.) for the various other computer related activities3. This activity (sorting) has a distinct place in the computing and programming domain. It could possible and quit obvious that some of the important contributors or pioneers name and their contribution may skipped from the study. Therefore readers have all the rights to extent this study with the valid proofs. Ultimately our objective behind this research is very much clear, that to provide strength to the evolutionary study of sorting algorithms and shift towards a good knowledge base to preserve work of our forebear for upcoming generation. Otherwise coming generation could receive hardly information about sorting problems and syllabi may restrict with some major/fundamental algorithms only. Evolutionary approach of sorting can make learning process alive and gives one more dimension to student for thinking4. Whereas, this thinking become a mark of respect to all our ancestors.

This paper investigates the characteristic of the sorting algorithms with reference to number of comparisons made and number of swaps made for the specific number of elements. Sorting algorithms are used by many applications to arrange the elements in increasing/decreasing order or any other permutation. Sorting algorithms, like Quick Sort, Shell Sort, Heap Sort, Insertion Sort, Bubble Sort etc. have different complexities depending on the number of elements to sort. The purpose of this investigation is to determine the number of comparisons, number of swap operations and after that plotting line graph for the same to extract values for polynomial equation. The values a, b and c got is then used for drawing parabola graph. Through this paper, a conclusion can be brought on what algorithm to use for a large number of elements. For larger arrays, the best choice is Quicksort, which uses recursion method to sort the elements, which leads to faster results. Program for each sorting algorithm in which a counter is used to get the number of comparisons, number of swap/assignment operations is used. The data is

IJSER

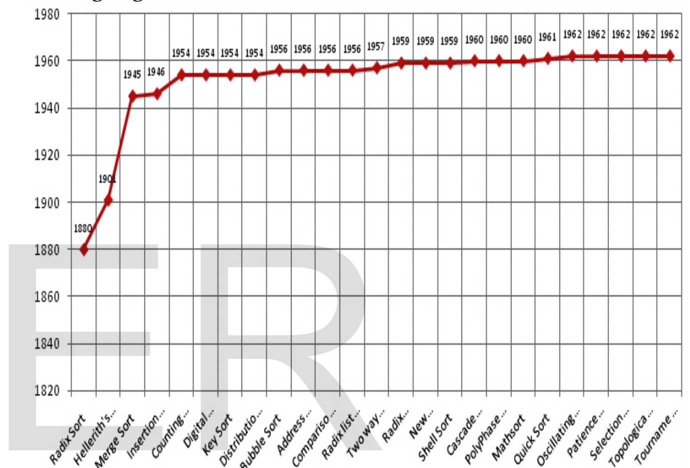
stored in a file, from where it is used for calculation purpose in an excel file. Least square method and Matrix inversion method is used to get the value of constants a, b and c for each polynomial equation of sorting algorithms. After calculatingthe values, Graph is drawn for each sorting algorithmfor the polynomial equation i.e. Y=AX^2+BX+C or Y=AX.logX+BX+C.

Below is the list of inventors of sorting and theirsorting invention along with invention year

