INTRODUCTION

➢**Algorithm :** Step by step procedure to solve a problem is called Algorithm. The steps should be unambiguous.

➢An algorithm is an efficient method that can be expressed within finite amount of Time and space.

➢To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Algorithms are never written to support a particular programming code.
-

Problem – Design an algorithm to add two numbers and display the result.

Step 1 – START

Step 2 – declare three integers **a**, **b** & **c**

Step 3 – define values of **a** & **b**

Step 4 – add values of **a** & **b**

Step 5 – store output of step 4 to **c**

Step 6 – print **c**

Step 7 – STOP

Algorithm vs Program

Algorithm	Program
1. Design phase	1. Implementation phase
2. Any language. Eg: English	2. Programming language. Eg: C, C++, Java, Python.
3. Does not depend on Hardware and OS.	3. Depends on Hardware and OS.

PROPERTIES OF ALGORITHM

TO EVALUATE AN ALGORITHM, WE HAVE TO SATISFY THE FOLLOWING CRITERIA:

1.INPUT: The Algorithm should be given zero or more input.

2.OUTPUT: At least one quantity is produced. For each input the algorithm produced value from specific task.

3.DEFINITENESS: Each instruction is clear and unambiguous.

4.FINITENESS: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5.EFFECTIVENESS: Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

ALGORITHM to code

Step-1: start

Step-2: Read a,b,c

Step-3:if a>b then go to step 4

otherwise go to step 5

Step-4:if a>c then

print a is largest otherwise

print c is largest

Step-5: if b>c then

print b is largest otherwise

print c is largest

step-6: stop

```java
public static void main(String[] args) {
    // Step 2: Read a, b, c
    Scanner scanner = new Scanner(System.in);
    int a = scanner.nextInt();
    int b = scanner.nextInt();
    int c = scanner.nextInt();

    // Step 3: Check if a > b
    if (a > b) {
        // Step 4: Check if a > c
        if (a > c) {
            System.out.println(x:"a is the largest");
        } else {
            System.out.println(x:"c is the largest");
        }
    } else {
        // Step 5: Check if b > c
        if (b > c) {
            System.out.println(x:"b is the largest");
        } else {
            System.out.println(x:"c is the largest");
        }
    }
}
```

Need of Algorithm

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.

PERFORMANCE ANALYSIS

Performance Analysis: An algorithm is said to be efficient and fast if it take less time to execute and consumes less memory space at run time is called Performance Analysis.

1. **SPACE COMPLEXITY:** The space complexity of an algorithm is the amount of Memory Space required by an algorithm during course of execution is called space complexity .
2. **TIME COMPLEXITY:** The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Time Complexity can be calculated by using Two types of methods:

- Step Count Method
- Asymptotic Notation.

Frequency count method (Time complexity analysis)

It works by counting the number of times each basic instruction within the algorithm is executed.

- The number of times that a comment executes is 0.
- In if-else statements the if statement is executed one time but the else statement will be executed zero or one time.
- A typical for($i = 1; i \leq n; i++$) statement will be executed "n+1" times for the first n times the condition is satisfied and the inner loop will be executed and for the (n+1)th time the condition is failed and the loop terminates.

Frequency count method (Time complexity analysis)

```
Sum(arr, n)
{
    int ans= 0;
    for(int i = 0; i < n; i++) {
            ans+= arr[i];
    }
    return ans;
}
```

- int ans= 0 is an initialization statement and executes 1 time.
- for(i = 0;i < n ; i++) is a loop and it takes n+1 times .
    - i= 0 execute 1 time
    - i < n execute n+1 times. Extra 1 for checking i = = n
    - i ++ execute n times
- ans+= arr[i] statement and takes n times.
- return ans and takes 1 time.
- Total Number of times it is executed is= 1 + (n+1) +n +1

Frequency count method (Time complexity analysis)

```
int n=4;
for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
        System.out.println(i+" "+j);
    }
}
```

- int n= 4 is an initialization statement and executes 1 time.
- for(int i=1; i<=n; i++)is a loop and it takes n+1 times .
- for(int j=1;j<=n;j++) executes n*(n+1) times
- System.out.println(i+" "+j) executes n*n times
- Total Number of times it is executed is= 1+ (n+1) + (n*n) + n + (n*n)= (2*n*n) + 2*n + 2

Usually, the time required by an algorithm falls under three types:

1.Worst-case: (usually)

- T(n) = maximum time of algorithm on any input of size n.

