Growth of Functions

Algorithm

- An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.
- Algorithms can be described using English language or pseudocode etc.

Evaluating algorithms

- One of the important criteria in evaluating algorithms is the time it takes to complete a job.
- To have a meaningful comparison of algorithms, the estimate of computation time:
  - must be independent of the programming language, compiler, and computer used;
  - must reflect on the size of the problem being solved;
  - and must not depend on specific instances of the problem being solved.
- The quantities often used for the estimate are the **worst case execution time, and average execution time of an algorithm**, and they are represented by the number of some key operations executed to perform the required computation.

Algorithm example

- **Example: Algorithm for Sequential Search**
- Algorithm **SeqSearch**(L, n, x)
- L is an array with n entries indexed 1, .., n, and x is the key to be searched for in L.
- Output: if x is in L , then output its index, else output 0.

1. *index := 1;*
2. *while (index <=n and L[index] ≠ x)*
3. *index := index + 1 ;*
4. *if (index > n) , then index := 0*
5. *return index .*

Asymptotic Efficiency

- When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the **asymptotic efficiency** of algorithms.
- How the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.
- An algorithm that is asymptotically more efficient will be the best choice

Asymptotic notation

- The notation used to describe the asymptotic running time of an algorithm is defined in terms of functions whose domains are the set of natural numbers.
- This notation refers to how the problem scales as the problem gets larger.
- Main concern is with algorithms for large problems, i.e. how the performance scales as the problem size approaches infinity.

Different asymptotic notations

1. O-notation
2. Ω-notation
3. Θ-notation

Asymptotic Notations:-

They are Mathematical way of representing the time complexity. They are used in prior Analysis.

prior analysis :- we don't execute the Algorithm.

