Data Structures & Algorithms

MODULE 1: GROWTH OF FUNCTIONS

TOPIC: Overview and Importance of Algorithms

INTRODUCTION

➢The word algorithm comes from the name of the author -Abu Jafar Mohammed Ibn Musa Al khowarizmi who wrote A text book entitled-"Algorithmi de numero indorum"

Now term "Algorithmi" in the title of the book led to the term Algorithm.

➢An algorithm is an *effective method* for *finding out the solution* for a given problem. It is a sequence of instruction that conveys the method to address a problem

➢**Algorithm :** Step by step procedure to solve a computational problem is called Algorithm.

INTRODUCTION

- An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.

- An algorithm can be expressed within finite amount of Time and space.

- The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum 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.

PROPERTIES OF ALGORITHM

To Evaluate an Algorithm we have to Satisfy the following Criteria:

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

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 (CONTD…)

- A well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output.*

- Written in a pseudo code which can be implemented in the language of programmer's choice.

PSEUDO CODE: A notation resembling a simplified programming language, used in program design.

How To Write an Algorithm

Step-1:start
Step-2:Read a,b,c
Step-3:if a>b
if a>c
print a is largest
else
if b>c
print b is largest
else
print c is largest
Step-4 : stop

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

Differences

Algorithm	Program
1.At design phase	1.At Implementation phase
2.Natural language	2.written in any programming language
3.Person should have Domain knowledge	3.Programmer
4.Analyze	4.Testing

ALGORITHM SPECIFICATION

Algorithm can be described (Represent) in four ways.

1.Natural language like English: (no ambiguity)

When this way is chooses, care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart:

This method will work well when the algorithm is small & simple.

3. Pseudo-code Method:

In this method, we should typically describe algorithms as program, which resembles language like Pascal & Algol(Algorithmic Language).

4. Programming Language:

we have to use programming language to write algorithms like C, C++, JAVA etc.

PSEUDO-CODE CONVENTIONS

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces { and }.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

```
node= record
        {
    data type 1 data 1;
    data type n data n;
      node *link;
        }
```

4. There are two Boolean values **TRUE** and **FALSE**.

Logical Operators

AND, OR, NOT

Relational Operators

<, <=,>,>=, =, !=

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. Compound data types can be formed with records. Here is an example,

Node. Record

```
{
 data type – 1   data-1;
.
.
.
 data type – n  data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

7. The following looping statements are employed.

For, while and repeat-until While Loop:

While < condition > do

{

<statement-1>

..

..

<statement-n>

}

For Loop:

For variable: = value-1 to value-2 step step do

{

<statement-1>

.

.

.

<statement-n>

}

repeat-until:

repeat
<statement-1>
.
.
.
<statement-n>
until<condition>

8. A conditional statement has the following forms.

→ If <condition> then <statement>
→ If <condition> then <statement-1>
Else <statement-1>

Case statement:

Case
{
: <condition-1> : <statement-1>
.
.
.
: <condition-n> : <statement-n>
: else : <statement-n+1>
}

9. Input and output are done using the instructions read & write. No format is used to specify the size of input or output quantities

Contd…

10. There is only one type of procedure: Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

consider an example, the following algorithm fields & returns the maximum of n given numbers:

```
1. algorithm Max(A,n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for i:= 2 to n do
6. if A[i] > Result then
7. Result :=A[i];
8. return Result;
9. }
```

Factorial of a number

