ALGORITHMS

Definition: An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem or to a different problem by breaking the problem into simple cases.

It must possess the following properties:

1. **Finiteness:** An algorithm should terminate in a finite number of steps.
2. **Definiteness:** Each step of the algorithm must be precisely (clearly) stated.
3. **Effectiveness:** Each step must be effective. i.e.; it should be easily convertible into program statement and can be performed exactly in a finite amount of time.
4. **Generality:** Algorithm should be complete in itself, so that it can be used to solve all problems of given type for any input data.
5. **Input/output:** Each algorithm must take zero, one or more quantities as input data and gives one of more output values.

An algorithm can be written in English like sentences or in any standard representations. The algorithm written in English language is called Pseudo code.

Example: To find the average of 3 numbers, the algorithm is as shown below.

Step1: Read the numbers a, b, c, and d.
Step2: Compute the sum of a, b, and c.
Step3: Divide the sum by 3.
Step4: Store the result in variable of d.
Step5: End the program.

Development Of An Algorithm

The steps involved in the development of an algorithm are as follows:

- Specifying the problem statement.
- Designing an algorithm.
- Coding.
- Debugging
- Testing and Validating
- Documentation and Maintenance.

Specifying the problem statement: The problem which has to be implemented in to a program must be thoroughly understood before the program is written. Problem must be analyzed to determine the input and output requirements of the program.

Designing an Algorithm: Once the problem is cleared then a solution method for solving the problem has to be analyzed. There may be several methods available for obtaining the required solution. The best suitable method is designing an Algorithm. To improve the **clarity** and **understandability** of the program flowcharts are drawn using algorithms.

Coding: The actual program is written in the required programming language with the help of information depicted in flowcharts and algorithms.

Debugging: There is a possibility of occurrence of errors in program. These errors must be removed for proper working of programs. The process of checking the errors in the program is known as _Debugging'.

There are three types of errors in the program.

Syntactic Errors: They occur due to wrong usage of syntax for the statements.

Ex: x=a*%b

Here two operators are used in between two operands. Runtime Errors : They are determined at the execution time of the program

Ex: Divide by zero

Range out of bounds.

Logical Errors : They occur due to incorrect usage of instructions in the program. They are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs.

Testing and Validating: Once the program is written , it must be tested and then validated. i.e., to check whether the program is producing correct results or not for different values of input.

Documentation and Maintenance: Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references. Maintenance is the process of upgrading the program, according to the changing requirements.

PERFORMANCE ANALYSIS

When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities.

- The **time complexity** of an algorithm is a function of the running time of the algorithm.
- The **space complexity** is a function of the space required by it to run to completion.
- The time complexity is therefore given in terms of **frequency count**.
- Frequency count is basically a count denoting number of times a statement execution

Asymptotic Notations:

- To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.
- Using asymptotic notations we can give time complexity as ―fastest possible‖, ―slowest possible‖ or ―average time‖.
- Various notations such as Ω, θ, O used are called asymptotic notions.

Big Oh Notation

Big Oh notation denoted by ‗O' is a method of representing the upper bound of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that C > 0 and for all integers $n > n_0$, $f(n) \leq c*g(n)$, then

f(n) = Og(n).

Various meanings associated with big-oh are

O(1)	constant computing time
O(n)	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
O(logn)	logarithmic

The relationship among these computing time is
$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n)$

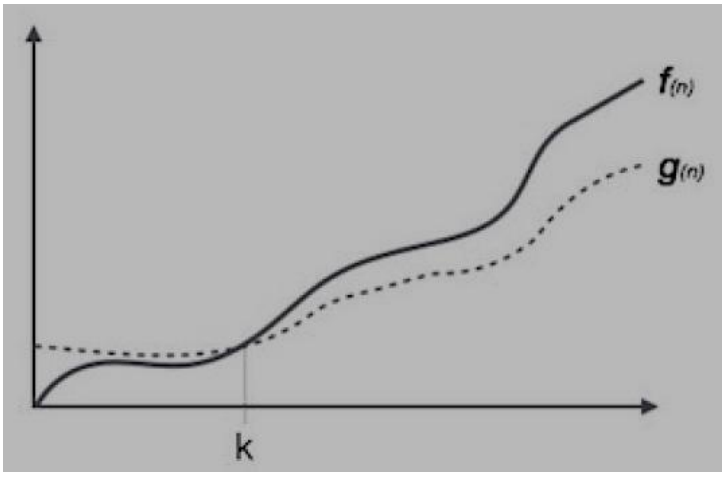
Omega Notation:-

Omega notation denoted ‗Ω' is a method of representing the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm to complete.

Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that C > 0 and for all integers $n > n_0$, $f(n) > c*g(n)$, then

$f(n) = \Omega\, g(n)$.

