Name - Dipendu Ghosh
Class - M.Sc Computer Science, 1st Semester
Roll No - 07

Complexity of Sorting Algorithms.

Insertion Sort :- (For n number of data elements)

Best Case :- If the set of data are initially sorted then only one comparison is made on each pass, so the sort is $O(n)$; where n is the number of data in the set.

Average Case :- The total number of comparisons for this case is :-

$$T(n) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2}$$

$$= \frac{n^2}{4} - \frac{n}{4}$$

$$T(n) = O(n^2)$$

Worst Case :- If the set of data are in the reverse order in the initial step then worst case occurs. For each element at index K, the number of comparisions for each element is (K-1). So, the total number of comparisions is :-

$$T(n) = 1 + 2 + \cdots + (n-1)$$

$$= \frac{n(n-1)}{2}$$

$$T(n) = O(n^2)$$

Selection Sort :- (For n number of data elements)

Best Case :-
Average Case :-
Worst Case :-

For selection sort there is no improvement if the set of data elements are sorted or unsorted completely, since the testing proceeds to completion without regard to the arrangement of the data in the set.

For all the cases, for pass 1 it requires $(n-1)$ comparisons, for pass 2 it is $(n-2)$ and so on. So, the total number of comparisions is :-

$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

$$T(n) = O(n^2)$$

Bubble Sort: (For n number of data elements)

Best Case:- If the set of data elements is sorted fully in the initial step, then the complexity is $O(n)$ which is the Best Case.

Average Case and Worst Case:- The time complexity for bubble sort is same for average case and worst case. The worst case occurs when the set of data elements is fully unsorted. So the 1st pass requires $(n-1)$ number of comparisions, 2nd pass requires $(n-2)$ number of comparisions and so on. So the total time is;-

$$T(n) = (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n-1)}{2}$$

$$T(n) = O(n^2)$$

Merge Sort:- (For n number of data elements)

Best Case, Average Case and Worst Case:-

We can write a recurrance relation for the running time of merge sort. We will assume that the number of elements n is a power of 2, so that we always split even halves for n=1, the time to merge sort is constant, which we will denote by 1. Otherwise, the time to merge sort n numbers is equal to the time to do two recursive merge sorts of size $n/2$, plus the time to merge, which is linear. The following equations say this exactly.

$$T(1) = 1 \quad \text{and}$$

$$T(n) = 2T(n/2) + n \qquad — (1)$$

From (1) we get

$$T(n/2) = 2T(n/4) + n/2 \qquad — (2)$$

From (1) and (2) we get

$$T(n) = 4T(n/4) + 2n \qquad — (3)$$

From (1) we get

$$T(n/4) = 2T(n/8) + n/4 \qquad — (4)$$

From (3) and (4) we get

$$T(n) = 8T(n/8) + 3n \qquad — (5)$$

Continuing in this manner we obtain

$$T(n) = 2^k T(n/2^k) + kn \qquad — (6)$$

Using $n/2^k = 1$, we get $n = 2^k$

or, $\log_2 n = K$

From (6) we get

$$T(n) = n T(1) + n \log_2 n$$

Quick Sort :- (For n number of data elements)

Best Case :-

In the best case, the pivot element is in the middle. To simplify the mathematical expression we assume that the 2 sub sets are each exactly half the size of the original. Therefore we can write

$$T(n) = 2\,T(n/2) + cn \qquad [\text{where } c \text{ is a constant}]$$

Dividing both sides of equation by n we get

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$

Using the above equation we get

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$

$$\vdots$$

$$\frac{T(n/2^{k-1})}{(n/2^{k-1})} = \frac{T(n/2^{k})}{n/2^{k}} + c$$

Adding all the above equations we get,

$$\frac{T(n)}{n} = \frac{T(n/2^k)}{n/2^k} + c\,\cancel{(\log_2 n)}\,k$$

Using $n/2^k = 1$, we get

$$n = 2^k$$

$$\text{or, } \log_2 n = k$$

Therefore from the above equation we get,

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c(\log_2 n)$$

$$\text{or, } T(n) = n + c(n\log n) \quad [$$

<u>Average Case and Worst Case</u>:- We will assume that the elements are distinct and that all permutations of the elements are equally likely. Let n be the number of elements in the section of the list being sorted, and let $A(n)$ be the average number of key comparisions done for lists of this size. Suppose the next time the split function is executed first element of the list put in the ith position in this sublist. Split does $(n-1)$ key comparisions and the sublists to be sorted next have $(i-1)$ keys and $(n-i)$ keys respectively.

