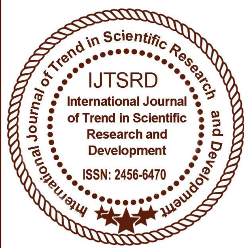


International Journal of Trend in Scientific Research and Development (IJTSRD)

International Open Access Journal
ISSN No: 2456 - 6470 | www.ijtsrd.com | Volume - 2 | Issue – 2

Performance Analysis of Four Different Types of Sorting Algorithms using Different Languages

Dr. I. Lakshmi
Assistant Professor, Department of Computer Science,
Stella Maris College, Chennai, India

ABSTRACT

This research paper presents the different types of sorting algorithms of data structure like quick, heap, insertion and merges and also gives their performance analysis with respect to time complexity. These four algorithms have been an area of focus for a long time but still the question remains the same of "which to use when?" which is the main reason to perform this research. This research provides a detailed study of how all the four algorithms work and then compares them on the basis of various parameters apart from time complexity to reach our conclusion.

***Keywords**: Quick sort, Heap sort, Insertion sort, Merge sort, time complexity, other performance parameters*

I. INTRODUCTION

In the present scenario an algorithm and data structure play a significant role for the implementation and design of any software. In data domain, sorting refers to the operation of arranging numerical data in increasing or decreasing order or non numerical data in alphabetical order [1]. Among quick, heap, insertion and merge it would be interesting to see their worst case complexities which are O(N^2), O(NlogN), O(N^2), O(NlogN) respectively[2]. The efficiency of a sorting algorithm depends on how fast and accurately it sorts a list and also how much space it requires in the memory. Among all, it can be seen that insertion and quick sort perform with the order of n^2 contrast to heap and merge performing with the order of nlogn. On the other hand if we study their space complexity we will find that the quick, heap and insertion have the complexity of the O(1) where as the space complexity of merge sort is O(n)[2]. So to assess the performance of an algorithm [3] the above two parameters are most important in their own.

II. WORKING PROCEDURE OF ALGORITHMS

A. *QUICK SORT:*

This sorting algorithm is based on Divide-and-Conquer paradigm that is the problem of sorting a set is reduced to the problem of sorting two smaller sets. The three step divide and conquer strategy for sorting a typical sub array A[p….r] is as follows:

1) **Divide:** The array A[p….r] is partitioned(rearranged) into two non-empty sub arrays A[p….q] and A[q+1....r] such that each element of A[p….q] is less than or equal to each element of A[q+1….r]. The index of q is completed as part of this partitioning procedure.
2) **Conquer:** The 2 sub arrays A[p….q] and A[q+1….r] are sorted by recursive calls to quick sort procedure[4].
3) **Combine:** Since the sub arrays are sorted in place, no work is headed to combine them, the entire array A[p....r] is now sorted.

The algorithm is divided into two parts. The first part gives a procedure called QUICK, which executes the reduction steps of the algorithm and the second part uses QUICK to sort the entire list.

1) Procedure: QUICK (A,N,BEGIN,END,LOCN)

Here A is an array of N elements. Parameters BEGIN and END contain the boundary values of the sub list of A to which this procedure applies. LOCN keeps the track of the position of the first element A[BEGIN] of the sub list during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned. Steps:

1. [Initialize] set LEFT:=BEGIN, RIGHT:=END, and LOCN:=BEGIN.
2. [Scan from right to left.]
   a. Repeat while A[(LOCN)<=A[RIGHT] and LOCN!=RIGHT: RIGHT :=RIGHT – 1. [End of loop.]
   b. If LOCN=RIGHT, then: return.
   c. If A[LOCN]>A[RIGHT], then:
      i. [Interchange A[LOCN] and A[RIGHT].] TEMP:=A[LOCN),A[LOCN]:=a[RIGHT),a[RIGHT]:=TEMP.
      ii. Set LOCN:=RIGHT.
      iii. iii)Go to Step 3. [End of If structure.]
3. [Scan from left to right.]
   a. Repeat while A[LEFT]<=A[LOCN) and LEFT!=LOCN: LEFT=LEFT+1. [End of Loop.]
   b. If LOCN=LEFT, then: Return. c) If A[LEFT]>A[LOCN], then
      i. [Interchange A[LEFT] and A[LOCN].] TEMP:=A[LOCN], A[LOCN]:=A[LEFT],A[LEFT]:=TEMP. ii) set LOCN:=LEFT. iii) Go to Step ii. [End of if structure.]