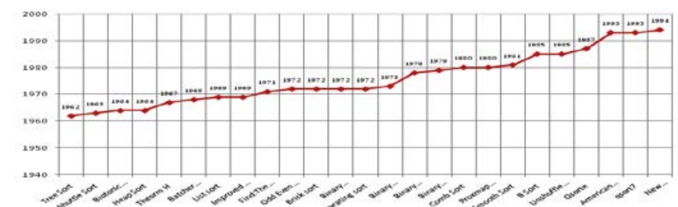
Sr. No.	Sorting Algorithm	Inventors Name	Invention Year
1	Radix Sort	Herman Hollerith	1880
2	Hellerith's Sorting Machine	Herman Hellerith	1901
3	Merge Sort	John Van Neumann	1945
4	Insertion Sort(Binary Insertion)	john Mauchly	1946
5	Counting Sort	] Harold H. Seward	1954
6	Digital Sorting		1954
7	Key Sort		1954
8	Distribution sort	H.Seward	1954
9	Bubble Sort(Exchange sort)	Inverson	1956
10	Address calculation sorting	Issac and singleton	1956
11	Comparison Sort	E.H.Friend	1956
12	Radix list sort	E.H.Friend	1956
13	Two way insertion sort	D.J.Wheeler	1957
14	Radix Sort(Modifed)	P.Hildebrandt,H.Rising,JScwartz	1959
15	New Merge Sort	B.K. Betz & W.C. Carter	1959
16	Shell Sort	Donald L Shell	1959
17	Cascade Merge Sort	R.L.Gilstad	1960
18	PolyPhase Merge/Fobinocii Sort	R.L.Gilstad	1960
19	Mathsort	W.Feurzeig	1960
20	Quick Sort (Partition Exchange sort)	CAR Hoare	1961
21	Oscillating Merge Sort	Sheldon Sobel	1962
22	Patience Sort	C. L. Mallow	1962
23	Selection Sort	NA	1962
24	Topological Sort	Kahn	1962
25	Tournament Sort(tree sort)	K.E..Iversion	1962
26	Tree Sort(Modified)	K.E..Iversion	1962
27	Shuttle Sort		1963
28	Biotonic Merge sort	US atent3228946(1969)K.E.Batcher	1964
29	Heap Sort	J.W.J Willams	1964
30	Theorm H	Douglas H.Hunt	1967
31	Batcher Odd-Even Merge Sort	Ken Batcher	1968
32	List sort/List merge sort	L.J.Woodrum&A.D.Woodall	1969
33	Improved Quick sort	Singleton	1969
34	Find:The Program	CAR Hoare	1971
35	Odd Even /Brickt sort	Habermann	1972
36	Brick sort	Habermann	1972
37	Binary Merge sort	F.K.Hawang&S.Lin	1972
38	gyrating sort	R.M.Karp	1972
39	Binary Merge sort	F.K.Hawang& D.N. Deutsh	1973
40	Binary Merge sort	C.Christen	1978
41	Binary Merge sort	G.K.Manacher	1979
42	Comb Sort	Wdzimierz	1980
43	Proxmap Sort	Thomas A. Standish	1980
44	Smooth Sort	EdsgerDijkstra	1981
44	B Sort	Wainright	1985
45	Unshuffle Sort	Art S. Kagel	1985
46	Qsorte	Wainright	1987
47	American Flag Sort		1993
48	New Efficient Radix Sort	Arne Anderson & Stefan Nilson	1994
49	Self-Indexed sort(SIS)	Yingxu Wang	1996
50	Splay sort	Moggat, Eddy &Petersson	1996
51	Flash Sort	Karl-Dietrich Neubert	1997
52	Introsort	David Musser	1997
53	Gnome Sort	Dr. Hamid Sarbazi-Azad	2000
54	Tim sort	Tim Peters	2002
55	Spread sort	Steven J. Ross	2002
56	Tim sort	Tim Peters	2002
57	Bead Sort	Joshua J. Arulanandham, Cristian S	2002
58	Burst Sort	Ranjansinha	2004
59	Libarary Sort/Gapped Insertion sort	Michael A. Bender, Martín	2004
60	Cycle Sort	B.K.Haddon	2005
61	Quicker sort	R.S. Scowen	2005
62	Pancake sorting	Professor Hal Sudborough	2008
63	U Sort	Upendra singh aswal	2011
64	Counting Position Sort	NitinArora	2012
65	Novel Sorting Algorithm	R.Shriniwas&A.RagaDeepthi	2013
66	Bogo Sort(Monkey sort)	NA	NA
67	Bucket Sort	NA	NA
68	SS06 Sort	K.K.Sudharajan&S.Chakraborty	NA
69	Stooge Sort	Prof.Howard Fine and Howard	NA
70	J Sort	Jason Morrison	NA
71	Strand Sort	NA	NA
72	Trim Sort	NA	NA
73	Punch Card Sorter	A. S. C. Ross NA	

Table 1: Inventors of sorting and their sorting invention along with invention year