```
// Declare variables
n, fact, i: integer
// Input n from user
write "Enter a positive integer n"
read n
// Initialize fact to 1
fact = 1
// Loop from 1 to n and multiply fact by i
for i = 1 to n
    fact = fact * i
end for
// Display the factorial of n
write "The factorial of", n, "is", fact
```

Issue in the study of algorithm

1. **How to create an algorithm.**
2. **How to validate an algorithm.**
3. **How to analyses an algorithm**
4. **How to test a program.**

1 .How to create an algorithm: To create an algorithm we have following design technique

a) Divide & Conquer
b) Greedy method
c) Dynamic Programming
d) Branch & Bound
e) Backtracking

2.How to validate an algorithm: Once an algorithm is created it is necessary to show that it computes the correct output for all possible legal input , this process is called algorithm validation.

3.How to analyses an algorithm: Analysis of an algorithm refers to computing Time & storage algorithms required.

a) Computing time-Time complexity: Frequency or Step count method
b) Storage space- To calculate space complexity we have to use number of input used in algorithms.

4.How to test the program: Program is nothing but an expression for the algorithm using any programming language. To test a program we need following

a) Debugging: It is processes of executing programs on sample data sets to determine whether faulty results occur & if so correct them.
b) Profiling or performance measurement is the process of executing a correct program on data set and measuring the time & space it takes to compute the result.

ANALYSIS OF ALGORITHM

PRIORI	POSTERIORI
1.Done priori to run algorithm on a specific system	1.Analysis after running it on system.
2.Hardware independent	2.Dependent on hardware
3.Approximate analysis	3.Actual statistics of an algorithm
4.They do not do posteriori analysis	4.Dependent on no. of time statements are executed

Problem: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Algorithmic Approach

1.Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60

2.FOR EACH Student PRESENT DO the following:
Increase the **Count** by One

3.Then Subtract **Count** from **total** and store the result in **absent**

4.Display the number of absent students

Problem: **Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?**

Psuedo Approach:

```
Procedure Strength ()
{
Count := 0, absent := 0, total := 60
for i:= 1 to total do
   Count := Count + 1
absent := total - Count
return absent
}
```

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.
5. To compare the performance of the algorithm with respect to other techniques.

Need of Algorithm

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.

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 . In general,

a) Instruction space: executable program
b) Data space: Required to store all the constant and variable data space.
c) Environment: It is required to store environment information needed to resume the suspended space.

2. TIME COMPLEXITY:

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Space complexity

Now there are two types of space complexity

a) Constant space complexity

b) Linear(variable)space complexity

1.Constant space complexity: A fixed amount of space for all the input values.

Example : int square(int a)

```
{
        return a*a;
}
```

Here algorithm requires fixed amount of space for all the input values.

2.Linear space complexity:

The space needed for algorithm is based on size.

- **Size of the variable ‘n’ = 1 word**
- **Array of a values = n word**
- **Loop variable = 1 word**
- **Sum variable = 1 word**

Example:

```
int sum(int A[],int n)
{                                   n
int sum=0,i;                        1
for (i=0;i<n;i++)                   1
Sum=sum+A[i];                       1
Return sum;
}
```

Ans : 1+n+1+1 = n+3 words

1. Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to complete.

Space needed by an algorithm is given by

S(P)=C(fixed part)+Sp(Variable part)

fixed part: independent of instance characteristics.

eg: space for simple variables, constants etc

Variable part: Space for variables whose size is dependent on particular problem instance.

Examples:

1.Algorithm sum(a,,b,c)

{

a=10; a-1

b=20; b-1

c=a+b; c-1

}

s(p)=c+sp

3+0=3

0(n)=3

2. algorithm sum(a[],n)

{ -> n

total-=0; -> 1

For i=1 to n do -> 1,1

Total=total+a[i]

Return total

Algorithm-1

```
Algorithm abc(a,b,c)
{
return a+b*c+(a+b-c)/(a+b) +4.0;

}
```

a→1
b→1
c→1

>= 3 units

Algorithm-2

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

i,n,s→1 unit each
a→n units

>= n+1 units

Algorithm-3