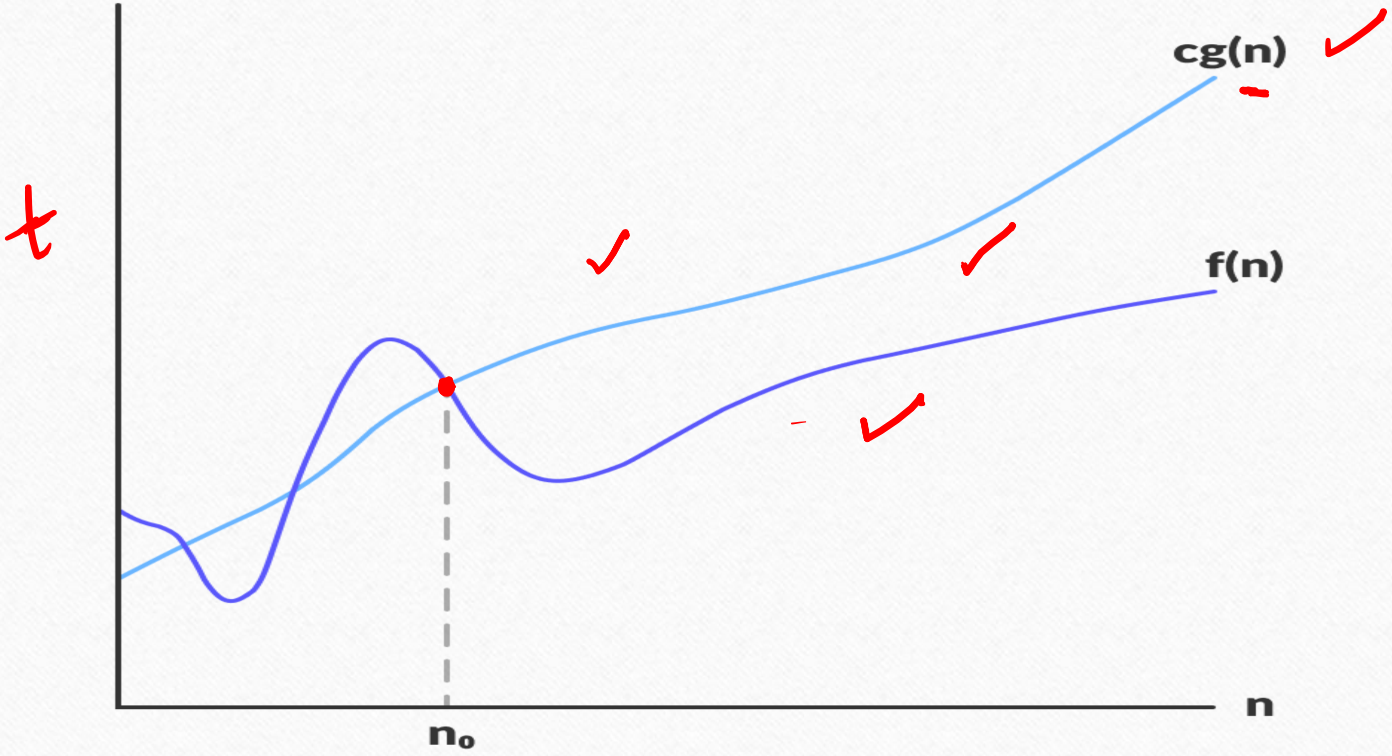
So there are 3- types of Notations:-

① O - Notation:- When we have only asymptotically upper bound of the Algorithm, then we use O - Notation.

$$f(n) = O(g(n))$$ ✓

$$0 \le f(n) \le c\,g(n) \quad \forall n \ge n_0$$

c, n_0 are +ve Constant



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

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

$0 \le f(n) \le c\,g(n)$

$\forall n \ge n_0$

c, n_0 are +ve constants.

① Ex

$f(n) = 3n+2$ $\quad g(n) = n$

Can we say that $f(n) = O(g(n))$

$f(n) \le c(g(n))$

$3n+2 \le cn$ ✓

$n_0 \ge 1$

$c = 4 \quad n = 1$

$c = 4$ ✓

$5 \le 4 \cdot 1 \quad F$

$n \ge n_0$

$n_0 \ge 2 \quad T$

$c = 4 \quad n = 2$

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

$6+2 \le 4 \cdot 2$

$8 \le 8$ ✓ T

Ex 2:- $n^3 + n + 5 \leq c\, n^3$

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

$c = 7 \quad n_0 = 1$

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

Big-O Notation is used to Capture all upper Bounds

But we prefer the least upper bound.

$f(n)$ is $O(n^3) \rightarrow O(n^3)$

$O(n^4)$

$O(n^5)$

⋮

Ex 3:-

Prove $3^n \neq O(2^n)$

Soln:- Proof by Contradiction

Suppose $3^n = O(2^n)$

$$3^n \leq c \cdot 2^n$$

Dividing both sides by 2^n

$$(1.5)^n \leq c$$

This is a Contradiction. Because c is a Constant. It can not depend on the value of n. So our assumption is wrong.

Hence proved.

② Ω Notation:-

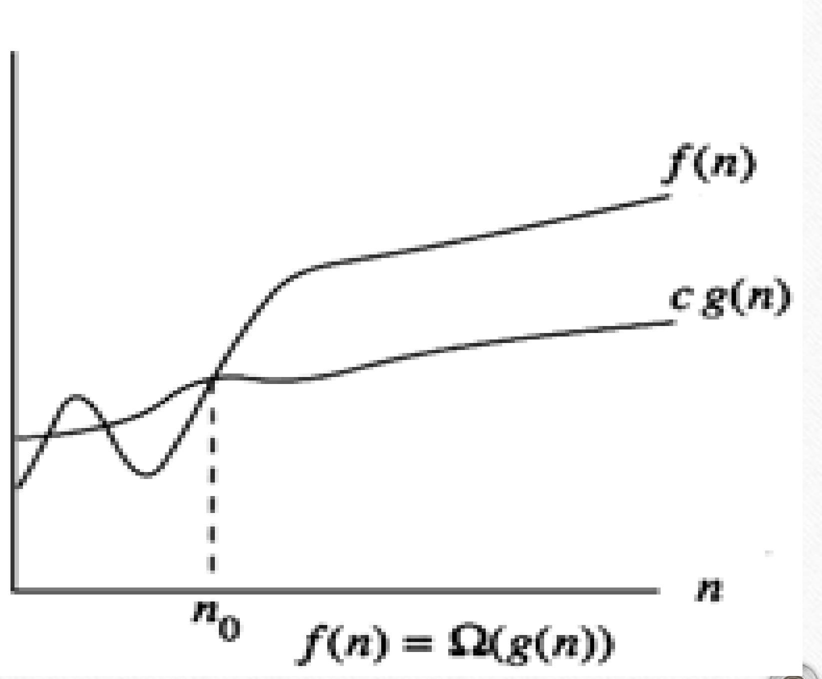
It provides Asymptotically lower bound on the function.