2) Algorithm[5]:

The quick sort algorithm sorts an array A with N elements in the following way.

1. [Initialize] TOP:=Null
2. [Push boundary values of A onto stacks when A has 2 or more elements.] If N>1, then TOP: TOP+1, LOWER [1]:=1, UPPER [1]=N.
3. Repeat steps 4 to 7 while TOP! =NULL.
4. [Pop sub lists form stacks.] Set BEGIN: =LOWER [TOP], END: =UPPER [TOP], TOP:=TOP-1.
5. Call QUICK (A, N, BEGIN, END, LOCN).[Procedure]
6. [Push left sub list onto stacks when it has 2 or more elements.] If BEGIN<(LOCN-1),then: TOP:=(TOP+1),LOWER[TOP]:=BEGIN, UPPER[TOP]=(LOCN-1). [end of if structure.]
7. [Push right sub list onto stacks when it has 2 or more elements.] If (LOCN+1)<END, then: TOP:=TOP+1, LOWER[TOP]:= LOCN+1, UPPER[TOP]:=END. [end of if structure.] [end of Step 3 loop.]
8. Exit.

The Figure1 below shows how quick sort algorithm works the elements in the list are: 3, 1, 2, 4, 5, 9, 6, 8, 7 the pivot element here is 5.

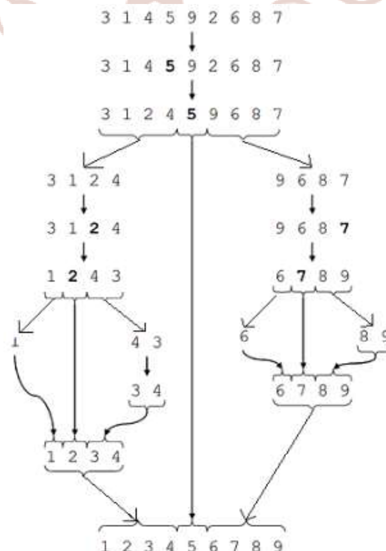


Figure 1: Working of Quick Sort

4) ***Time Complexity Of Quick Sort:*** The running time of a sorting algorithm is usually measured by the number f(n) of comparisons required to sort n elements[6]. The recurrence relation for quick sort is given by:

i. Best-case analysis:

The pivot is in the middle:
T(N) = 2T(N/2) + cN
Dividing by N: T(N) / N = T(N/2) / (N/2) + c
On solving: T(N/2) / (N/2) = T(N/4) / (N/4) + c T(N/4) / (N/4) = T(N/8) / (N/8) + c…… T(2) / 2 = T(1) / (1) + c
Adding all equations: T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + …. + T(2) / 2 = (N/2) / (N/2) + T(N/4) / (N/4) + … + T(1) / (1) + c.logN
After crossing the equal terms: T(N)/N = T(1) + cLogN = 1 + cLogN T(N) = N + NcLogN Therefore T(N) = O(NlogN)

ii. Average case analysis:

Similar computations, resulting in T(N) = O(NlogN)
The average value of T(i) is 1/N times the sum of T(0) through T(N-1) 1/N S T(j), j = 0 thru N-1 T(N) = 2/N (S T(j)) + cN
Multiply by N NT(N) = 2(S T(j)) + cN*N
To remove the summation, we rewrite the equation for
N-1: (N-1)T(N-1) = 2(S T(j)) + c(N-1)2, j = 0 thru N-2 and
Subtract: NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN -c
On solving continuously, rearrange terms, drop the insignificant c:
NT(N) = (N+1)T(N-1) + 2cN
Divide by
N(N+1): T(N)/(N+1) = T(N-1)/N + 2c/(N+1)
On solving:
T(N)/(N+1) = T(N-1)/N + 2c/(N+1) T(N-1)/(N) = T(N-2)/(N-1)+ 2c/(N) T(N-2)/(N-1) = T(N-3)/(N-2) + 2c/(N-1)…. T(2)/3 = T(1)/2 + 2c /3
Add the equations and cross equal terms: T(N)/(N+1) = T(1)/2 +2c S (1/j), j = 3 to N+1 T(N) = (N+1)(1/2 + 2c S(1/j)) The sum S (1/j), j =3 to N-1, is about LogN
Thus T(N) = **O(nlogn)**.