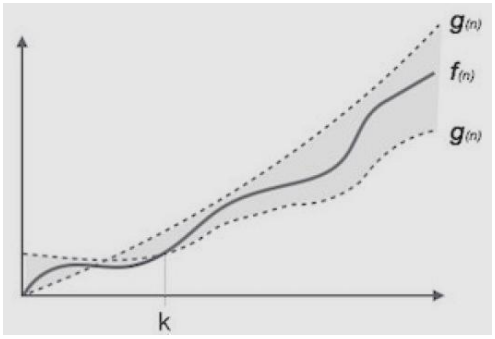


Theta Notation:-

Theta notation denoted as _θ‘ is a method of representing running time between upper bound and lower bound.

Definition:

Let, f(n) and g(n) are two non-negative functions. There exists positive constants C_1 and C_2 such that $C_1\ g(n) \le f(n) \le C_2\ g(n)$ and $f(n) = \theta\ g(n)$



How to compute time complexity

1	Algorithm Message(n)	0
2	{	0
3	for i=1 to n do	n+1
4	{	0
5	write(—Hello‖);	n
6	}	0
7	}	0
	total frequency count	2n+1

While computing the time complexity we will neglect all the constants, hence ignoring 2 and 1 we will get n. Hence the time complexity becomes O(n).

f(n) = Og(n).
i.e f(n)=O(2n+1)
=O(n) // ignore constants

1	Algorithm add(A,B,m,n)	0
2	{	0
3	for i=1 to m do	m+1
4	for j=1 to n do	m(n+1)
5	C[i,j] = A[i,j]+B[i,j]	mn

6	}	0
	total frequency count	2mn+2m+1

f(n) = Og(n).

=> O(2mn+2m+1)// when m=n;

= $O(2n^2+2n+1)$; By neglecting the constants,

we get the time complexity as $O(n^2)$.

The maximum degree of the polynomial has to be considered.

Best Case, Worst Case and Average Case Analysis

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** complexity.
- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case time** complexity.
- The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.

Space Complexity: The space complexity can be defined as amount of memory required by an algorithm to run.

Let p be an algorithm,To compute the space complexity we use two factors: constant and instance characteristics. The space requirement S(p) can be given as

S(p) = C + Sp

where C is a constant i.e.. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers.

- Sp is a space dependent upon instance characteristics. This is a variable part whose space
  requirement depend on particular problem instance.

Eg:1

```
Algorithm add(a,b,c)
{return a+b+c;
}
```

If we assume a, b, c occupy one word size then total size comes to be 3

S(p) = C

Eg:2

```
Algorithm add(x,n)
{
        sum=0.0;
        for i= 1 to n do
        sum:=sum+x[i];
        return sum;
}
```

$S(p) \geq (n+3)$

The n space required for x[], one space for n, one for i, and one for sum

Searching: Searching is the technique of finding desired data items that has been stored

within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods. The appropriate search algorithm often depends on the data structure being searched.

Search algorithms can be classified based on their mechanism of searching. They are

- Linear searching
- Binary searching

Linear or Sequential searching: Linear Search is the most natural searching method and It is very simple but very poor in performance at times .In this method, the searching begins with searching every element of the list till the required record is found. The elements in the list may be in any order. i.e. sorted or unsorted.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends and position of the element is returned. Otherwise, we will move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list by returning position as - 1.

For example consider the following list of elements.

55 95 75 85 11 25 65 45

Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 55. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons at position 4 starting from position 0.

Linear search can be implemented in two ways.i)Non recursive ii)recursive

Algorithm for Linear search

```
Linear_Search (A[ ], N, val , pos )
Step 1 : Set pos = -1 and k = 0
Step 2 : Repeat while k < N
          Begin
Step 3 :  if A[ k ] = val
                Set pos = k
                print pos
                Goto step 5
          End while
Step 4 : print ―Value is not present‖
Step 5 : Exit
```

BINARY SEARCHING

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub- array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Before applying binary searching, the list of items should be sorted in ascending or descending order.

Best case time complexity is O(1)

Worst case time complexity is O(log n)

Algorithm:

```
Binary_Search (A [ ], U_bound, VAL)
Step 1 : set BEG = 0 , END = U_bound , POS = -1
Step 2 : Repeat while (BEG <= END )
Step 3 :     set MID = ( BEG + END ) / 2
Step 4 :     if A [ MID ] == VAL then
                POS = MID
                print VAL — is available at —, POS
                GoTo Step 6
             End if
             if A [ MID ] > VAL then
               set END = MID – 1
             Else
               set BEG = MID + 1
             End if
          End while
Step 5 :  if POS = -1 then
             print VAL — is not present —
          End if
Step 6 : EXIT
```

SORTING

Arranging the elements in a list either in ascending or descending order. various sorting algorithms are

- Bubble sort
- selection sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort

BUBBLE SORT

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming i.e. takes more number of comparisons to sort a list . In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

ALGORITHM:

