Divide and conquer :- This breaks down the algorithm to solve the problem in different methods. It allows you to break down problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

Greedy algorithm :- It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting best solution. It is easy to implement and has faster execution time. But there are very rare cases in which it provides the optimal solution.

The major categories of algorithms are given below:

Sort :- Algorithm developed for sorting the items in a certain order.

Search :- Algorithm developed for searching the items inside a data structure.

Delete :- Algorithm developed for deleting the existing element from the data structure.

Insert :- Algorithm developed for inserting an item inside a data structure.

Update :- Algorithm developed for updating the existing element inside a data structure.

Algorithm Analysis :-

The algorithm can be analyzed in two levels i.e. first is before creating the algorithm, and second is after creating the algorithm.

There are two analysis of an algorithm.

Priori Analysis :-

Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.

Posterior Analysis :-

Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing algorithm using any programming language.

Algorithm complexity :-

The performance of the algorithm can be measured in two factors :

Time complexity :-

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation.

Here big O notation is the asymptotic notation to represent time complexity. The time complexity is mainly calculated by counting the number of steps to finish execution.

```
sum = 0 ;
// suppose we have to calculate the sum of n
 numbers.
for i=1 to n
sum = sum + i ;
// when the loop ends then sum holds the sum
of n numbers.
return sum;
```

In above code, the time complexity of the loop statement will be atleast n, and if value of n increases, then time complexity also increases.

We generally consider the worst-time complexity as it is maximum time taken for any given input size.

Space complexity :-

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. similar to the time complexity, space complexity is also expressed in big o notation.

Space complexity = Auxiliary space + Input size.

The following are the types of algorithms :

Search Algorithm :-

on each day, we search for something in our day to day life.

similarly, with the case of computer, huge data is stored in a computer that whenever user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search data in an array :

- Linear search
- Binary search

sorting Algorithms :-

sorting algorithms are used to rearrange elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements :

Asymptotic Analysis :-

The time required by an algorithm comes under three types :

Worst case :- It defines the input for which the algorithm takes a huge time.

Average case :- It takes average time for the program execution.

Best case :- It defines the input for which the algorithm takes the lowest time.

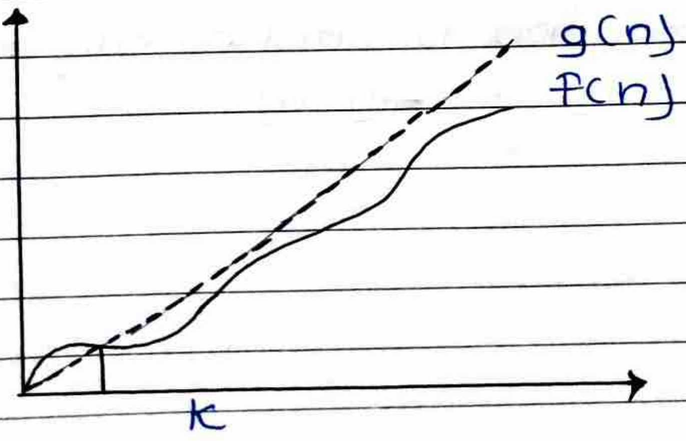
Asymptotic Notations :-

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below :

1). Big oh notation (O) :-

This measures the performance of an algorithm by simply providing the order of growth of the function.

This notation provides an upper bound on a function which ensures that function never grows faster than the upper bound.



TOPPER
WORLD

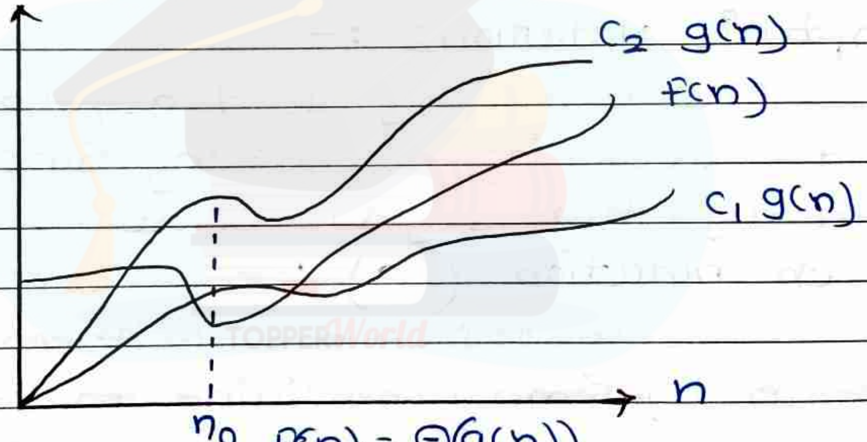
Example :- If f(n) and g(n) are two functions defined for positive integer,

then $f(n) = O\ g(n)$ as f(n) is big oh of g(n) or f(n) is on order of g(n)) if there exists constants c and no such that :

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

2). Omega Notation (Ω) :-

It basically describes best case scenario which is opposite to big o notation. It is the formal way to represent lower bound of an algorithm's running time.