```
Algorithm RSum(a,n)
{
if(n≤0) then return 0.0;
else return Rsum(a,n-1)+a[n];
}
```

Rsum(a,n)→1(a[n])+1(n)+1(return)=3units
Rsum(a,n-1)→1(a[n-1])+1(n)+1(return)
.
.
.
Rsum(a,n-n)→1(a[n-n])+1(n)+1(return)

Total→>=3(n+1) units

2. Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to complete.

T(P)= compile time+ execution time

T(P)= tp (execution time)

Step count:

- For algorithm heading→0
- For Braces→0
- For expressions→1
- for any looping statements→no.of times the loop is repeating.

1.Constant time complexity : If a program required fixed amount of time for all input values is called Constant time complexity .

Example : int sum(int a , int b)

```
{
return a+b;
}
```

2.Linear time complexity: If the input values are increased then the time complexity will changes.

- comments = 0 step
- Assignment statement= 1 step
- condition statement= 1 step
- loop condition for n times = n+1 steps
- body of the loop = n steps

Example : int sum(int A[],int n)

{

int sum=0,i;

for (i=0;i<n;i++)

sum=sum+A[i];

return sum;

cost	repetation	total
1	1	1
1+1+1	1+(n+1)+n	2n+2
1	n	1n
1	1	1
		4n+4

Algorithm-1

```
1. Algorithm abc(a,b,c)
2. {
3. return a+b*c+(a+b-c)/(a+b) +4.0;
4. }
```

```
→0
→0
→1
→0
--------
1 unit
--------
```

Algorithm-2

```
1. Algorithm sum(a,n)
2. {
3. s=0.0;
4. for i=1 to n do
5.  s= s+a[i];
6.  return s;
7. }
```

```
→0
→0
→1
→n+1
→n
→1
→0
--------
2n+3
```

Algorithm-3

```
Algorithm RSum(a,n)
{
if(n≤0) then return 0.0;
else return Rsum(a,n-1)+a[n];
}
```

$$T(n) = \begin{cases} 2 & \text{if } n=0 \\ 2 + T(n-1) & \text{if } n>0 \end{cases}$$

$$T(n) = 2 + T(n-1)$$
$$= 2 + (2+T(n-2)) = 2*2+T(n-2)$$
$$= 2*2+(2+T(n-3)) = 2*3+T(n-3)$$
$$\vdots$$
$$= 2*n+T(n-n) = 2n+T(0)$$
$$T(n) = 2n+2$$

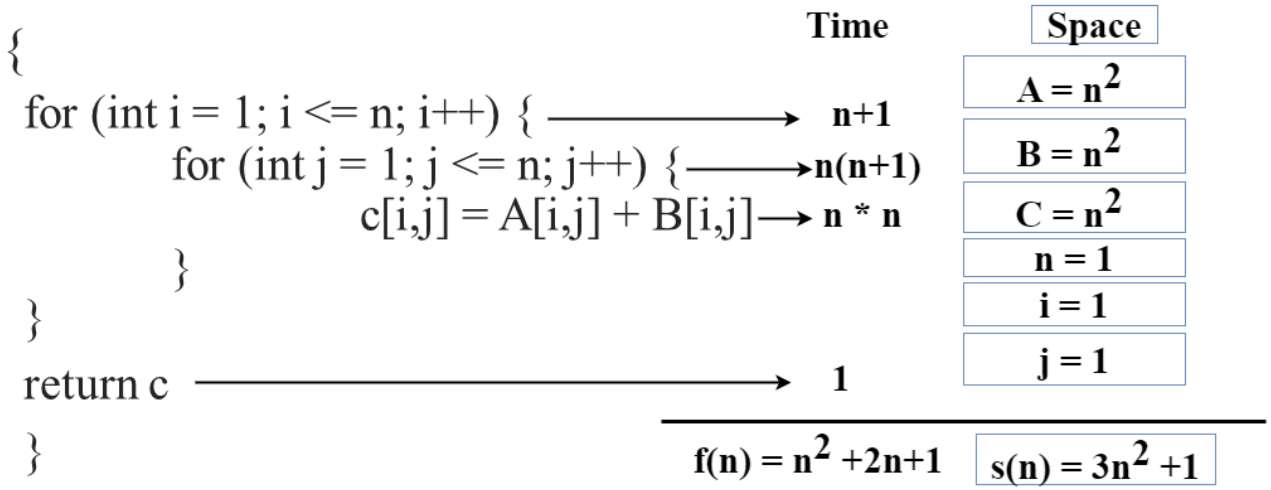
TIME COMPLEXITY

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for i=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Examples to solve

Algorithm Add(A,B,n)



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

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

Examples to solve

Algorithm Multiply(A,B,n)