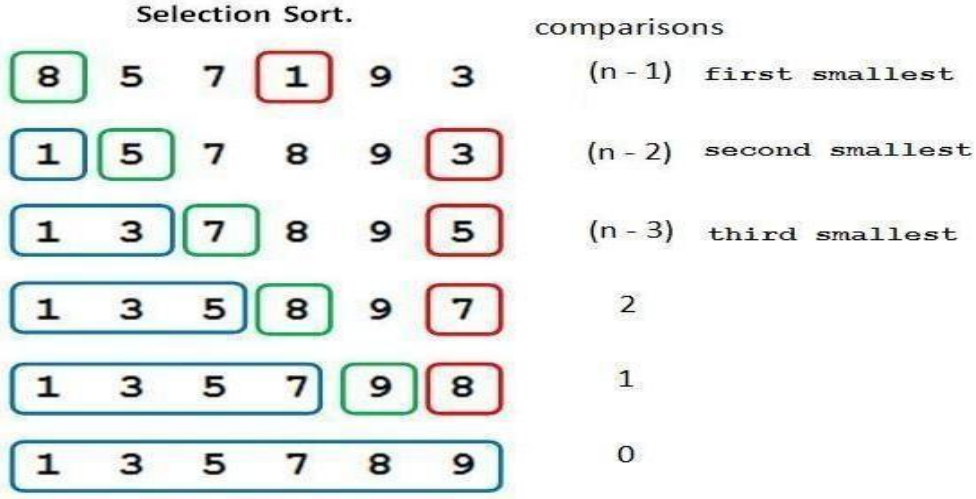
Bubble_Sort (A [ ] , N)

Step 1: Start
Step 2: Take an array of n elements
Step 3: for i=0,…..... n-2
Step 4: for j=i+1,…….n-1
Step 5: if arr[j]>arr[j+1] then
Interchange arr[j] and arr[j+1]
End of if
Step 6: Print the sorted array arr
Step 7:Stop

SELECTION SORT

selection sort:- Selection sort (Select the smallest and Exchange):

The first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position.Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.



INSERTION SORT

Insertion sort: It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

ALGORITHM:

Step 1: start
Step 2: for i ← 1 to length(A)
Step 3: j ← i
Step 4: while j > 0 and A[j-1] > A[j]
Step 5: swap A[j] and A[j-1]
Step 6: j ← j - 1
Step 7: end while
Step 8: end for
Step9: stop

QUICK SORT

Quick sort: It is a divide and conquer algorithm. Developed by Tony Hoare in 1959. Quick sort

first divides a large array into two smaller sub-arrays: the low elements and the high elements.
Quick sort can then recursively sort the sub-arrays.
ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

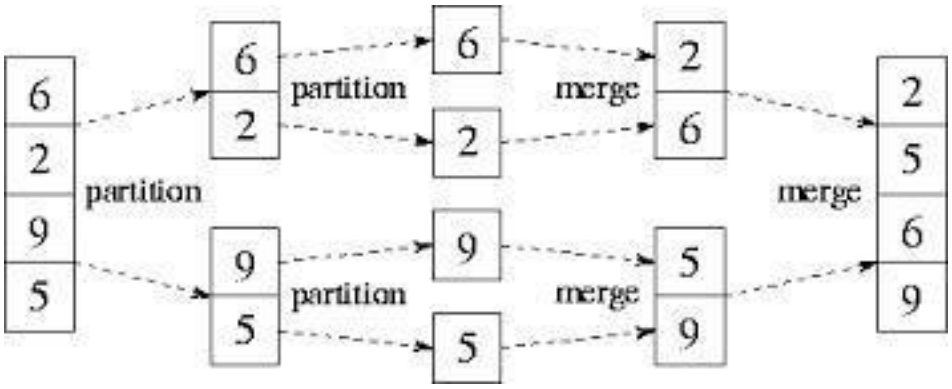
Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

MERGE SORT

Merge sort is a sorting technique based on divide and conquer technique. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast, and has a time complexity of O(n log n).

C**onceptually, merge sort works as follows:**

1. Divide the unsorted list into two sub lists of about half the size.
2. Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. Merge the two sub lists back into one sorted list.




```
int main()
{
int n,i;
int list[30];
        cout<<"enter no of elements\n";
        cin>>n;
        cout<<"enter "<<n<<" numbers ";
        for(i=0;i<n;i++)
        cin>>list[i];
        mergesort (list,0,n-1);
        cout<<" after sorting\n";
        for(i=0;i<n;i++)
        cout<<list[i]<<‖\t‖;
return 0;
}
```

RUN 1:

```
enter no of elements 5
enter 5 numbers 44 33 55 11 -1
  after sorting -1 11 33 44 55
```

HEAP SORT

It is a completely binary tree with the property that a parent is always greater than or equal to either of its children (if they exist). first the heap (max or min) is created using binary tree and then heap is sorted using priority queue.

Steps Followed:

a) Start with just one element. One element will always satisfy heap property.
b) Insert next elements and make this heap.
c) Repeat step b, until all elements are included in the heap.

a) Exchange the root and last element in the heap.
b) Make this heap again, but this time do not include the last node.
c) Repeat steps a and b until there is no element left.

Algorithm	Worst case	Average case	Best case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Linear search	O(n)	O(n)	O(1)
Binary search	O(log n)	O(log n)	O(1)