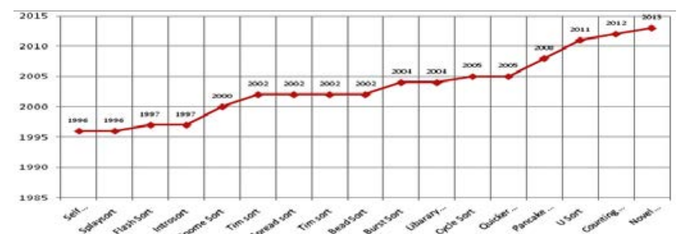
The graphical representation of evaluation of sorting algorithms:



Graph 1:Sorting Algorithms (1880-1962)



Graph2:Sorting Algorithms (1962-1994)



Graph 3:Sorting Algorithms (1996-2013)

IJSER

ISSN 2229-5518

Complexity of Algorithm

There are two aspects of algorithmic performance:

- Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- Space
  - Data structures take space
  - kind of data structures can be used?
  - How does choice of data structure affect the runtime?

Here we will focus on time: How to estimate the time required for an algorithm

T(n)	Name	Problems
O(1) O(log n) O(n) O(n log) O(n2) O(n3)	Constant Logarithmic Linear Linear-log. Quadratic Cubic	Easy-solved
O(2n) O(n!)	Exponential Factorial	Hard-solved

Mathematical vs. Empirical Analysis

Mathematical Analysis	Empirical Analysis
The algorithm is analyzed with the help of mathematical deviations and there is no need of specific input.	The algorithm is analyzed by taking some sample of input and no mathematical deviation is involved
The principal weakness of these types of analysis is its limited applicability.	The principal strength of Empirical analysis is it is applicable to any algorithm.
The principal strength of Mathematical analysis is it is independent of any input or the computer on which algorithmis running.	The principal weakness of Empirical analysis is that it depends upon the sample input taken and the computer on which the algorithm is running’

Mathematical Analysis of Some Sorting Algorithms

The common sorting algorithms can be divided into two classes by the complexity of their algorithms as, (n2), which includes the bubble, insertion, selection, and shell sorts , and (n log n) which includes the heap, merge, and quick sorts.

(A) Selection Sort

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 \in \Theta(n2)$ comparisons (see arithmetic progression). Each of these scans requires one swap for $n - 1$ elements (the final element is already in place). Among simple average-case $\Theta(n2)$ algorithms, selection sort almost always outperforms bubble sort and gnome sort, but is generally outperformed by insertion sort. Insertion sort is very similar in that after the kth iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs in order to place the k + 1st element, while selection sort must scan all remaining elements to find the k + 1st element. Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort, although it can perform just as many or far fewer depending on the order the array was in prior to sorting. It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably. However, this is more often an advantage for insertion sort in that it runs much more efficiently if the array is already sorted or "close to sorted." While selection sort is preferable to insertion sort in terms of number of writes ($\Theta(n)$ swaps versus $O(n2)$ swaps), it almost always far exceeds (and never beats) the number of writes that cycle sort makes, as cycle sort is theoretically optimal in the number of writes. This can be important if writes are significantly more expensive than reads, such as with EEPROM or Flash memory, where every write lessens the lifespan of the memory.

IJSER

(B) Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's the slowest one. 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 (in other words, all items are in the correct order This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. While the insertion, selection and shell sorts also have O (n2) complexities, they are significantly more efficient than the bubble sort. A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items. Clearly, bubble sort does not require extra memory.

(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. Like the bubble sort, the insertion sort has a complexity of O (n2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort. It is relatively simple and easy to implement and inefficient for large lists. Best case is seen if array is already sorted. It is a linear function of n. The worst-case occurs; when array starts out in reverse order .It is a quadratic function of n. The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple

hundred items. Since multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array, Insertion sort is stable. This algorithm does not require extra memory.

(D) Quick Sort

From the initial description it's not obvious that quick sort takes $O(n \log n)$ time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses O (n) time. In versions that perform concatenation, this operation is also O (n).

In the best case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only nested calls before we reach a list of size 1. This means that the depth of the call tree is . But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only O ($n \log n$) time.