```
{
 for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= n; j++) {
                   c[i,j] = 0;
                  for (k=0; k<n; k++) {
                       c[i,j] = c[i,j] + A[i,k] + B[k,j]

                   }
            }
 }
 return c
 }
```

s(n) = ?

f(n) = ?

Example

For (i=0; i<n; i++) = O(n)

For (i=0; i<n; i=i+2) = n/2 = O(n)

For (i=n;i>n; i--) = O(n)

For (i=1; i<n; i=i*2) = $O(\log_2 n)$

for (i=1; i<n; i=i*3) = $O(\log_3 n)$

For (i=n; i>1; i = i/2) = $O(\log_2 n)$

Types of Time functions

$O(1)$	-	Constant
$O(\log n)$	-	Logarithmic
$O(n)$	-	Linear
$O(n^2)$	-	Quadratic
$O(n^3)$	-	Cubic
$O(2^n)$	-	Exponential

KINDS OF ANALYSIS

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.
- Need assumption of statistical distribution of inputs.

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 time grows as a function of the problem size

Analysis of an Algorithm

- The goal of analysis of an algorithm is to compare algorithm in running time and also Memory management.
- Running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

Running time of an algorithm depends on

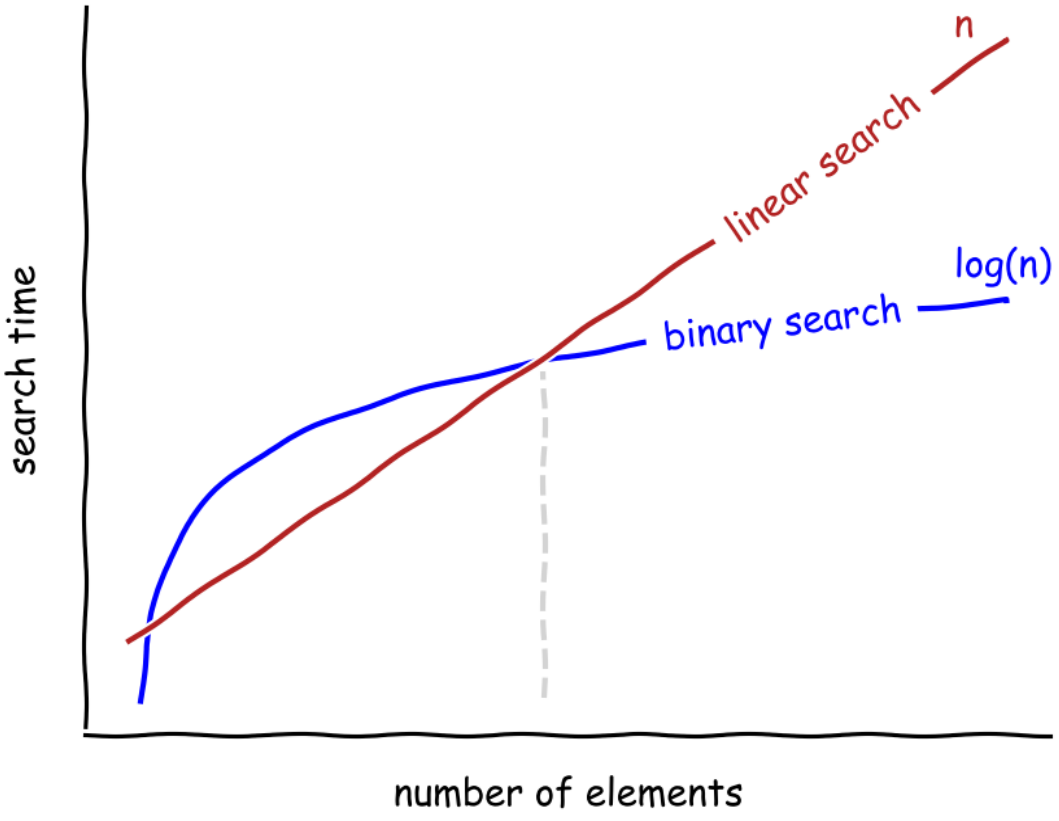
1.Speed of computer

2.Programming language

3.Compiler and translator

Examples: binary search, linear search

Linear vs Binary search



ASYMPTOTIC NOTATION

ASYMPTOTIC NOTATION: The mathematical way of representing the Time complexity.

The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers.

Definition : It is the way to describe the behavior of functions in the limit or without bounds.

Growth rate **Time +memory**

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

1. Big oh (O)notation - Upper bound
2. Big omega (Ω) notation - Lower bound
3. Theta(Θ) notation - Average bound

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < \ldots < n^n$

1.Big oh (O)notation

1.Big oh (O)notation : Asymptotic “less than”(slower rate).

This notation mainly represent upper bound of algorithm run time.

Big oh (O)notation is useful to calculate maximum amount of time of execution.