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

$$f(n) \geq c\,g(n) \geq 0$$

$$n \geq n_0$$

c, n_0 are +ve Constants.



Ex 1

$$f(n) = 3n+2 \qquad g(n) = n$$

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

$$f(n) \geq c\, g(n)$$

$$3n+2 \geq cn$$

$$c=1 \quad n_0 \geq 1$$

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

Ex2:-

$$n^3+n+5 \geq c \cdot n^3$$

$c=1 \quad n_0=1$

$7 \geq 1$

$c=1 \quad n_0=2$

$15 \geq 8$

$c=1 \quad \forall\, n_0 \geq 1$

Hence it is $\Omega(n^3)$

Ex 3:-

Prove $3n^2+2 \neq \Omega(n^3)$

$$3n^2+2 \geq c \cdot n^3$$

Dividing both sides by n^3, we get

$$\frac{3n^2+2}{n^3} \geq c$$

This is a contradiction. Because value of c is a constant. But in this case it is dependent on n.

So this is true.

$$3n^2+2 \neq \Omega(n^3)$$

③ Θ-Notation:-

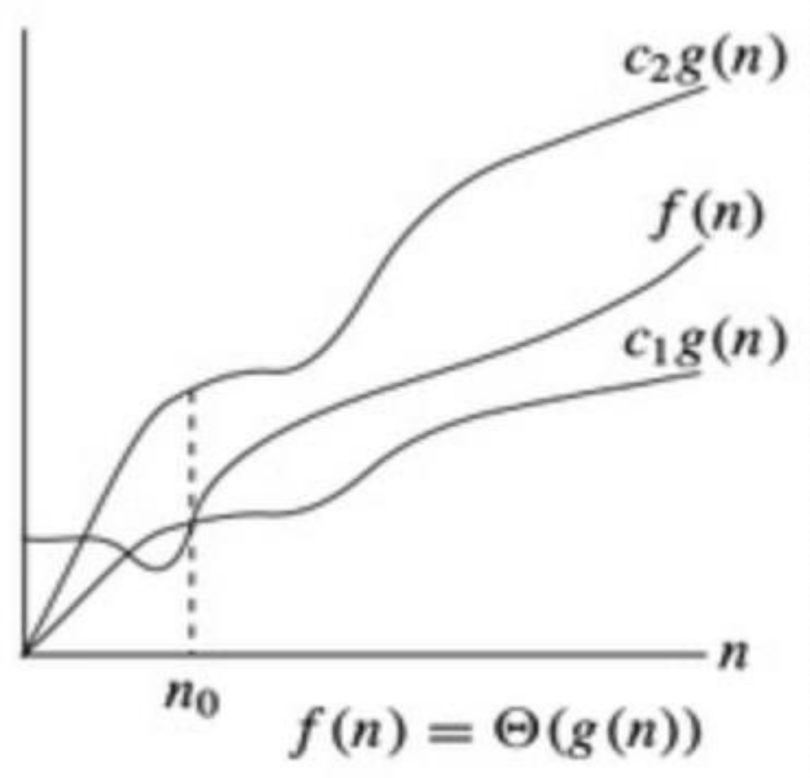
When our function $f(n)$ is

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

$$c_1(g(n)) \le f(n) \le c_2(g(n))$$

$$\forall\, n \ge n_0 \quad \&$$

c, n_0 +ve constants.



Θ-Notation can be used to denote tight bounds of the Algorithm.

