**Chapter -1 :** **The Role of Algorithms in Computing**

**Algorithm:**

Algorithmis a well-defined computational procedure that takessome value or set of values as inputand produces some value or set of values as output. An algorithm is a sequence of computational steps that transform the input into the output.

**Applications of algorithm:**

* In different types of sorting process
* in genome sequencing for analyzing data
* manipulation of large volume of data
* public key cryptography and digital signatures
* finding the possible shortest rute
* in hash technique
* in searching technique
* foe designing hardware

**Data Structure:**

Data structureis a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

**Technique:**

The techniques which are necessary are applied to the algorithm so that the algorithm gives the expected result.

**Hard Problems:**

Generally, we calculate the efficiency of an algorithm by checking how long time it will take to execute for each case. There are some problems, however, for which no efficient solution is known. This type of problem can be solved by NP complete problem.

**Parallelism:**

Almost every machine or computer has more than one processing core now a days. So, for better efficiency of our algorithm, we should develop our algorithm keeping in mind that in a single second more than one tasks can be completed or executed. This type of algorithm is called multithreaded algorithm which takes the advantages of multiple cores.

**Efficiency:**

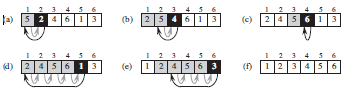
Algorithm’s efficiency depends on how much time it will take to be executed. For example, Insertion sort takes n^2 time to sort n items, while merge sort takes nlogn times to sort n items. Which means merge sort is faster than insertion sort. The efficiency also depends upon the computer architecture. Suppose, we want to sort 10 million numbers in two different computers which have different clock speed . 1st computer execute 10 billions instructions per second and 2nd computer execute 10 millions instructions per second . If 1st computer uses insertion sort algorithm ,it takes 20000 seconds where as 2nd computer takes approximately 1100 seconds using merge sort algorithm. So, 2nd computer is 18 times faster than 1st computer.

**Chapter -2 : Getting Started**

**Insertion Sort:**

Insertion sort is a sorting algorithm which works fast and perfectly for a small number

of items. It's complexity is n^2



**Algorithm of insertion sort:**

1 for j = 2 to A.length

2 key = A[j]

3 // Insert A[j] into the sorted sequence A[1 .. j - 1]

4 i = j - 1

5 while i > 0 and A[i] > key

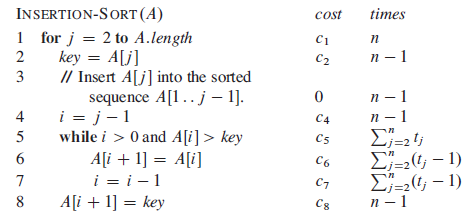
6 A[i + 1] = A[i]

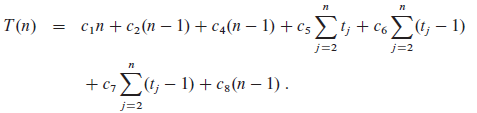
7 i = i - 1

8 A[i + 1] = key

**Analysis of Insertion sort:**

Generally, the size of the input increases with the time taken by an algorithm which means it takes longer time to sort one thousand numbers than four numbers. Suppose, the cost of all the statement in the algorithm is c and the number of items are n. The cost of 1st statement will be c1 & the cost of 2nd statement will be c2 & so on. According to the Insertion sort algorithm, the 1st statement will the executed for n times, the 2nd statement will be executed for n -1 times and so on. the statement will be executed for the summation of of t j times from j=2 to n and the followings:



So the total running time will be : 

**Worst case & average case analysis:**

Worst case will happen only when the input array is already sorted in descending order and best

case will happen only when the input array is already sorted in ascending order. The running time of best case is linear function an+b and the worst case is the quadratic function

.

To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. To insert the

second to last element, we need at most n−2 comparisons and at most n−2 swaps, and so on. The

number of operations needed to perform insertion sort is therefore: 2\*(1+2+....+n-2+n-1). To

calculate the recurrence relation for this algorithm, use the following summation:

p

Σ = {(p(p+1))/2}.

q=1

It follows that

{2(n-1)(n-1+1)/2}=n(n-1).

Use the master theorem to solve this recurrence for the running time. As expected, the

algorithm's complexity is O(n ^2 ).O(n^2 )

When analyzing algorithms, the average case often has the same complexity as the worst case.

So insertion sort, on average, takes O(n^2 ) time

**The divide and Conquer approach:**

**Divide:** The main problem is divided into a number of some problems that is smaller distance of

the main problem.

**Conquer:** Solving the problems using recursive manner is called the conquering.

**Combine:** Lastly the recursive solutions are combined into the solutions of the original

problem. Merge sort algorithm is the example of divide and conquer approach.

the main array is divided into several similar sub-arrays.The sub-arrays are solved using

recursive method . After after solving the sub-array, their combined to get the solution of the

main problem.

**Chapter-3: Growth Of Functions**

**Asymptotic notation**: Asymptotic notation describes the algorithm efficiency and performance in a meaningful way. It describes the behaviour of time or space complexity for large instance characteristics.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

**Θ-notation:**

The Θ-notation asymptotically bounds a function from above and below. When we have

only an asymptotic tight bound, we use Θ-notation.

**O-notation:**

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

**Ω-notation:**

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

**o-notation:**

We use o-notation to denote an upper bound that is not asymptotically tight. The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.

**ω-notation:**

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation

to denote a lower bound that is not asymptotically tight.