Analytical Comparison A limitation of the empirical comparison is that it is system-dependent. A more effective way of comparing algorithms is through their time complexity upper bound to guarantee that an algorithm will run at most a certain time with order of magnitude $O(f(n))$ where is the number of items in the list to be sorted. This type of comparison is called asymptotic analysis. The time complexities of the algorithms studied are shown in below table.

Algorithm	Time Complexity		
	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n$^2)	$O(n$^2)
Insertion Sort	$O(n)$	$O(n$^2)	$O(n$^2)
Selection Sort	$O(n.\lg(n))$	$O(n$^2)	$O(n$^2)
Quick Sort	$O(n.\lg(n))$	$O(n.\lg(n))$	$O(n$^2)

Shell Sort	$O(n.\lg(n))$	$O(n.\lg(n))$	$O(n\^2)$
Heap Sort	$O(n.\lg(n))$	$O(n.\lg(n))$	$O(n.\lg(n))$

Table2: The time complexities of the algorithms

Although all algorithms have a worst-case runtime of $O(n\^2)$, only Quicksort & Shell Sort has a best and average runtime of $O(n.\lg(n))$' This means that Quicksort & Shell Sort, on average, will always be faster than Bubble, Insertion and Selection sort, if the list is sufficiently large'
$O(n)$ factor work plus two recursive calls on lists of size in the best case, the relation would be.$T(n) = O(n) + T(n/2)$

An alternative approach is to set up a recurrence relation for the $T(n)$ factor, the time needed to sort a list of size. Because a single quick sort call involves

The master theorem tells us that $T(n) = O(n \log n)$. In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other (or any other fixed fraction), the call depth is still limited to, so the total running time is still $O(n \log n)$.

Empirical Analysis of Some Sorting Algorithms

General Plan for Empirical Analysis of Algorithms:

1. Understand the purpose of experiment of given algorithm
2. Decide the efficiency matrix M. Also decide the measurement. For example operation's count vs. time.
3. Decide on characteristic of input.
4. Create a program for implementing the algorithm. This program is ready experiment.
5. Generate a sample of input.
6. Run the algorithm for some set of input sample. Record the result obtained.
7. Analyze the resultant data

Empirical comparison

a. Tests made

The tests were made using the C. Each algorithm was run on the lists of length of 1, 3, 5, 7, 10, and 15 lakhs. The number of comparisons and number of Assignment/Swap operations was recorded by using a counter for number of comparisons and number of Assignment/Swap operations. The code was run on Windows 7, with an Intel Core i5 processor and 3GB of RAM. The raw results were recorded by the reading and writing in the file. These raw results were tabulated, Calculated, and graphed using C and MS-Excel.

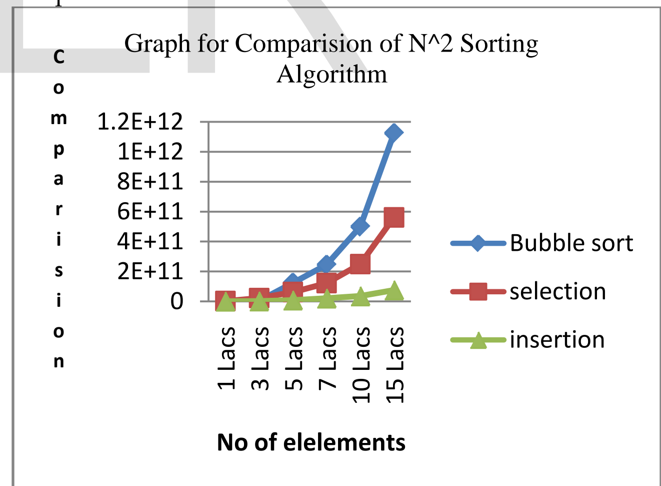
b. Results

Total Results The total results for all runs for each algorithm and each list length are shown on Table and Graph: Table for number of comparisons of sorting algorithm on given number of elements