$f(n) = \Theta(g(n))$

Example :- let f(n) and g(n) be functions of n where n is steps required to execute program:

$$f(n) = \Theta\ g(n)$$

The above condition is satisfied only if when:

$$c1. g(n) <= f(n) <= c2.g(n)$$

TOPPER WORLD

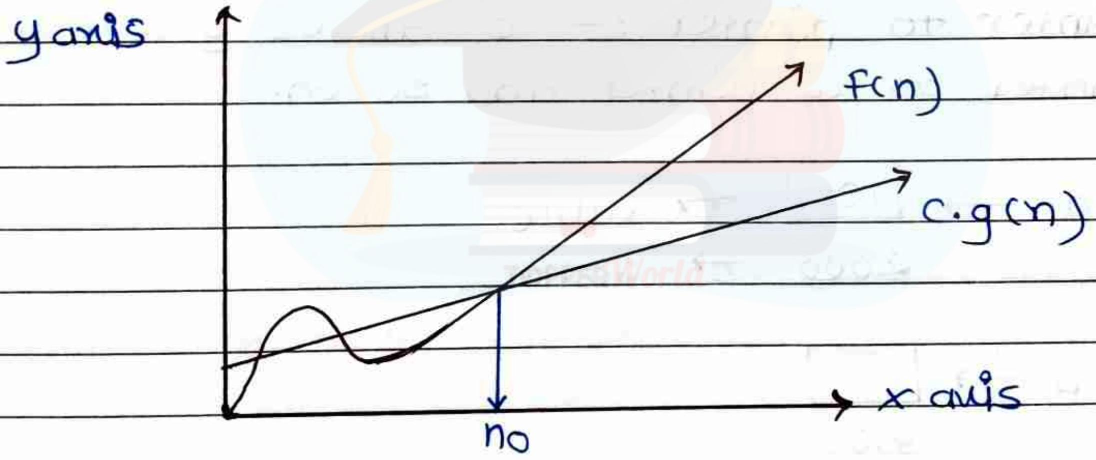
2). omega Notation (Ω)

It basically describes best-case scenario which is opposite to big-o notation It is formal way to represent lower bound to an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or best case time complexity.

Example :- If f(n) and g(n) are two functions defined for positive integers,

then $f(n) = \Omega\, g(n)$ as f(n) is omega of g(n) or f(n) is on the order of g(n) if there exists constants c and no such that :

$f(n) >= c \cdot g(n)$ for all $n \geq n_0$ and $c > 0$



3). Theta Notation (Θ)

The theta notation mainly describes average case scenarios.

It represents realistic time complexity of an algorithm. Big theta is mainly used when the value of worst-case and best case is same.

TOPPER WORLD

Pointer :-

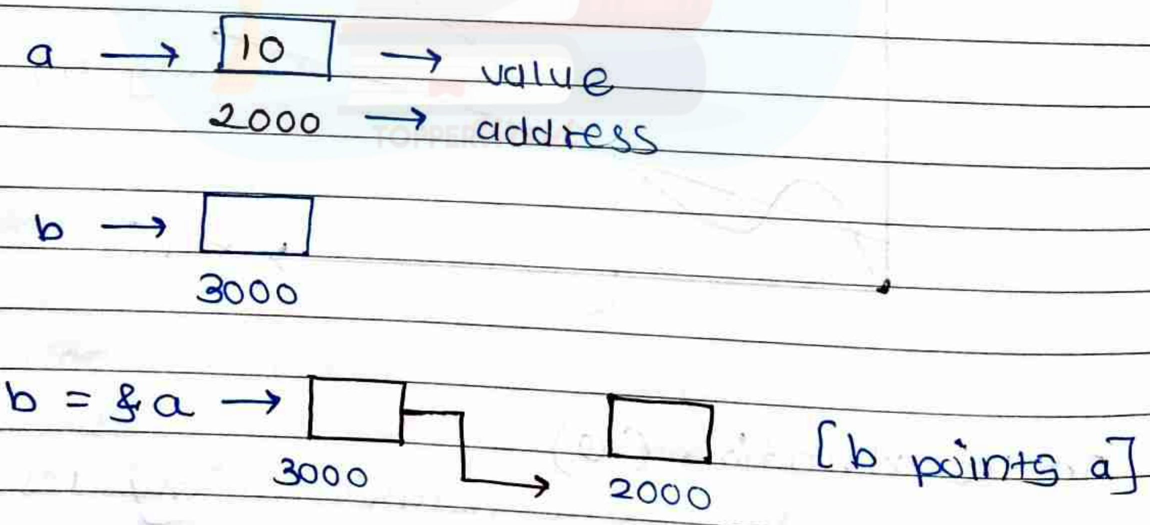
Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at location is known as dereferencing pointer.

Pointer arithmatic :-

4 arithmatic operators that can be used in pointers : ++, --, +, -

Array of pointers :- You can define array of to hold a number of pointers.

Pointer to pointer :- C allows you to have pointer on a pointer and so on.



Program
Pointer →

```
#include < stdio.h>
int main ()
```

```
{
 int a = 5;
 int *b;
 b = &a;
 printf ("value of a = %d \n", a);
 printf ("value of a = %d \n", *(&a));
 printf ("value of a = %d \n", *b);
 printf ("address of a = %u \n", &a);
 printf ("address of a = %d \n", b);
 printf ("address of b = %u \n", &b)
 printf ("value of b = address of a = %u", b);
 return 0;
}
```

output →

```
value of a = 5
value of a = 5
address of a = 3010494292
address of a = -1284473004
address of b = 3010494296
value of b = address of a = 3010494292.
```

Program :-
pointer to pointer :-

```
#include <stdio.h>
int main ()
 {
  int a = 5;
  int *b;
  int **c;
b = &a;
c = &b;
printf ("value of a = %d \n", a);
printf ("value of b = address of a = %u \n", b);
printf ("value of c = address of b = %u \n", c);
printf (" address of b= %u \n", c);
printf (" address of c = %u \n", &c);
return 0;
}
```

output →

```
value of a = 5
value of b = address of a = 2831685116
value of c = address of b = 2831685120
address of b = 2831685120
address of c = 2831685128.
```

Structure :-

A **structure** is a composite data type that defines a grouped list of variables that are to be placed under one name in block of memory.

Program :-
structure →

```
struct structure_name
 {
  data-type member 1;
  data - type member 2;
```