Volume : 2 | Issue : 3 | March 2013 ISSN - 2250-1991

Research Paper Engineering

# A Comparison based Analysis of different types of Sorting Algorithms with their Performances

* Ms. Nidhi Chhajed ** Mr. Simarjeet Singh Bhatia

* Assistant Professor, C.S.E Dept. PIES, Indore (M.P)

** Assistant Professor, C.S.E Dept. PIES, Indore (M.P)

ABSTRACT

*This research paper presents the different types of sorting algorithms of data structure like quick, heap and insertion and also gives their performance analysis with respect to time complexity. These three 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 three 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, 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 and insertion it would be interesting to see their worst case complexities which are

O(N^2), O(NlogN), O(N^2), 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 the three it can be seen that quick and insertion sort performs with the order of n^2 contrast to heap performing with the order of nlogn. On the other hand if we study their space complexity we will find that all sorting techniques have the complexity of the O(1). So to assess the performance of an algorithm[3] the above two parameters are most important in their own.

## II. WORKING PROCEDURE OF ALGORITHMS

### 1) 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:

a) 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.

b) Conquer: The 2 sub arrays A[p….q] and A[q+1….r] are sorted by recursive calls to quick sort procedure[4].

c) 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.

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) 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.]
```

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],E
   ND:=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.

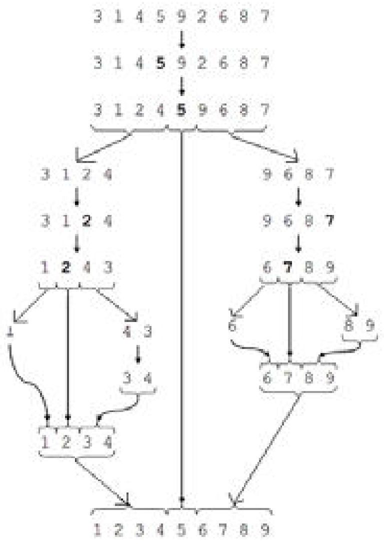


Figure 1: Working of Quick Sort

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:

**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)

**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).

**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)

**2) 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]:

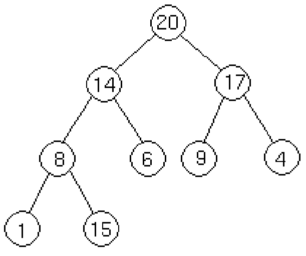


Figure 2: Structure of a heap

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.

**3) 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.

**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
```

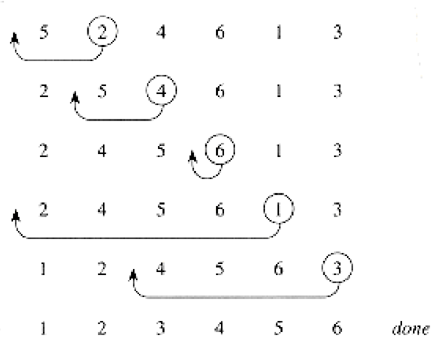


Figure 3 shows the process of insertion sorting

**Time Complexity of Insertion Sort[11]-**
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 ci steps to execute and executed n times contributes ci*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

$T(n) = {}_{c1n} + {}_{c2}(n-1) + 0(n-1) + {}_{c4}(n-1) + {}_{c5}\sum_{2 \le j \le n}(t_j) + {}_{c6}\sum_{2 \le j \le n}(t_j - 1) + {}_{c7}\sum_{2 \le j \le n}(t_j - 1) + {}_{c8}(n-1)$-----Eq.1

In the above equation we supposed that tj 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

$T(n) = {}_{c1n} + {}_{c2}(n-1) + {}_{c4}(n-1) + {}_{c5}\sum_{2 \le j \le n}(t_j) + {}_{c6}\sum_{2 \le j \le n}(t_j - 1) + {}_{c7}\sum_{2 \le j \le n}(t_j - 1) + {}_{c8}(n-1)$ Eq.2

**III. EXPERIMENT AND RESULT TO MEASURE THE PERFORMANCE OF ALGORITHMS**
In this experiment we have used Turbo C++ 3.0 compiler 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 table2 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 |

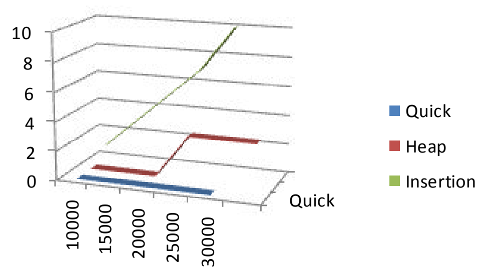
**TABLE 1: shows the number of clock ticks taken by the three algorithms for sorting**

| 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 |

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

| | QUICK | HEAP | INSERTION |
|---|---|---|---|
| METHOD | Partitioning | Selection | Incremental |
| TIME COMPLEXITY<br>BEST<br>AVERAGE<br>WORST | O(nlogn)<br>O(nlogn)<br>O(n^2) | O(nlogn)<br>O(nlogn)<br>O(nlogn) | O(n)<br>O(n^2)<br>O(n^2) |
| SPACE COMPLEXITY | O(1) | O(1) | O(1) |
| STRATEGY | CONCEPT OF PIVOT ELEMENT | CREATES A HEAP OF ELEMENTS | SCAN ALL THE ELEMENTS & DOES SORTING |
| COMPARISON BASED | YES | YES | YES |
| INPLACE | YES | YES | YES |
| TYPE | INTERNAL | INTERNAL | INTERNAL |
| STABLE | DEPENDS ON ELEMENTS | YES | YES |

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



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

Figure 4: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 and quick sorting techniques. If we take worst case complexity of all the three 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. Insertion sort takes more time among all the three techniques and has exponential growth rate as number of elements increases.

**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] 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). | [9] 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. | [10] 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 | [11] Let Us C by Yashvant Kanethkar, 8th edition(BPB publications).b-14 Connaught place, New Delhi-110001. | [12] MERRITT S. M. (1985), "An inverted taxonomy of Sorting Algorithms. Programming Techniques and Data Structures", Communications of ACM, Vol. 28, Number 1, ACM. | [13] 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 |