2.Average-case: (sometimes)

- T(n) = expected time of algorithm over all inputs of size n.

3.Best-case:

- T(n) = minimum time of algorithm on any input of size n.

COMPLEXITY: Complexity refers to the rate at which the storage or time grows as a function of the problem size.

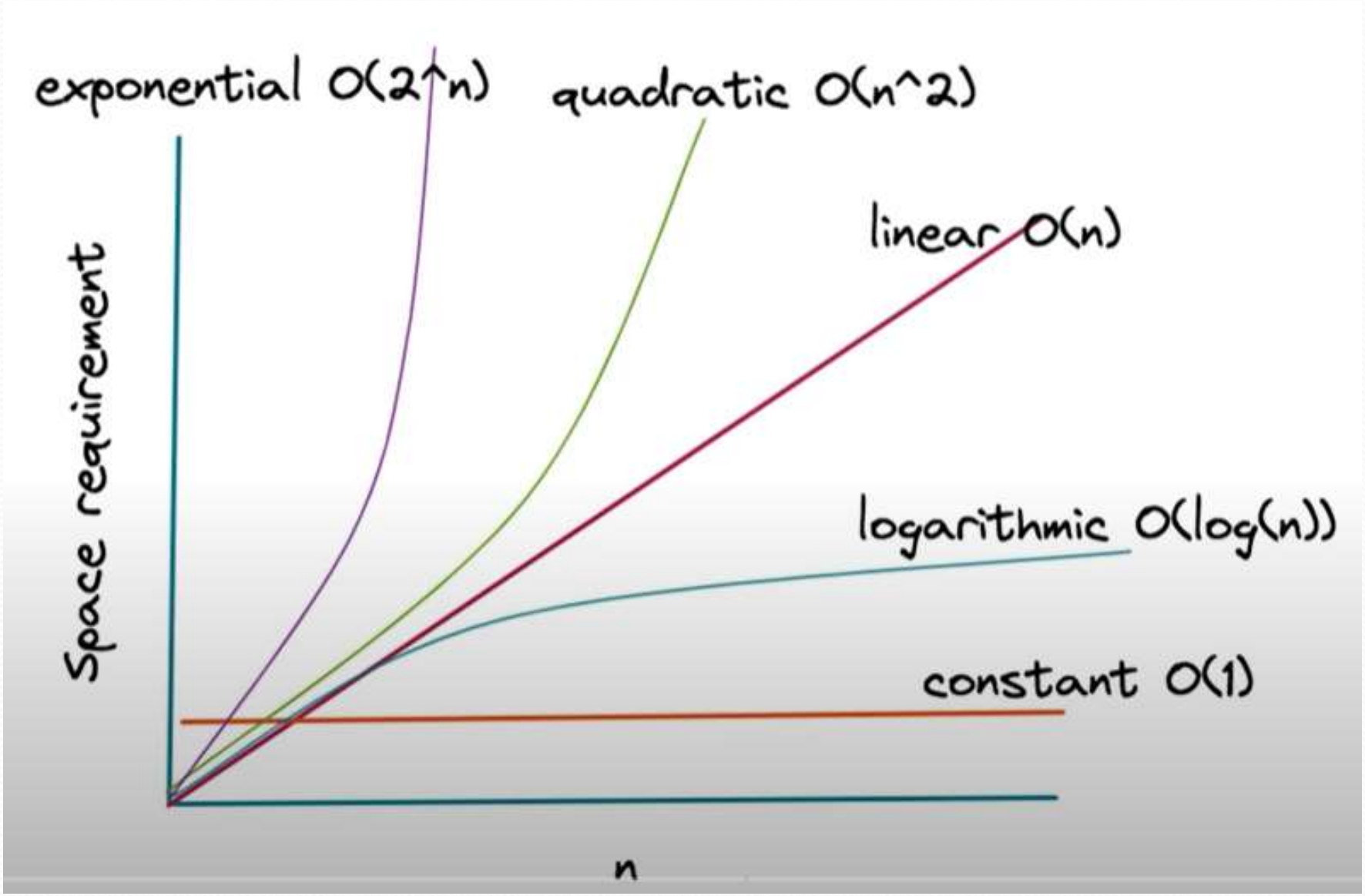
ASYMPTOTIC NOTATION

ASYMPTOTIC NOTATION: Asymptotic notations describe the behaviour of algorithm and determine the rate of growth of a function.