By using Big-oh notation we have to calculate worst case time complexity.

Formula : $f(n)<=c\ g(n)$ $n>=n_0$, $c>0$, $n_0>=1$

Definition: Let f(n) ,g(n) be two non negative (positive) function

now the

f(n)=O(g(n)) iff there exist two positive constant c, n0

such that

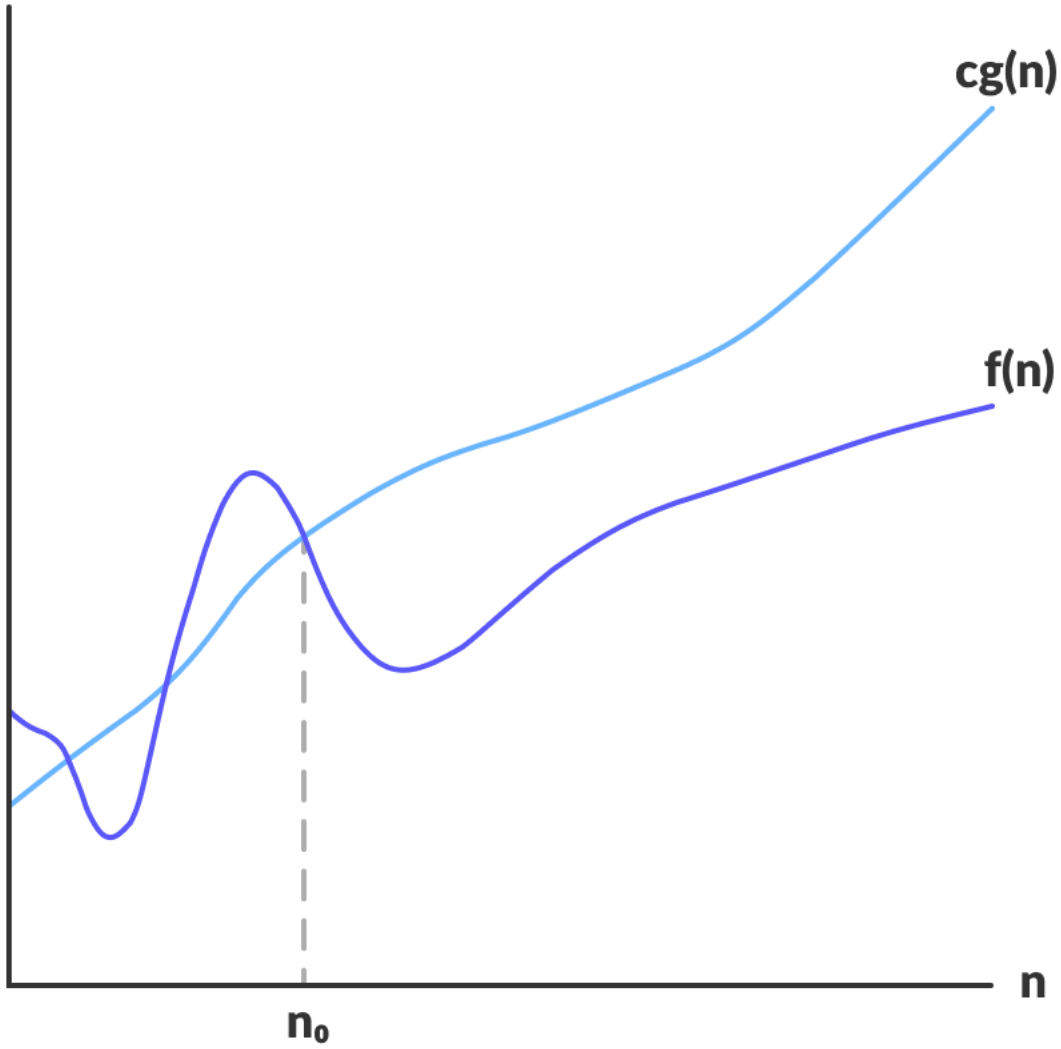
f(n)<= c.g(n) for all value of n>0 & c>0

1.Big O-notation

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

- We use *O*-notation to give an asymptotic upper bound of a function, to within a constant factor.
- $f(n) = O(g(n))$ means that there existes some constant c s.t. $f(n)$ is always $\le cg(n)$ for large enough n.



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

Example : f(n)=3n +2 & g(n)= n

Formula : f(n)<=c g(n) $n>=n_0$, c>0 ,n_0 >=1

f(n)=3n+2 & g(n)=n

Now 3n+2<=c.n

3n+2<=4.n

Put the value of n =1

5<=4 false

N=2 8<=8 true now n0>2 For all value of n>2 & c=4

now f(n)<= c.g(n)

3n+2<=4n for all value of n>2

Above condition is satisfied this notation takes maximum amount of time to execute. So that it is called worst case complexity.