iii. Worst Case Analysis:

This happens when the pivot is the smallest (or the largest) element. T(N) = T(i) + T(N - i -1) + cN T(N) = T(N-1) + cN, N > 1
On continuously solving:
T(N-1) = T(N-2) + c(N-1) T(N-2) = T(N-3) + c(N-2) T(N-3) = T(N-4) + c(N-3) T(2) = T(1) + c.2
Adding all equations we get:
T(N) + T(N-1) + T(N-2) + … + T(2) = T(N-1) + T(N-2) + … + T(2) + T(1) + c(N) + c(N-1) + c(N-2) + … + c.2 T(N) = T(1) + c(2 + 3 + … + N) T(N) = 1 + c(N(N+1)/2 -1)
Therefore T(N) = **O(n2)**

B. HEAP SORT: The heap(binary) data structure is an array object that can be viewed as a complete binary tree as shown in figure1[7]:

Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array B that represents a heap is an object with two attributes: length[B] which is the number of elements in the array and heap-size[B], the number of elements in the heap stored within array B. The root of the tree is B[1] and given the index I of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child. RIGHT(i) can be computed simply: PARENT(i): return └i/2┘ LEFT(i): return 2i

RIGHT(i): return 2i+1; Heaps also satisfy the “heap property” for every node I other than the root, A[PARENT(i)]>=A[i] i.e, the value of a node is at most the value of its parent. Thus, the largest element in a heap is stored at the root, and the sub trees rooted at a node contain smaller values smaller values than does the node itself.

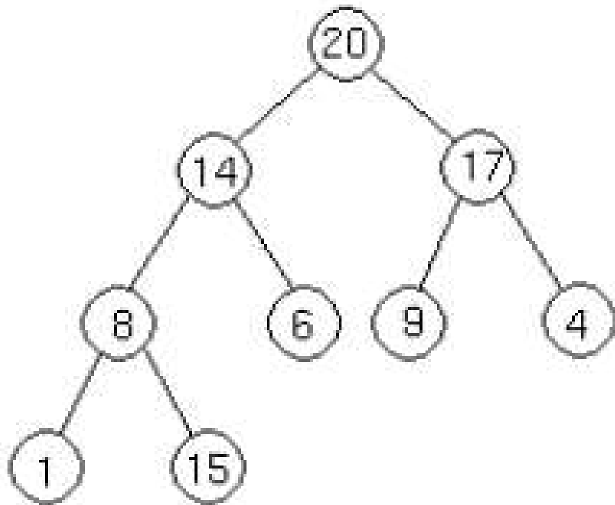


Figure 2: Structure of a heap

1) Algorithm[8]:

The three basic procedures used in the heap sort are:-

a) ***Heapify*:** The function of Heapify is to let the value at B[i] “float down” in the heap so that the sub tree rooted at index I becomes a heap.

Algorithm Heapify(B,i):

1. l➔ LEFT(i)
2. r➔RIGHT(i)
3. If l<=heap-size[B] and B[l]>B[i]
4. Then largest➔l
5. Else largest➔i
6. If r<=heap size[B] and B[r]>B[largest]
7. Then largest➔r
8. If largest! =i
9. Then exchange B[i]➔B[largest]
10. Heapify(B,largest)

b) ***Build-Heap(B):*** goes through the remaining nodes of the tree and runs Heapify on each one. The order in which the nodes are processed guarantees that the sub trees rooted at children at children of a node I are heaps before Heapify is run on that node.

Build-Heap (B)

1. heap-size[B]➔length[A]
2. For i➔ └length [B]/2┘ down to 1.
3. Do Heapify(B,i). Running time of Build-HEAP is O(n).

a. The HEAPSORT procedure (algorithm): starts by using Buil-Heap to build a heap on the i/p array B[1….n], where n=length[B].