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

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

&

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

Then it is $f(n) = \Theta(g(n))$

Ex 1:-

$$f(n) = 3n+2 \qquad g(n) = n$$

$$f(n) \le c_2\, g(n)$$

$$3n+2 \le c_2\, n$$

$$c_2 = 4 \qquad n_0 \ge 1$$

$$f(n) \ge c_1\, g(n)$$

$$3n+2 \ge n$$

$$c_1 = 1 \qquad n_0 \ge 1$$

Θ is also called Asymptotically equal.

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

Ex 2:- $10n^2 + 4n + 2 = O(n^2)$

(1) $O(n^2)$

$$10n^2 + 4n + 2 \le c_1 \cdot n^2 \quad \forall n \ge n_0$$

$$10 + 4 + 2 \le 20 \cdot 1 \qquad n_0 = 1, \; c_1 = 20$$

$$16 \le 20 \quad T \qquad n_0 = 2, \; c_1 = 20$$

$$50 \le 80 \quad T$$

So this is $O(n^2)$.

② $\Omega(n^2)$

$$10n^2+4n+2 \geq c_2 n^2$$

$16 \geq 10$ T $\qquad c_2=10,\ n_0=1$

T $\qquad c_2=10,\ n_0=2$

for $c_1=20,\ n_0=1$

Both the functions are true.

So we can say that this is $\Theta(n^2)$.

Complexity of Algorithms

Complexity

- An Algorithm can be analyzed on two accounts space and time:
- Memory Space: Space occupied by program code and the associated data structures.
- **CPU Time:** Time spent by the algorithm to solve the problem.

Counting operations

- Instead of measuring the actual timing, we count the number of operations
    - Operations: arithmetic, assignment, comparison, etc.
- Counting an algorithm’s operations is a way to assess its efficiency.
    - An algorithm’s execution time is related to the number of operations it requires

Example: Counting Operations

```
for (int i = 1; i <= n; i++){
    perform 100 operations; // A
        for (int j = 1; j <= n; j++){
        perform 2 operations; // B
    }
}
```

Asymptotic Analysis

Asymptotic analysis is an analysis of algorithms that focuses on :

- Analyzing problems of large input size.
- Consider only the leading term of the formula.
- Ignore the coefficient of the leading term

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example

$f(n) = 2n^2 + 100n$

- $f(1000) = 2(1000)^2 + 100(1000) = \mathbf{2{,}0}00{,}000 + \mathbf{1}00{,}000$
- $f(100000) = 2(100000)^2 + 100(100000) = \mathbf{20{,}000}{,}000{,}000 + \mathbf{10}{,}000{,}000$ Hence, lower order terms can be ignored.

Examples: Leading Terms

- $a(n) = n + 4$

  Leading term:

- $b(n) = 240n + 2n^2$

  Leading term:

- $c(n) = n\lg(n) + \lg(n) + n\lg(\lg(n))$

  Leading term:

Upper Bound: Big-Oh Notation

$T(n)=O(g(n))$ is defined as: $T(n) <= c*g(n)$ where $c>0$

- From the above relation we can say that for a large value of n, the function 'g' provides an upper bound on the growth rate 'T.

Order of growth:

- $O(1) < O(\log_k n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$

Running Time Calculations

Rule 1:

- Simple program statements are assumed to take a constant amount of time which is **O(1)** i.e. Not dependent upon n.

Example:

- One arithmetic operation (eg., +, *)
- One assignment
- One test (e.g. x==0)
- One read(accessing an element from an array)

Running Time Calculations

Rule 2: **Loops**

- The running time of a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations of the loop.

Example;

for (i = 0; i < N; i++) {

statement(s) of O(1)

}

Running Time Calculations

Rule 3 - **Nested loops**

- The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops
- Example:

```
for (i = 0; i < N; i++) {
for (j = 0; j < M; j++) {
statement(s) of O(1)
}
}
```

Running Time Calculations

Rule 4: Conditional Statements

- The running time of a conditional statement is never more than the running time of the test plus the largest of the running times of the various blocks of conditionally executed statements.

Rule 5:Consecutive statements

- These just add
- Only the maximum is the one that counts

Example

Parameters: A finite-length list, L, of positive integers.