Rate of growth: The time taken by algorithm increases with increase in size of input.

Time complexity of an algorithm is represented by **f(n)**, where **n** is the input size.

F(n)= (2*n*n) + 2*n + 1


exponential O(2^n)
quadratic O(n^2)
linear O(n)
logarithmic O(log(n))
constant O(1)
Space requirement
n

They are 3 asymptotic notations are mostly used to represent time complexity of algorithm.

1.Big oh (O)notation

2.Big omega (Ω) notation

3.Theta(Θ) notation

Example: Suppose you are making a building:

1. Builder gives you exact cost.
2. Builder gives you maximum cost that might be required.
3. Builder gives minimum cost that will be required.

1.Big oh (O) notation

1.Big oh (O)notation

- represent upper bound of algorithm run time.
- calculate maximum amount of time of execution or worst case time complexity.

Given two functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there exist constants **c > 0** and $\mathbf{n_0}$ **>= 0** such that **f(n) <= c*g(n)** for all **n >=** $\mathbf{n_0}$.

F(n)= (2*n*n) + 2*n + 1 is time.

G(n)= n, n*n, Log(n), n*log(n), etc…., the function which is inside big-oh.

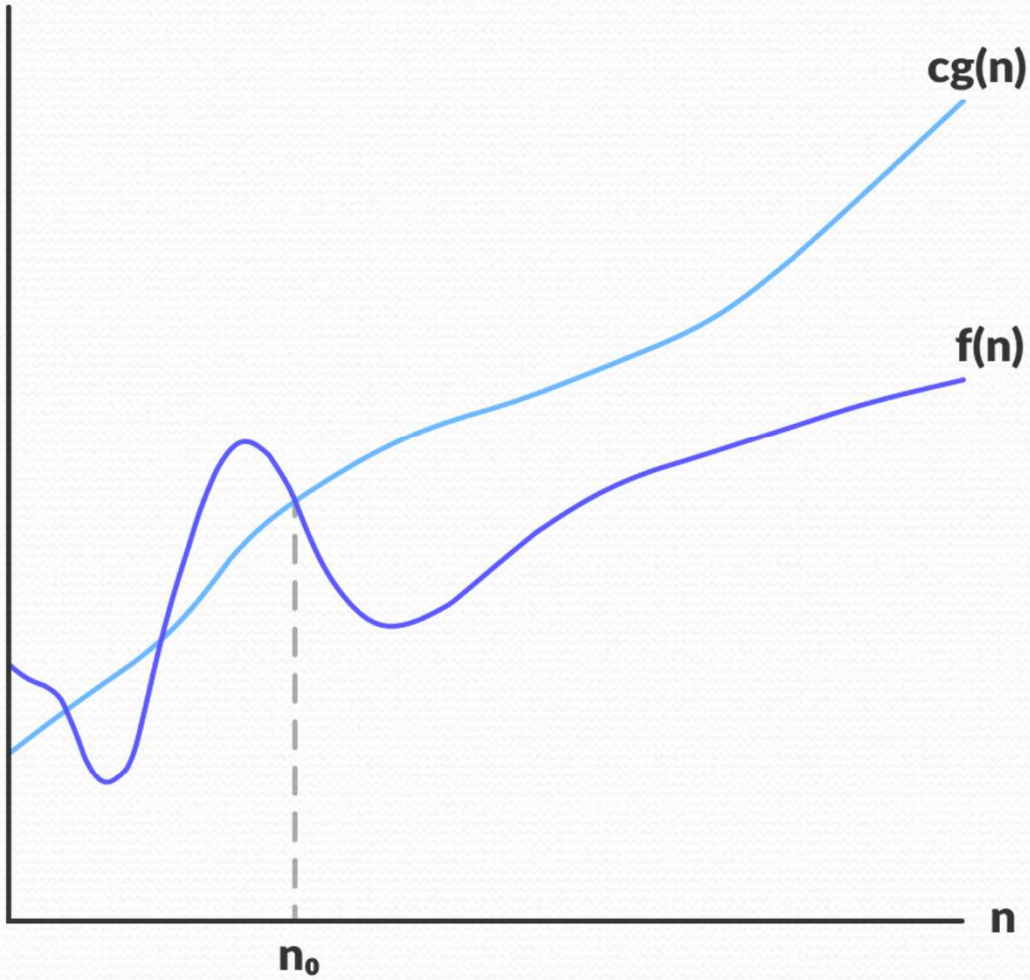
1.Big oh (O)notation

F(n)= $(2*n*n) + 2*n + 1$ is time.

G(n)= n, n*n, Log(n), n*log(n), etc…., the function which is inside big-oh.

Now f(n) is difficult to plot and also difficult to analyse.

Now g(n) are well known functions and easier to analyse and plot.



f(n) = O(g(n))