Algorithm: *HEAPSORT(B):*

1. Build-Heap (B)
2. For i➔length [B] down to 2
3. Do exchange B[1]➔B[i]
4. heap-size [B]➔heap-size [B]-1.
5. Heapify[B,1]

2) Time Complexity Of Heap Sort[9]- 1. Creation of Heap is the process in which a number of comparisons takes place thereby taking logn time. 2. Denoting depth in form of nodes(n): 20+21+22+…..2d-1=n(no. of nodes) Geometric Progression: Sum=a(rn-1)/r-1 n=1(2d-1)/2-1 log n=log 2d-1/1 log n = log 2d-log 1 d log22=log(n+1) d=log(n+1)=>log n depth of a complete Binary tree=log n for 1 element=> number of comparison takes log n time therefore for n elements time taken for comparisons is T(n)=**O(nlogn).** Time complexity for heap sort in average as well as worst case lies the same i.e T(n)=**O(nlogn).**

C. INSERTION SORT:

This algorithm considers the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm. It builds the sorted sequence one element at a time.

1) Algorithm[11]:

We use a procedure INSERTION_SORT. It takes an array *A*[1.. *n*] as parameter. The array *A* is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time. The algorithm for insertion sort is as follows: INSERTION_SORT (*A*) 1. **FOR** j ← 2 **TO** length[*A*] 2. **DO** key ← *A*[*j*] 3. {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]} 4. *i* ← *j* − 1 5. **WHILE** *i* > 0 and *A*[*i*] > key 6. **DO** *A*[*i* +1] ← *A*[*i*] 7. *i* ← *i* − 1 8. *A*[*i* + 1] ← key

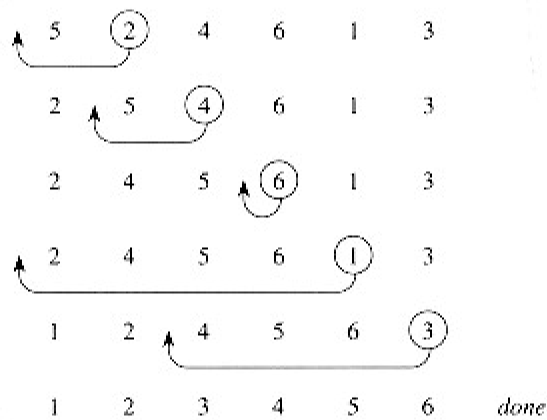


Figure 3 shows the process of insertion sorting

2) Time Complexity of Insertion Sort[10]

Since the running time of an algorithm on a particular input is the number of steps executed, we must define "step" independent of machine. We say that a statement that takes c_i steps to execute and executed n times contributes $c_i * n$ to the total running time of the algorithm. To compute the running time, $T(n)$, we sum the products of the cost and times column. That is, the running time of the algorithm is the sum of running times for each statement executed. So, we have

$T(n) = c_1 n + c_2 (n - 1) + 0 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} (t_j) + c_6 \sum_{2 \le j \le n} (t_j - 1) + c_7 \sum_{2 \le j \le n} (t_j - 1) + c_8 (n - 1)$-----Eq.1 In the above equation we supposed that t_j be the number of times the while-loop (in line 5) is executed for that value of j. Note that the value of j runs from 2 to $(n - 1)$. We have

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} (t_j) + c_6 \sum_{2 \le j \le n} (t_j - 1) + c_7 \sum_{2 \le j \le n} (t_j - 1) + c_8 (n - 1)$ Eq.2

i. Best-Case Analysis[12]:

The best case occurs if the array is already sorted. For each value of $j = 2, 3, ..., n$, we find that $A[i]$ is less than or equal to the key when i has its initial value of $(j - 1)$. In other words, when $i = j - 1$, always find the key $A[i]$ upon the first time the WHILE loop is run. Therefore, $t_j = 1$ for $j = 2, 3, ..., n$ and the best-case running time can be computed using equation (2) as follows:

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} (1) + c_6 \sum_{2 \le j \le n} (1 - 1) + c_7 \sum_{2 \le j \le n} (1 - 1) + c_8 (n - 1)$

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1)$

$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8) n + (c_2 + c_4 + c_5 + c_8)$

This running time can be expressed as $an + b$ for constants a and b that depend on the statement cost c_i. Therefore, $T(n)$ it is a linear function of n. The main concept here is that the while-loop in line 5 executed only once for each j. This happens if given array A is already sorted. $T(n) = an + b = O(n)$ It is a linear function of n.