Returns: The sum of the integers in the list.

```
{ sum := 0;
for each x in L
{ sum := sum + x;
}
 return sum;
 }
```

Complexity of Algorithms

Example 1

```
A( )
{ int a=0;
    for(int i=0; i<m; i++)
    {--}
    for(int j=0; j<n; j++)
    {--}
}
```

Example 2

```
A( )
{ int a=0;
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
            {--}
}
```

Example 3

```
A( )
{ int a=0;
    for(int i=1; i<=n; i=i*2)
                    {--}
}
```

Example 4

```
A( )
{ int a=0;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j=j+2)
            {--}
}
```

Example 5

```
A( )
{ int a=0;
    for(int i=1;i<=n;i++)
        for(int j=1;j<n;j=j*2)
            for(int k=1;k<=n;k++)
                {--}
}
```

Example 6

```
A( )
{for(int i=n; i>0; i/=2)
        for(int j=1; j<=i; j++)
                {--}
}
```

Example 7

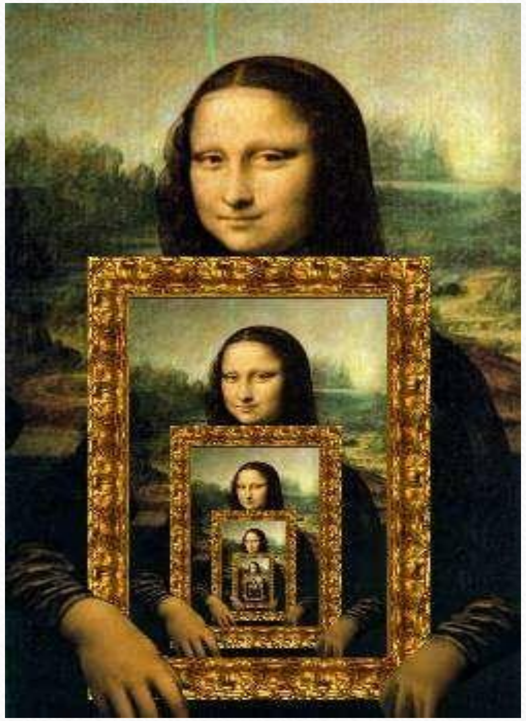
```
A( )
{for(i=1;i<=n;i++)
    for(j=1;j<=i;j++)
        for(k=1;k<=100;k++)
            {-- }
}
```

Recursive Functions

Recursion

- Sometimes it is possible to define an object (function, sequence, algorithm, structure) in terms of itself. This process is called **recursion.**
- An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.

Recursive Examples



Recursive definition

- There are two parts:
  - Basic case (basis): the most primitive case(s) of the entity are defined without reference to the entity.
  - Recursive (inductive) case: new cases of the entity are defined in terms of simpler cases of the entity.
- Recursive sequences
  - A sequence is an ordered list of objects, which is potentially infinite.
  - A sequence is defined recursively by explicitly naming the first value (or the first few values) in the sequence and then define later values in the sequence in terms of earlier values.

Examples

- A recursively defined sequence:
  - $S(1) = 2$ …… basis case
  - $S(n) = 2 * S(n-1)$, for $n >= 2$ …… recursive case
  - what does the sequence look like?

  - $T(1) = 1$ …… basis case
  - $T(n) = T(n-1) + 3$, for $n >= 2$ …… recursive case
  - what does the sequence look like?

Recursive Definitions of Important Functions

- Some important functions/sequences defined recursively

1. Factorial function:

2. Fibonacci numbers:

Recursively defined functions

- **Example:** Assume a recursive function on positive integers:
  - $f(0) = 3$
  - $f(n+1) = 2f(n) + 3$
- **What is the value of $f(2)$?**

Example

- Give a recursive definition of the following sets of objects:

  1, 2, 4, 7, 11, 16, 22, …

Solution:

Ackermann function

- The Ackermann function is a classic example of a recursive function. It grows very quickly in value, as does the size of its call tree.
- The Ackermann function is:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Example

McCarthy 91 function

- The McCarthy 91 function is a recursive function, defined by computer scientist John McCarthy.
- The McCarthy 91 function is defined as

$$M(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ M(M(n+11)), & \text{if } n \le 100 \end{cases}$$

Example

Thank You