Each possible position for the split point i is equally likely. (has probability $1/n$). So we have the recurrance relation as:-

$$A(n) = (n-1) + \sum_{i=1}^{n} \frac{1}{n}\left(A(i-1) + A(n-i)\right) \quad \forall\, n \geq 2$$

where $A(1) = A(0) = 0$

Now,
$$A(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}\left(A(i-1) + A(n-i)\right)$$
$$= (n-1) + \frac{1}{n}\sum_{i=1}^{n} A(i-1) + \frac{1}{n}\sum_{i=1}^{n} A(n-i)$$
$$\therefore A(n) = (n-1) + \frac{1}{n}\{A(0) + A(1) + \cdots + A(n-1)\}$$
$$+ \frac{1}{n}\{A(n-1) + A(n-2) + \cdots + A(0)\}$$
$$= (n-1) + \frac{2}{n}\{A(0) + A(1) + \cdots + A(n-1)\}$$
$$= (n-1) + \frac{2}{n}\{A(2) + \cdots + A(n-1)\}$$
$$= (n-1) + \frac{2}{n}\sum_{i=2}^{n-1} A(i) \quad \text{—— (1)}$$
$$A(n-1) = (n-2) + \frac{2}{n-1}\sum_{i=1}^{n-2} A(i) \quad \text{—— (2)}$$

Now, equation (1) ~~with~~ $\times\, n$ $-$ $(n-1)\times$ (2) we get

$$n A(n) - (n-1)A(n-1) = n(n-1) + 2\sum_{i=2}^{n-1} A(i) - (n-1)(n-2) - 2\sum_{i=2}^{n-2} A(i)$$
$$= n(n-1) - (n-1)(n-2) + 2\sum_{i=2}^{n-2} A(i) +$$

$$n\,A(n) = 2(n-1) + 2A(n-1) + (n-1)A(n-1)$$

$$= 2(n-1) + (n+1)A(n-1)$$

dividing by $n(n+1)$ on both sides we get

$$\frac{A(n)}{n+1} = \frac{2n-2}{n(n+1)} + \frac{A(n-1)}{n}$$

Now let $B(n) = \frac{A(n)}{n+1}$

$\therefore$ From previous steps we get $B(n) = \frac{2n-2}{n(n+1)} + B(n-1)$

$$B(n-1) = \frac{2(n-1)-2}{(n-1)(n-1+1)} + B(n-2)$$

$$B(n-2) = \frac{2(n-2)-2}{(n-2)(n-2+1)} + B(n-3)$$

$$\vdots$$

$$B(2) = \frac{2\cdot 2-2}{2(2+1)} + B(1)$$

$\because A(1)=0=A(0)$

$\therefore B(1)=B(0)=0$

$$\therefore B(n) = \frac{2n-2}{n(n+1)} + \frac{2(n-1)-2}{(n-1)(n-1+1)} + \frac{2(n-2)-2}{(n-2)(n-2+1)} + \cdots + B(1)$$

$$\therefore B(n) = \sum_{i=2}^{n} \frac{2i-2}{i(i+1)}$$

$$B(n) \approx 2\ln(n)$$

$$= \ln(2)\log_2 n$$

$$= 2(0.693)\log_2(n)$$

$$= 1.386\log_2 n$$

Now, $\frac{A(n)}{n+1} = 1.386\log_2 n$

or $A(n) = 1.386(n+1)\log_2 n$

Heap Sort:- (For n number of data elements)

Best Case, Average Case, Worst Case:- Let the heapsort algorith. be applied on a set A of n elements. The algorithm has 2 phases.