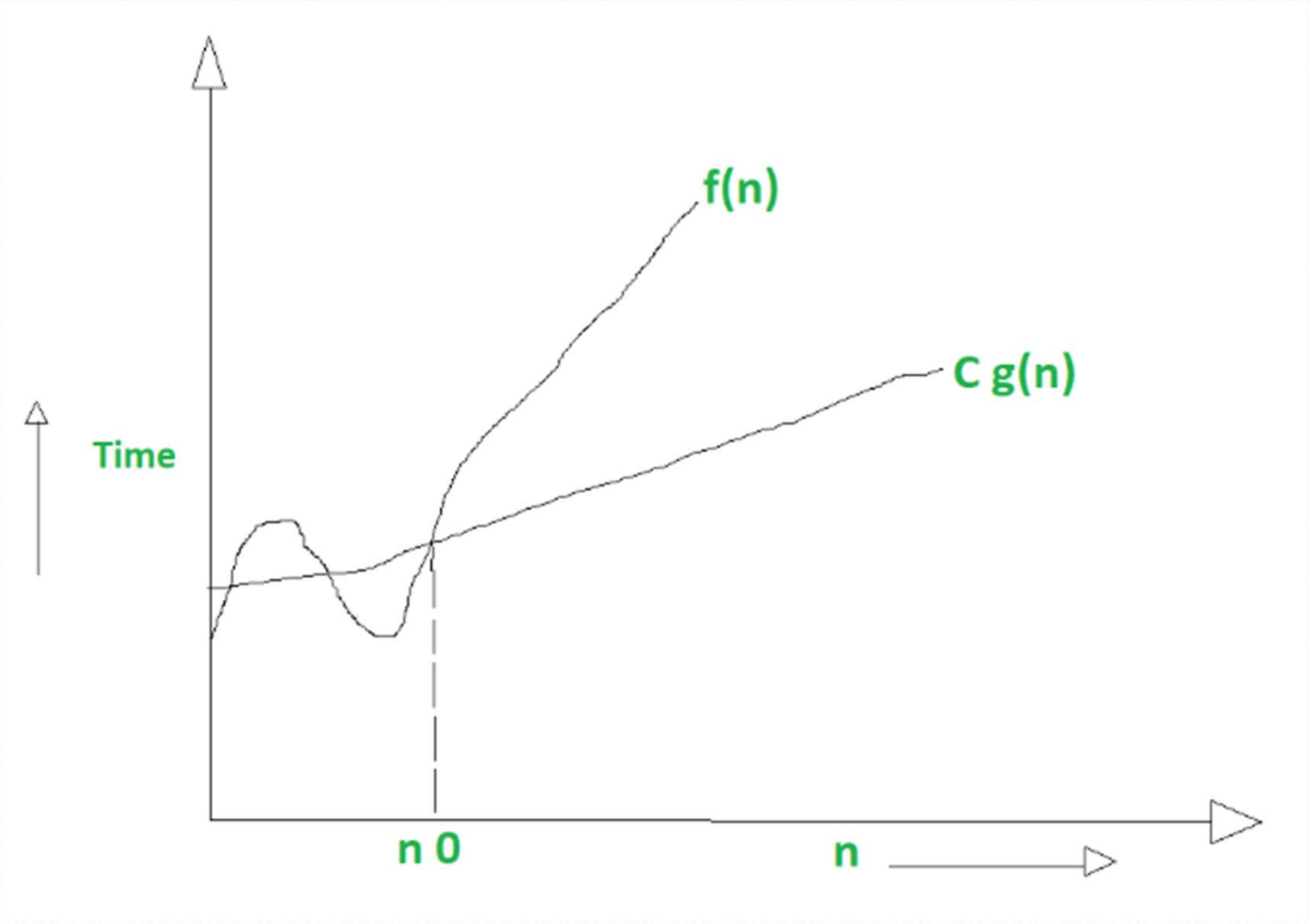
2.Ω-Omega notation

Ω-Omega notation : It represent Lower bound of algorithm run time. By using Big Omega notation we can calculate minimum amount of time. We can say that it is best case time complexity.