2.Ω-Omega notation

Ω-Omega notation : Asymptotic "greater than"(faster rate).

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 : $f(n)>=c\ g(n)$ $n>=n_0$, $c>0$,$n_0>=1$

where c is constant, n is function

- Lower bound
- Best case

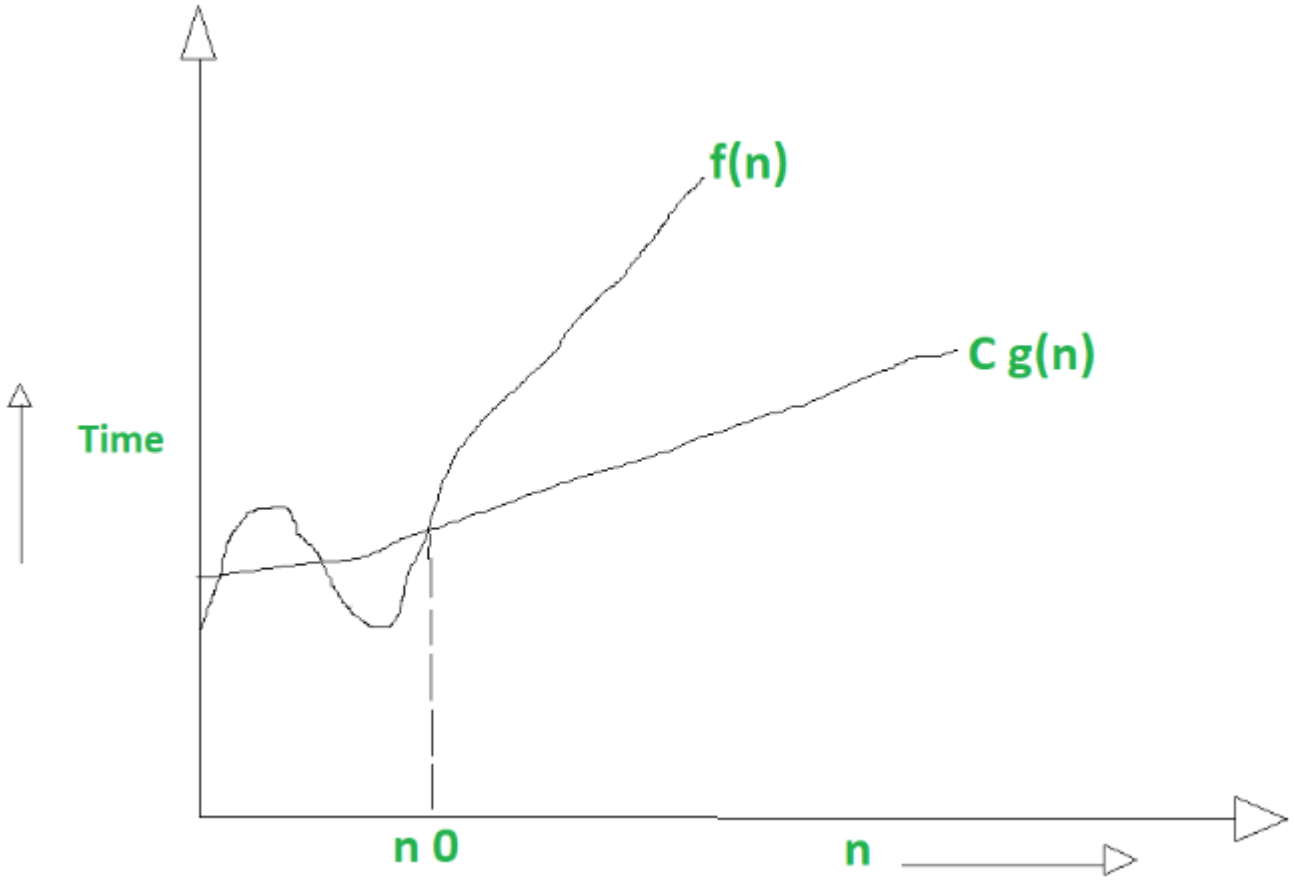
Ω-Omega notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

- We use *Ω*-notation to give an asymptotic lower bound on a function, to within a constant factor.
- $f(n) = \Omega(g(n))$ means that there exists some constant c s.t.

  is always $f(n) \ge cg(n)$ for large enough n.


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

Examples

Example : f(n)=3n +2

Formula : f(n)>=c g(n) $n>=n_0$, $c>0$, $n_0>=1$

f(n)=3n+2

3n+2>=1*n, c=1 put the value of n=1

n=1 5>=1 true n0>=1 for all value of n

It means that f(n)= Ω g(n).

3. Θ-Theta notation

Theta (Θ) notation : Asymptotic “Equality”(same rate).

It represent average bond of algorithm running time.

By using theta notation we can calculate average amount of time.