ii. Worst-Case Analysis[13]:

The worst-case occurs if the array is sorted in reverse order i.e., in decreasing order. In the reverse order, we always find that $A[i]$ is greater than the key in the while-loop test. So, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 .. j - 1]$ and so $t_j = j$ for $j = 2, 3, ..., n$. Equivalently, we can say that since the while-loop exits because i reaches to 0, there is one additional test after $(j - 1)$ tests. Therefore, $t_j = j$ for $j = 2, 3, ..., n$ and the worst-case running time can be computed using equation (2) as follows:

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} (j) + c_6 \sum_{2 \le j \le n} (j - 1) + c_7 \sum_{2 \le j \le n} (j - 1) + c_8 (n - 1)$

And using the summations, we have: $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} [n(n+1)/2 + 1] + c_6 \sum_{2 \le j \le n} [n(n - 1)/2] + c_7 \sum_{2 \le j \le n} [n(n - 1)/2] + c_8 (n - 1)$ $T(n) = (c_5/2 + c_6/2 + c_7/2) n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8) n - (c_2 + c_4 + c_5 + c_8)$

This running time can be expressed as $(an^2 + bn + c)$ for constants a, b, and c that again depends on the statement costs c_i. Therefore, $T(n)$ is a quadratic function of n. Here the main concept is that the worst-case occurs, when line 5 executed j times for each j. This can happens if array A starts out in reverse order $T(n) = an^2 + bn + c = O(n^2)$ It is a quadratic function of n^2.

D. MERGE SORT:

This algorithm is also based on Divide-and-Conquer approach. Given a sequence of elements also called keys c[1],….,c[n], the general idea is to imagine them split into two sets c[1],….c[└n/2┘] and c[└n/2┘ +1],….,c[n].Each set is individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements[9].

*Algorithm [14][15]:*The algorithm is divided into two parts: the first part will be procedures MERGEPASS,

which is used to execute a single pass of the algorithm and the second part will repeatedly apply MERGEPASS until C is sorted.

Algorithm MERGEPASS(C,N,L,D): The N element array A is composed of sorted sub arrays where each sub array has L elements possibly the last sub array, which may have fewer than L elements. The procedure merges the pairs of sub arrays of C and assigns them to the array D. Dividing n by 2*L, we obtain the quotient Q, which tells the number of pairs of L-element sorted sub arrays ; that is Q=INTEGER(N/(2*L))

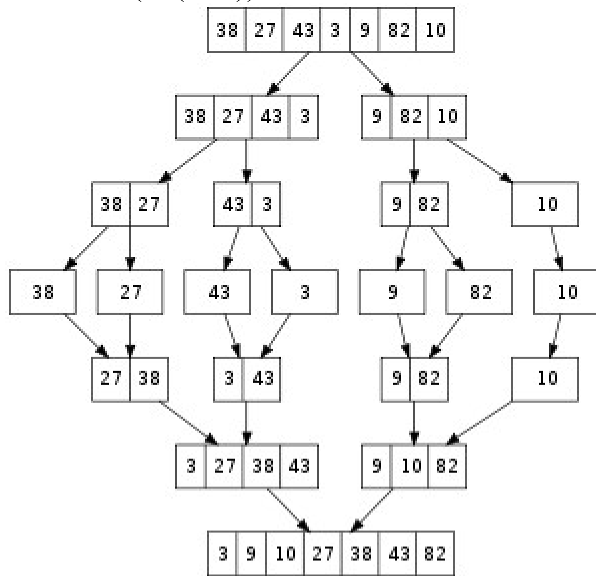


Figure 4 shows the process of merge sorting

1. Set Q= INTEGER(N/(2*L)), S:=2*L*Q(total no. of elements in Q pairs of sub arrays, and R=N-S(no. of remaining elements)
2. Merge the Q pairs of sub arrays.[repeat for J=1,2,….Q:
   a. Set LB(lower bound):=1+(2*J-2)*L.
   b. Call MERGE(C, L, LB, A, L, LB, D, LB). [end of loop.]
3. [only one sub array left?] If R<=L, then: Repeat for j=1,2,….R: Set D(S+J):=C(S+J). [end of loop.]
   Else: call MERGE(C,L,S+1,C,R,L+S+1,B,S+1). [end of if structure.]
4. return.