Phase 1:- Let H be a heap. The number of comparisions to find the appropriate position of a new element ITEM in H cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. The total number T(n) of comparisions to insert n elements in phase 1 into H is bounded as follows:-

$$T(n) \leq n \log_2 n$$

So the running time of Phase 1 of heapsort is proportional to $n\log_2 n$

Phase 2:- Let H is a complete tree with m elements, and Suppose the left and right subtrees of H are heaps and L is the root of H. The reheaping uses 4 comparisions to move the node L one step down the tree H. So the depth of H does not exceed $\log_2 m$, reheaping uses at most $4\log_2 m$ comparisions to find the appropriate place of L in the tree H. This means that the total number h(n) of comparisions to delete the n elements of phase 1 from H, which requires reheaping n times, is bounded as follows:-

$$h(n) \leq 4n \log_2 n$$

So the running time of Phase 2 of heapsort is also proportional to $n\log_2 n$

So the complexity is of order as follows:-

$$T(n) = O(n\log_2 n)$$

Binary Search Tree:-

The number of comparisions required to access a key is 1 more than the number required when the node was inserted. But the number required to insert a key equals the number required in an unsuccessful search. for that key before it was inserted.

Thus $S_n = 1 + \frac{(u_0 + u_1 + u_2 + \cdots + u_{n-1})}{n}$ [where u_i is the time required to insert a node]

Combining this with the equation

$$S_n = \left(\frac{n+1}{n}\right) u_n - 1$$

$$\left(\frac{n+1}{n}\right) u_n - 1 = 1 + \frac{(u_0 + u_1 + \cdots + u_{n-1})}{n}$$

or, $$\frac{(n+1)}{n} u_n = 2 + \frac{(u_0 + u_1 + \cdots + u_{n-1})}{n}$$

or, $$(n+1) u_n = 2n + (u_0 + u_1 + u_2 + \cdots + u_{n-1}) \quad \forall n \quad \text{—(1)}$$

Replacing n by $(n-1)$ we get

$$n u_{n-1} = 2(n-1) + u_0 + u_1 + \cdots + u_{n-2} \quad \text{—(2)}$$

Now, equation (1) − equation (2) we get

$$(n+1) u_n - n u_{n-1} = 2 + u_{n-1}$$

or $$u_n = \frac{2 + u_{n-1}(n+1)}{n+1}$$

$$= \frac{2}{n+1} + u_{n-1}$$

Now $$u_{n-1} = u_{n-2} + \frac{2}{n}$$

$$u_{n-2} = u_{n-3} + \frac{2}{n-1}$$

$\vdots$

$$u_3 = u_2 + \frac{2}{4}$$

$$u_2 = u_1 + 2$$

After adding all the terms we get,

$$u_n = u_1 + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1}$$

$$= 1 + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1} \quad [u_1 = 1]$$

$\therefore$, $S_n = \left(\frac{n+1}{n}\right) u_n - 1$

$$\therefore S_n = \frac{n+1}{n}\left[1 + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1}\right] - 1$$

$$= \frac{2(n+1)}{n}\left[1 + \frac{1}{2} + \frac{2}{3} + \cdots + \frac{1}{n}\right] - 3$$

As n grows large $\left(\frac{n+1}{n}\right)$ is approximately 1, and we have $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is approximately $\log_e n$

Thus for large n S_n may be approximately be

$$2 \ln n - 3$$

$$= 1.3862 \log_2 n - 3$$

$$= O(\log_2 n)$$

This means the tree sort requires $O(\log_2 n)$ time.

Radix Sort:-

Let A be a set of n elements. Let d denote the radix and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1} d_{i2} \cdots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence $C(n)$ the number of comparisions is as follows:-

$$C(n) \leq d \times s \times n$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$ so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. Thus, Radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Shell Sort:-

It has been shown that the order of the Shell sort can be approximated by $O(n(\log n)^2)$ if an appropriate sequence of increments is used. For other ~~cases~~ series ~~by~~ of increments, the running time can be proven to be $O(n^{1.5})$. Empirical data indicates that the running time is of the form $a \times n^b$, where a is between 1.1 and 1.7 and b is approximately 1.26, or of the form $c * n * * (\ln(n))^2 - d * n * \ln(n)$, where c is approximately 0.3 and d is between 1.2 and 1.75. In general the shell sort is recommended for moderately sized data set of several hundred elements.