So it called average case time complexity of algorithm.

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

where c is constant, n is function

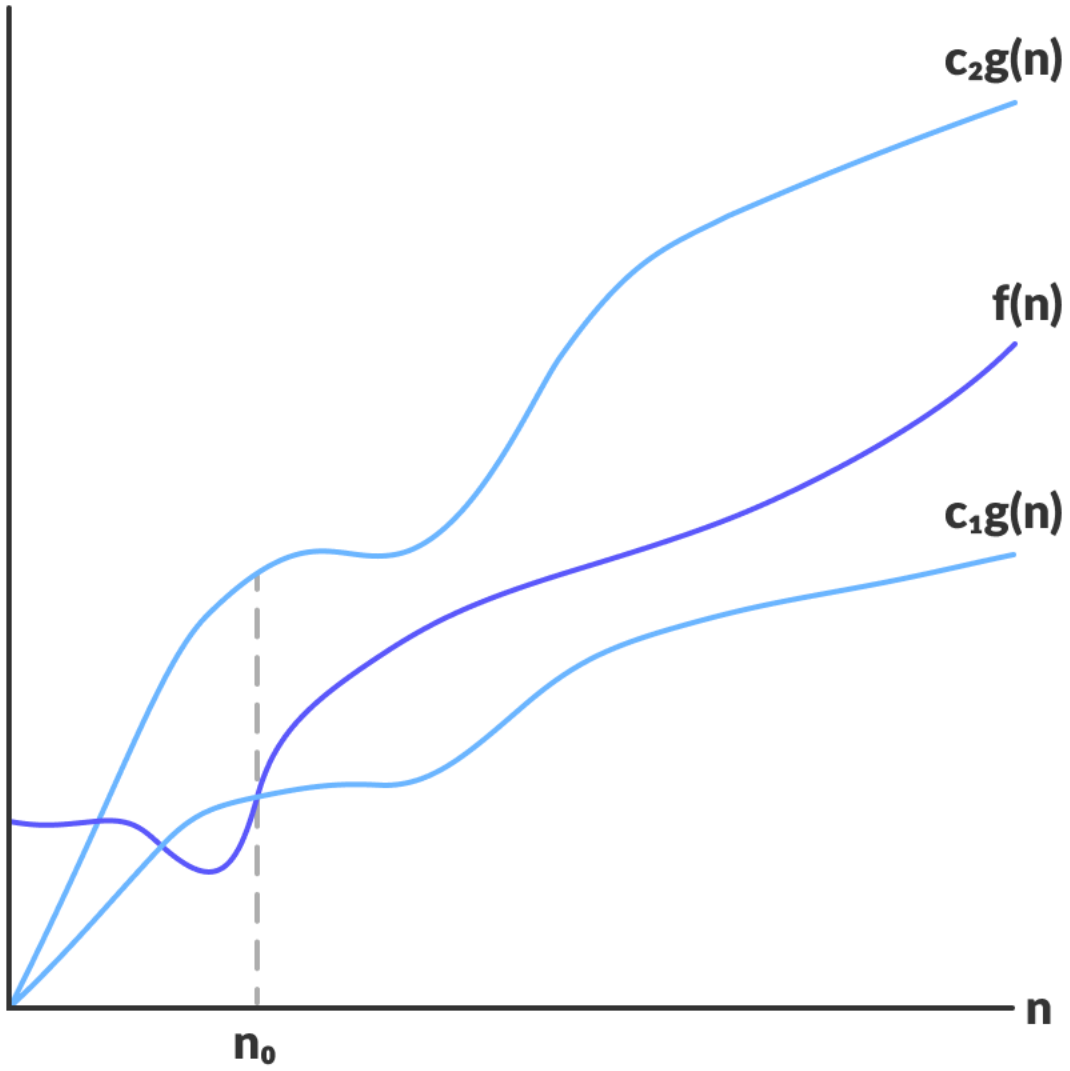
- Average bound

Θ-Theta notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ s.t.} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

- A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sand-wiched" between $c_1 g(n)$ and $c_2 g(n)$ or sufficiently large n.
- $f(n) = \Theta(g(n))$ means that there exists some constant c_1 and c_2 s.t. $c_1 g(n) \le f(n) \le c_2 g(n)$ for large enough n.



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

Examples

Example : f(n)=3n+2

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

f(n)=2n+3

1*n<=3n+2<=4*n now put the value of n=1

we get 1<=5<=4 false

n=2 we get 2<=8<=8 true

n=3 we get 3<=11<=12 true

Now all value of n>=2 it is true above condition is satisfied.

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

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

Thank You