Formula : $f(n)>=c\ g(n)$ $n>=n_o$, $c>0$, $n_o>=0$

where c is constant, n is function

- Lower bound
- Best case


f(n)
C g(n)
Time
n 0
n

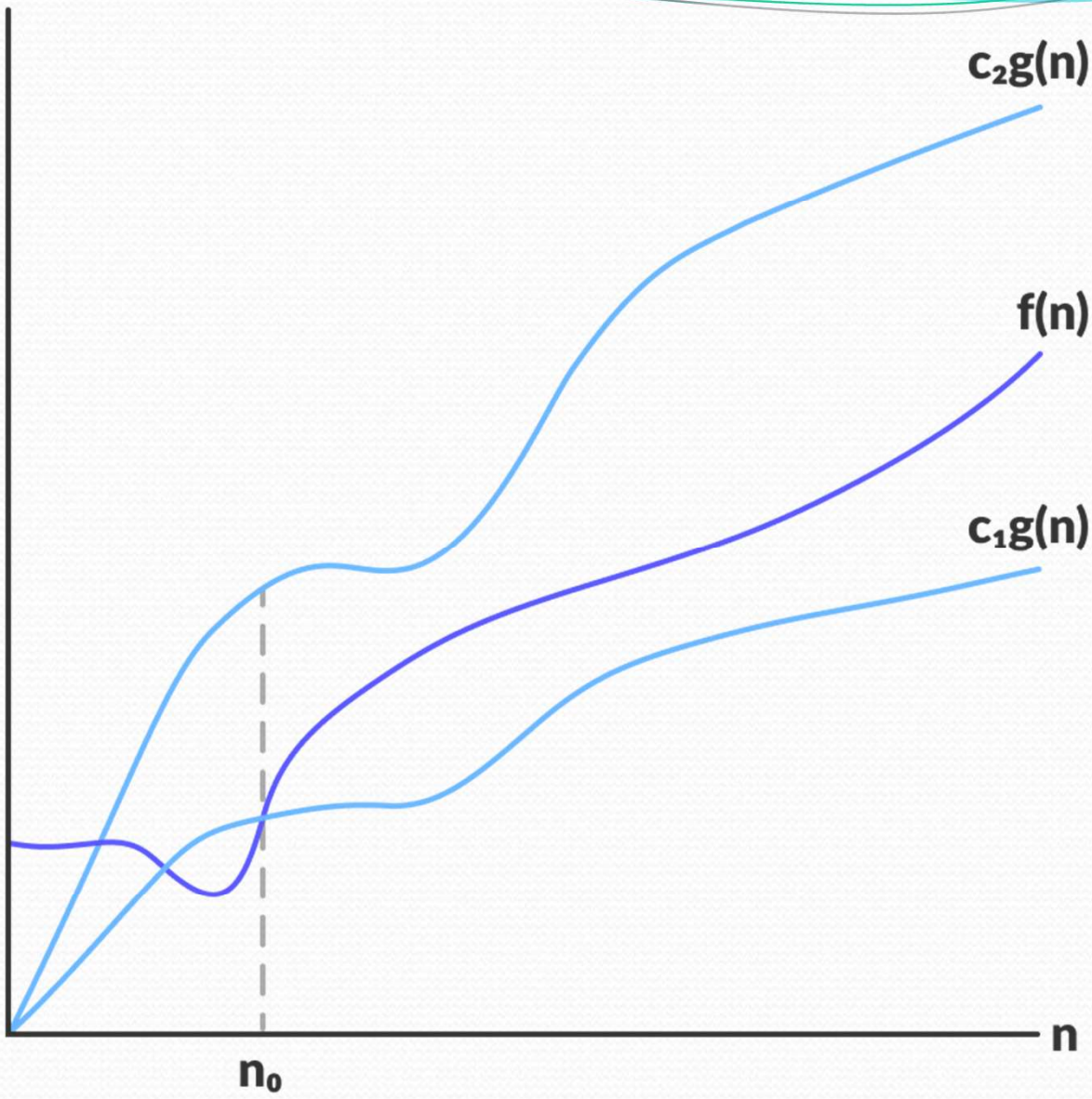
3.Θ -Theta notation

Theta (Θ) notation : It represent average bond of algorithm running time. By using theta notation we can calculate average amount of time.

Formula : $c_1\ g(n) <= f(n) <= c_2\ g(n)$

where c is constant, n is function

- Average bound



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

Example of asymptotic notation

Problem:-Find upper bond ,lower bond & tight bond range for functions: f(n)= 2n+5

Solution:-Let us given that f(n)= 2n+5 , now g(n)= n

lower bond=2n, upper bond =3n, tight bond=2n

For Big –oh notation(O):- according to definition

f(n)<=cg(n) for Big oh we use upper bond so

f(n)=2n+5, g(n)=n and c=3 according to definition

2n+5<=3n

Put n=1 7<=3 false Put n=2 9<=6 false Put n=3 14<=9 false

Put n=4 13<=12 false Put n=5 15<=15 true

now for all value of n>=5 above condition is satisfied. C=3 n>=5

2. Big - omega notation :- f(n)>=c.g(n) we know that this Notation is lower bond notation so c=2

Let f(n)=2n+5 & g(n)=2.n

Now 2n+5>=c.g(n);

2n+5>=2n put n=1