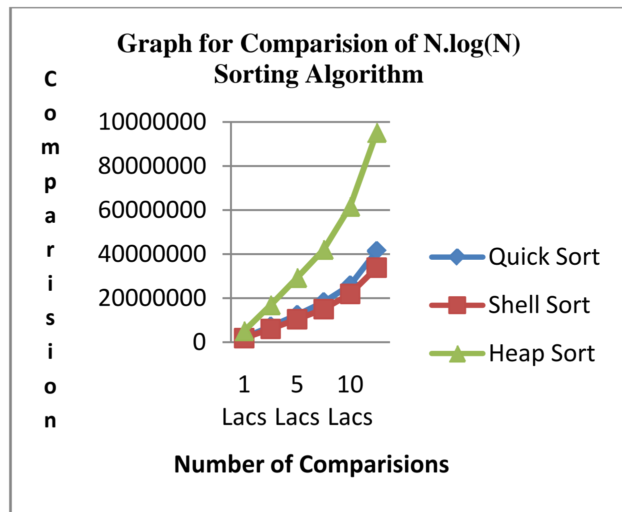
Algo.	Number of elements					
	1 lacs	3 lacs	5 lacs	7 lacs	10 lacs	15 lacs
Bubble Sort	499950001	44999850001	124999750001	244999650001	499999500001	1124999250001
Insertion Sort	2503921057	22500033726	62489124089	122464377656	249931402775	562246099741
Selection Sort	717121978	2040124110	8248512308	21247437765	35983140378	76314509973
Quick Sort	2083013	7154514	12439847	18266103	25881883	41478603
Shell Sort	1868928	6075712	10475712	15052412	21951424	33902848
Heap Sort	5173930	16938139	29321482	42113995	61645978	95209435

Table 3: Number of comparisons for sorting algorithms

Graph drawn from values obtained from Table 3



Graph 4: Graph for Comparison of Sorting Algorithm 2 ^N



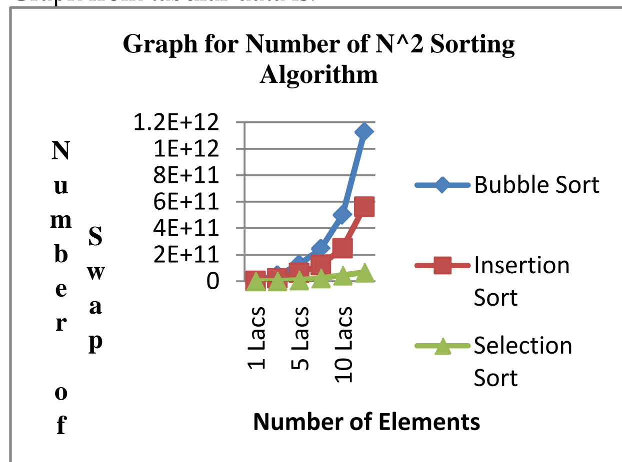
Graph 5: Graph for Comparison of N.log (N) Sorting Algorithm

Table for number of Swap/Assignment of sorting algorithm on given number of elements:

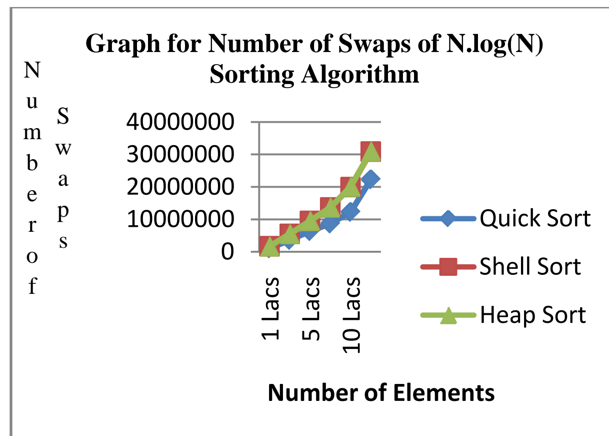
Algo.	Number of elements					
	1 lacs	3 lacs	5 lacs	7 lacs	10 lacs	15 lacs
Bubble Sort	4999950001	44999850001	124999750001	244999650001	499999500001	1124999250001
Insertion Sort	2503921057	22500033726	62489124089	122464377656	249931402775	562246099741
Selection Sort	740393104	750103362	8258923407	23246435765	446931412710	67224709954
Quick Sort	1026729	3609352	6335691	8665471	12252113	22301526
Shell Sort	1668928	5475712	9475712	13651424	19951424	30902848
Heap Sort	1674642	5496045	9523826	13687997	20048658	30986477