Algorithm MERGESORT(C,N) This algorithm sorts the N-element array C using an auxiliary array D.

1. set L:=1.[initialize the no. of elements in the sub arrays.]
2. Repeat steps 3 to 6 while L<N:
3. 3.call MERGEPASS(C,N,L,D).
4. 4.call MERGEPASS(D,N,2*L,A).
5. set L:=4*L. [end of step 2 loop.]
6. exit.

2) Time Complexity Of Merge Sort[16][17]:

The recurrence relation for the merge sort is as follows[11]:

2T(n/2)+cn When n=power of 2,n=2k,

solving the above recurrence relation by successive substitution we get: T(n)=2(2T(n/4)+cn/2)+cn = 4T(n/4)+2cn = 4(2T(n/8)+cn/4)+2cn - - - = 2kT(1)+kcn = an+cnlog n 2k<n<2k+1 T(n)<=T(2k+1) Therefore, T(n)=**O(nlogn)** Time complexity for heap sort in average as well as worst case lies the same i.e T(n)=**O(nlogn).**

III. EXPERIMENT AND RESULT TO MEASURE THE PERFORMANCE OF ALGORITHMS

In this experiment we have used C# in which the data set contains random numbers. The initial range of data set starts from 50 to 10000 elements with increment of 100 elements and later the size of elements increased and reached to 30000 with the interval of 1000 elements. Table1 shows this data set and clock tick measurement and the table 2 shows the total time taken by the algorithm in seconds to sort the elements. The table 3 shows the comparative study of their characteristics, time as well as space complexities.

NUMBER OF CLOCK TICKS /NO. OF ELEMENTS	10000	15000	20000	25000	30000
QUICK SORT	Nil	Nil	Nil	Nil	Nil
HEAP SORT	Nil	Nil	Nil	3	3
INSERTION SORT	1	3	5	7	10
MERGE SORT	Nil	Nil	-	-	-

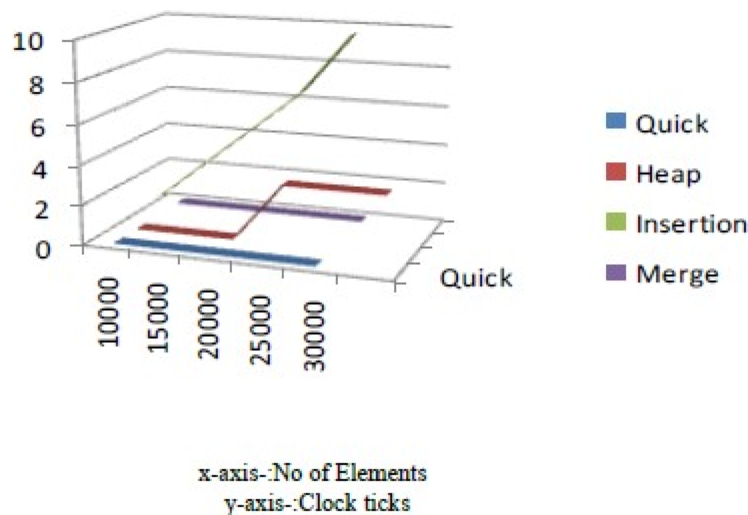
TABLE 1: shows the number of clock ticks taken by the three algorithms for sorting

Sorting Algorithms	10000	15000	20000	25000	30000
Quick	Nil	Nil	Nil	Nil	Nil
Heap	Nil	Nil	Nil	0.164835	0.164835
Insertion Sort	0.054945	0.164835	0.274725	0.384615	0.549451
Merge	Nil	Nil	-	-	-