We get 7>=2 true for all value of n>=1,c=2 condition is satisfied.

3. Theta notation :- according to definition

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

Properties to remember:

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n)), (g_1(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\, f_2(n)$ is in $O(g_1(n)\, g_1(n))$.

Prove these properties. Might come in exam. Don't prove it simply taking some examples.

If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.

$f(n) = O(k \times g(n))$, $k > 0$.

$\Rightarrow f(n) <= c \cdot k \cdot g(n)$,

Def: $f(n) = O(g(n))$

$\Rightarrow f(n) <= c \cdot g(n)$.

$\Rightarrow f(n) <= K \cdot g(n)$.

$\Rightarrow f(n) = O(g(n))$

C*k= K, where c,k,K are constants

If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\ f_2(n)$ is in $O(g_1(n)\ g_1(n))$.

To prove:

$f_1(n) \cdot f_2(n) = O(g_1(n)\ g_2(n))$

$f_1(n) \cdot f_2(n) <= C \cdot g_1(n) \cdot g_2(n)$

Given:

$\equiv f1(n) = O(g_1(n))$, $f2(n) = O(g_2(n))$

$\Rightarrow f1(n) <= c_1 \cdot g_1(n)$, $f2(n) <= c_2 \cdot g_2(n)$

Multiply

$f1(n) \cdot f2(n) <= c_1\ g_1(n) \cdot c_2\ g_2(n)$

$f_1(n) \cdot f2(n) <= C \times g_1(n) \cdot g_2(n)$

$\Rightarrow \quad f_1(n) \cdot f_2(n) \equiv O(g_1(n) \cdot g_2(n))$