Table 4: Number of Swap/Assignment operations for sorting algorithms

Graph from tabular data is:



Graph6: Graph for number of Swap of Sorting Algorithm 2 ^N

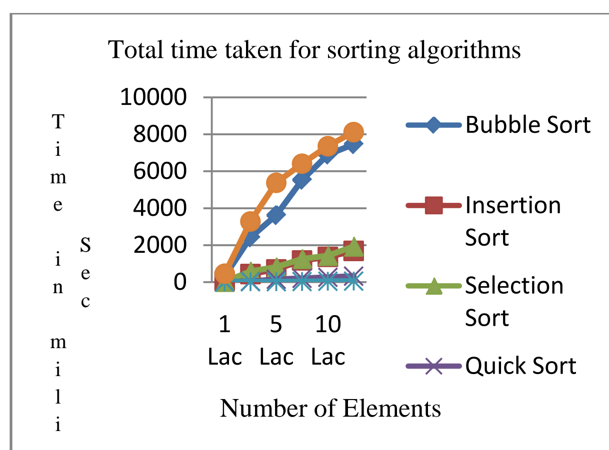


Graph 7: Total number of Swap/Assignment for sorting algorithms

Table for time taken (in mili Seconds) for sorting algorithm on given number of elements:

Algo.	Number of elements					
	1 lacs	3 lacs	5 lacs	7 lacs	10 lacs	15 lacs
Bubble Sort	303	2414	3619	5534	6899	7462
Insertion Sort	42	432	689	1172	1363	1689
Selection Sort	22	583	814	1261	1429	1932
Quick Sort	9	36	148	203	256	321
Shell Sort	13	18	22	26	44	55
Heap Sort	448	3281	5375	6411	7357	8121

Table 5: Total time taken for sorting algorithms



Graph 8: Total time taken for sorting algorithms

Calculate Results

The calculated results (least square fitting) for the above results of Bubble and Quick sorting algorithm are shown on Table and Graph. Here X is number of elements and Y is number of comparisons

1. Bubble Sort

X (No. of elements)	**Y** (No. of Comparison)	X ^2	X ^3	X ^4	Y.X	Y.X ^2
100000	4.9999E+9	1.0E+10	1E+15	1E+20	4.99995E+14	4.99995E+19
300000	4.4998E+10	9E+10	2.7E+16	8.1E+21	1.35E+16	4.04999E+21
500000	1.25E+11	2.5E+11	1.25E+17	6.25E+22	6.24999E+16	3.12499E+22
700000	2.45E+11	4.9E+11	3.43E+17	2.401E+23	1.715E+17	1.2005E+23
1000000	5E+11	1E+12	1E+18	1E+24	5E+17	5E+23
1500000	1.125E+12	2.25E+12	3.375E+18	5.0625E+24	1.6875E+18	2.53125E+24
ΣX=4100000	ΣY=2.045E+12	ΣX2=4.09E+12	Σ X**3**=4.871E+18	ΣX4=6.373E+24	ΣYX=2.4355E+18	Σ YX**2**=3.18665E+24

Table 6: Bubble Sort calculation using Least square fitting method

6a1+4100000a2+4090000000000a3=006 2044997950

004100000 2435497955= 3 000000000a4871000000+ 000a2 4090000000+ 4100000a1

24 + 450409E 3.18664756= 24a3 + 6.3733E + 2 000000000a4871000000+ 000a14090000000

Calculating the values of a1, a2, a3 by using Matrix Inversion Method:

999999 0.49999999= a3 3352761, -0.5000000= a2 250000, 0.98144531= a1

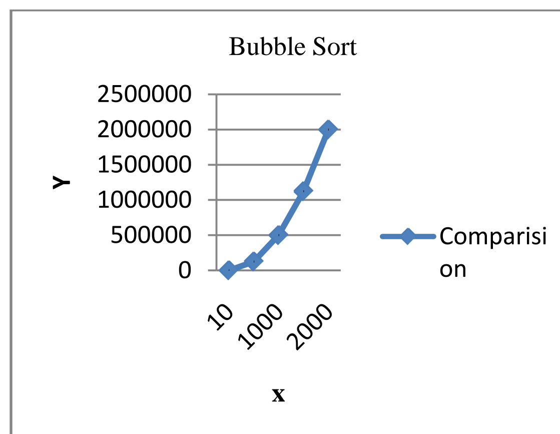
999999x2 0.49999999+ 352761x 0.50000003- 250000 0.98144531= y

Calculated values from the above values are:

X	Y
10	45.981444977223899999999999
500	124750.981428546000000
1000	499500.981411775000000
1500	1124250.981395000000000
2000	1999000.981378220000000

Table 7: Calculated Values of X and Y

Graph from above tabular data is:



Graph 9: Bubble Sort Graph for X-Y values.

2. Quick Sort

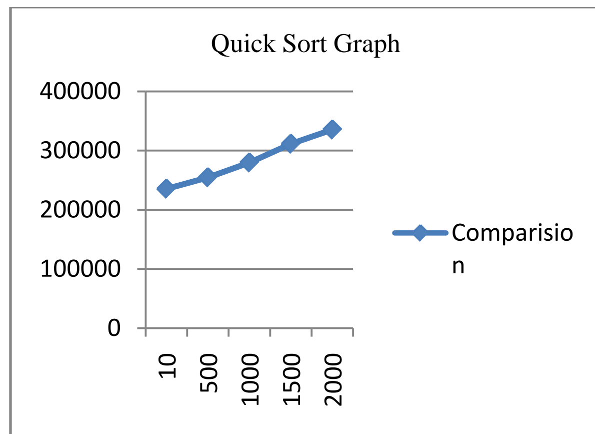
Number of Elements	100000	300000	500000	700000	1000000	1500000
Number of Comparison	2083013	7154514	12439847	18266103	25881883	41478603

12439847 =c + 500000b + 00000a 500000log5
12439847 =C + 500000B + 466208A 9465784.28
25881883 = C + 1000000B +693241A 19931568.5
41478603 = C + 1500000B +05068A 30774796.6
Solving the above equations using Matrix Inversion Method we get the values of A, B and C as:
A=5.7086227916717434
B=-12.6063574003357
C=235321.896
Y=5.70867916717434X.logX-92.606357400272373X+4706433.79167138

X	Y
10	235385.468770355
500	254609.857557621
1000	279606.441907242
1500	311257.710256986
2000	335308.233398483

Table 8: Quick Sort calculation using simple matrix inversion method

IJSER



Graph 10: Quick Sort Graph for X-Y values

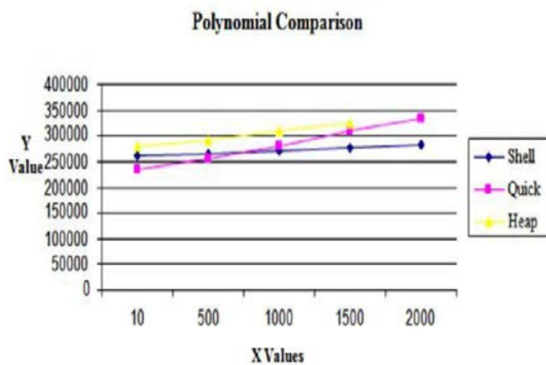
Summary

Based on the above calculations the values calculated can be tabulated as follows:

Sorting Algorithm	A	B	C
Shell Sort	1.2603520355992	-2.909018983	260352.0356
Heap Sort	3.2825049996173	-3.500006479	279490.9996
Quick Sort	5.70862279167174	-12.60635740033	235321.896
Insertion Sort	-9.2744141	95.404689602554	0.24983064989564
Bubble Sort	0.98144531250000	-0.50000003352761	0.49999999999999

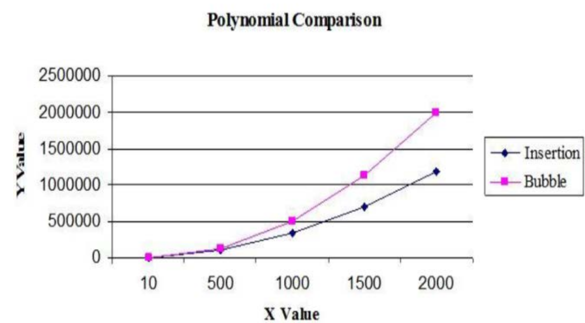
Table9: Summary of calculations of Sorting Algorithms.