TABLE 2: shows time taken(in seconds) by the three algorithms to sort the array

	QUICK	HEAP	INSERTION	MERGE
METHOD	Partitioning	Selection	Incremental	Merging
TIME COMPLEXITY BEST AVERAGE WORST	 O(nlogn) O(nlogn) O(n^2)	 O(nlogn) O(nlogn) O(nlogn)	 O(n) O(n^2) O(n^2)	 O(nlogn) O(nlogn) O(nlogn)
SPACE COMPLEXITY	O(1)	O(1)	O(1)	O(N)
STRATEGY	CONCEPT OF PIVOT ELEMENT	CREATES A HEAP OF ELEMENTS	SCAN ALL THE ELEMENTS & DOES SORTING	DIVIDES AN ARRAY INTO TWO SEPARATE LISTS(SUB ARRAYS)
COMPARISON BASED	YES	YES	YES	YES
INPLACE	YES	YES	YES	NO
TYPE	INTERNAL	INTERNAL	INTERNAL	CAN BE BOTH INTERNAL AND EXTERNAL
STABLE	DEPENDS ON ELEMENTS	YES	YES	NO

TABLE 3: shows comparison of the three sorting techniques on various parameters



x-axis-:No of Elements
y-axis-:Clock ticks

Figure 5: Graph comparing all the three algorithms.

IV. CONCLUSION

From the above analysis it can be said that in a list of random numbers from 10000 to 30000, insertion sort takes more time to sort as compare to heap, quick and merge sorting techniques. If we take worst case complexity of all the four sorting techniques then insertion sort and quick sort technique gives the result of the order of N^2, but here if one needs to sort a list in this range then quick sorting technique will be more helpful than the other techniques.

REFERENCES:

1. Data Structures by Seymour Lipschutz and G A Vijayalakshmi Pai (Tata McGraw Hill companies), Indian adapted edition-2006,7 west patel nagar,New Delhi-110063

2. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, fifth Indian printing (Prentice Hall of India private limited), New Delhi-110001
3. Computer Algorithms by Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Galgotia publications,5 Ansari road, Daryaganj, New Delhi-110002
4. C.A.R. Hoare, Quicksort, Computer Journal, Vol. 5, 1, 10-15 (1962)
5. P. Hennequin, Combinatorial analysis of Quick-sort algorithm, RAIRO: Theoretical Informatics and Applications, 23 (1988), pp. 317–333
6. Lecture Notes on Design & Analysis of Algorithms G P Raja Sekhar Department of Mathematics I I T Kharagpur
7. The external Heapsort by L M Wegner, J I Teuhola IEEE Transactions on Software Engineering (1989) Volume: 15, Issue: 7, Pages: 917-925 ISSN: 00985589 DOI: 10.1109/32.29490
8. Heapsort by J. W. J. Williams. Communications of the ACM, Vol. 7, No. 6, pp. 347-348 (June 1964) Heapsort algorithm
9. Worst-case analysis of a generalized heapsort algorithm A. Paulik Institut für Numerische und Angewandte Mathematik, Lotzestrasse 16–18, D-3400 Göttingen, FRG, (science direct.com)
10. Alexandros Agapitos and Simon M. Lucas, “Evolving Efficient Recursive Sorting Algorithms”, 2006 IEEE Congress on Evolutionary Computation Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada July 16- 21, 2006
11. Knuth D. (1997) “The Art of Computer Programming, Volume 3: Sorting and Searching’’, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. pp. 138–141, of Section 5.2.3: Sorting by Selection
12. Let Us C by Yashvant Kanethkar, 8th edition(BPB publications).b-14 Connaught place, New Delhi-110001
13. MERRITT S. M. (1985), “An inverted taxonomy of Sorting Algorithms. Programming Techniques and Data Structures”, Communications of ACM, Vol. 28, Number 1, ACM
14. G. Franceschini. An in-place sorting algorithm performing O(n log n) comparisons and O(n) data moves. In Proc. 44th IEEE Ann. Symp. on Foundations of Computer Science, pages 242–250, 2003
15. Parallel merge sort by Richard Cole 27th Annual Symposium on Foundations of Computer Science sfcs 1986 (1986) Volume: 17, Issue: 4, Publisher: Ieee, Pages: 770-785 ISSN: 02725428 ISBN: 0818607408 DOI: 10.1109/SFCS.1986.41
16. Merge sort:- Merge sort algorithm, C. BronTechnological Univ., Eindhoven, The Netherlands, Communications of the ACM Volume 15 Issue 5, May 1972, ACM New York, NY, USA
17. Optimal stable merging by A. Symvonis. The Computer Journal, Vol. 38, No. 8 (1995). In-place and stable merge sort