Performance Evaluation of Sort Algorithm

Sahil Bhardwaj
Dept. of Computer Science and Engineering
Dronacharya College of Engineering, Gurgaon
E-mail: sahilsep6@gmail.com

Sachin Malik
Dept. of Computer Science and Engineering
Dronacharya College of Engineering, Gurgaon
E-mail: maliksachin464@gmail.com

Sahil Munjal
Dept. of Computer Science and Engineering
Dronacharya College of Engineering, Gurgaon
E-mail: sahil5141.munjal@gmail.com

ABSTRACT

This research paper presents performance evaluation of Sort Algorithms like Insertion, Merge Sort and gives their performance analysis with respect to time complexity. These two algorithms has 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 - Heap sort, Insertion sort, time complexity, other performance parameters.

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 insertion and Merge it would be interesting to see their worst case complexities which are O(N^2) and O(NlogN)respectively. 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 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 insertion have the complexity of the O(1) where as the space complexity of merge sort is O(n). So to assess the performance of an algorithm the above two parameters are most important in their own.

WORKING PROCEDURE OF ALGORITHMS

A. 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 :

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

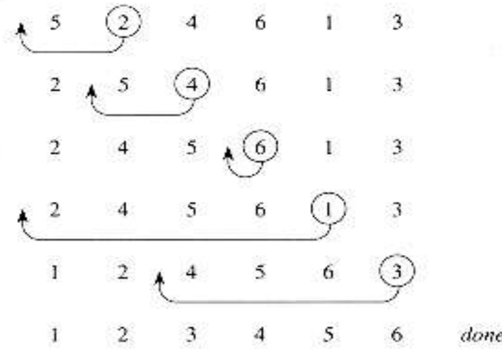

Figure shows the process of insertion sorting

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

$$T(n) = c_1 n + c_2 (n-1) + 0 \ (n-1) + c_4 (n-1) + c_5 \sum_{2 \le j \le n} (t_j) + c_6 \sum_{2 \le j \le n} (t_j - 1) + c_7 \sum_{2 \le j \le n} (t_j - 1) + c_8 (n-1) \text{----- Eq.1}$$

In the above equation we supposed that t 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

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

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. 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.

B. 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.

1) Algorithm : 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)).

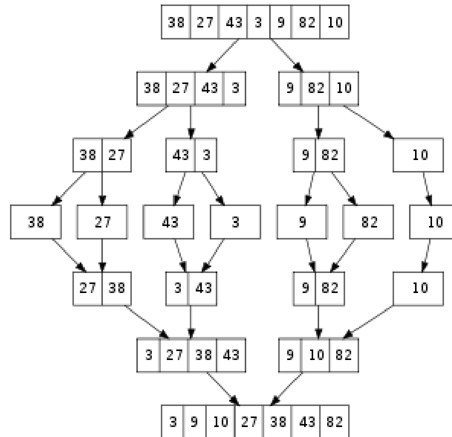

Figure shows the process of merge sorting

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.call MERGEPASS(C,N,L,D).
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:
The recurrence relation for the merge sort is as follows:
2T(n/2)+cn
When n=power of 2,$n=2^k$
, 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
-
-
-
$= 2^k$

=T(1)+kcn
= an+cnlog n
$2^k<n<2^{k+1}$
$T(n)<=T(2^{k+1})$
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 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 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.

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

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

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

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

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

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

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 merge sorting techniques. If we take worst case complexity of all the two sorting techniques then insertion 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 IIT Kharagpur