Graph showing Polynomial Equations Comparison for algorithms N.log (N). Algorithms:



Graph 11: Polynomial Equation Comparison for algorithm N.log (N)

Graph showing Polynomial Equations Comparison for algorithms: N^2



Graph 12: Polynomial Equation Comparison for N^2 algorithm.

Based on the calculated value of A, B and C the values of X and Y can be tabulated as follows:

Table for calculated no. of Comparisons for sorting algorithm on given number of elements:

Algo.	Number of elements				
	10	**500**	**1000**	**1500**	**2000**
Bubble Sort	45.9814 4497722	124750. 9814285	499500. 9814118	1124250 .981395	1999000 .981378
Insertion Sort	969.755 5469151	110150. 7328611	345226. 0650841	705216. 7222549	1190122 .704374
Quick Sort	235385. 4687704	254609. 8575576	279606. 4419072	311257. 7102570	335308. 2333985
Shell Sort	260364. 81340	264547. 54835	270003. 41313	275934. 98991	282175. 49473
Heap Sort	279565. 04199	292456. 11223	308703. 72986	326190. 30849	474409. 0364815

Table10: No. of Calculated Comparisons using simple matrix inversion method

Further Study

This study was carried out with a single computing device. In the future, researchers could use differentcomputing resources with varying computing speed to compare the effect of processor speed on these datasamples. Also, only integers were used as data sample; it is the interest of the researchers to know what willhappen to character arrays in respect to internal sorting in the future

ISSN 2229-5518

Conclusion

The empirical data obtained by using the program reveals the speed of each algorithm, from fastest to slowest for very large list and ranks as follows:

1. Quicksort
2. Shell Sort
3. Heap sort
4. Insertion sort
5. Selection sort
6. Bubble sort

There is a large difference in the time taken to sort very large lists between the fastest three and the slowest three. This is due to the efficiency of Quick Sort, Shell Sort and Heap sort have over the others when the list to sort is sufficiently large.

References

1. Donald E. Knuth et al. "The Art of Computer Programming," Sorting and Searching Edition 2, Vol.3.
2. Cormen et al. "Introduction to Algorithms," Edition 3, 31 Jul, 2009.
3. Ahmed M. Aliyu, Dr. P. B. Zirra, "A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays," The International Journal Of Engineering And Science (IJES)., ISSN(e): 2319 – 1813 ISSN(p): 2319 – 1805.
4. John Harkins, Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, "Performance and Analysis of Sorting Algorithms on the SRC 6 Reconfigurable Computer," in The George Washington University, 2 Nov. 2005.
5. Wikipedia,(2007) :Sorting Algorithm, Retrieved from http://en.wikipedia.org/wiki/Sorting_algorithm: 24-05-2013 .
6. Wikipedia,(2007) :Selection Sort, Retrieved from http://en.wikipedia.org/wiki/Selection_sort 26-05-2013.
7. Robert L(2002): Data Structures and Algorithms, 2nd Ed.24-28.
8. Kadam, P.K.a.S., Root to Fruit (1): An Evolutionary Study of Sorting Problem. *Oriental Journal Of Computer Science & Technology*, 7(1): p. 111-116 (2014).
9. Astrachan, O., Bubble Sort: An Archaeological Algorithmic Analysis. 2003.

Author Biography

S.Choudaiah
Professor
Department of MCA
Siddharth Institute of Engineering &Technology
Puttur, A.P, India

P.Chandu Chowdary
2nd Year MCA
Siddharth Institute of Engineering &technology
Puttur, A.P India

M.Kavitha
2nd Year MCA
Siddharth Institute of Engineering &technology
Puttur, A.P, India

IJSER