Performance analysis

1. Space Complexity
2. Time Complexity
3. Asymptotic Notations

   i) Big oh

   ii) Big Omega

   iii)Big Theta

Space Complexity

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution.

For any algorithm, memory is required for the following purposes...

1. Memory required to store program instructions
2. Memory required to store constant values
3. Memory required to store variable values
4. And for few other things

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

1. Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...
2. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
3. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
4. **Data Space:** It is the amount of memory used to store all the variables and constants.

calculate the space complexity

To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler).

For example, the C Programming Language compiler requires the following...

- 2 bytes to store Integer value,
- 4 bytes to store Floating Point value,
- 1 byte to store Character value,
- 6 (OR) 8 bytes to store double value

Example 1

Consider the following piece of code...

```
int square(int a)
{
    return a*a;
}
```

In above piece of code, it requires 2 bytes of memory to store variable '**a**' and another 2 bytes of memory is used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity*.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity

Example 2

Consider the following piece of code...

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

In above piece of code it requires

'n*2' bytes of memory to store array variable 'a[]'
2 bytes of memory for integer parameter 'n'
4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

Time Complexity

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

Time complexity of an algorithm can be defined as follows...

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

Generally, running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine
3. **Read** and **Write** speed of the machine.
4. The time it take to perform Arithmetic operations , logical operations return value and assignment operations ect.
5. **Input** data

Calculating Time Complexity

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

calculate time complexity of an algorithm

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. Single processor machine
2. 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Example 1

Consider the following piece of code...

```
int sum(int a, int b)
{
  return a+b;
}
```

In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Example 2

Consider the following piece of code...

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

	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
int sumOfList(int A[ ], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 Total Time required

In above calculation

Cost is the amount of computer time required for a single operation in each line.

Repeatation is the amount of computer time required by each operation for all its repeatations.

Total is the amount of computer time required by each operation to execute.

So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity

Asymptotic Notation

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Example

For example, consider the following time complexities of two algorithms...

Algorihtm 1 : $5n^2 + 2n + 1$

Algorihtm 2 : $10n^2 + 8n + 3$

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. '**n**' value). In above two time complexities, for larger value of '**n**' the term in algorithm 1 '**2n + 1**' has least significance than the term '**$5n^2$**', and the term in algorithm 2 '**8n + 3**' has least significance than the term '**$10n^2$**'.

Here for larger value of '**n**' the value of most significant terms ($\mathbf{5n^2}$ and $\mathbf{10n^2}$) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of '**n**' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Asymptotic Notations

- Majorly, we use THREE types of Asymptotic Notations and those are as follows...
- **Big - Oh (O)**
- **Big - Omega (Ω)**
- **Big - Theta (Θ)**

Big - Oh notation

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

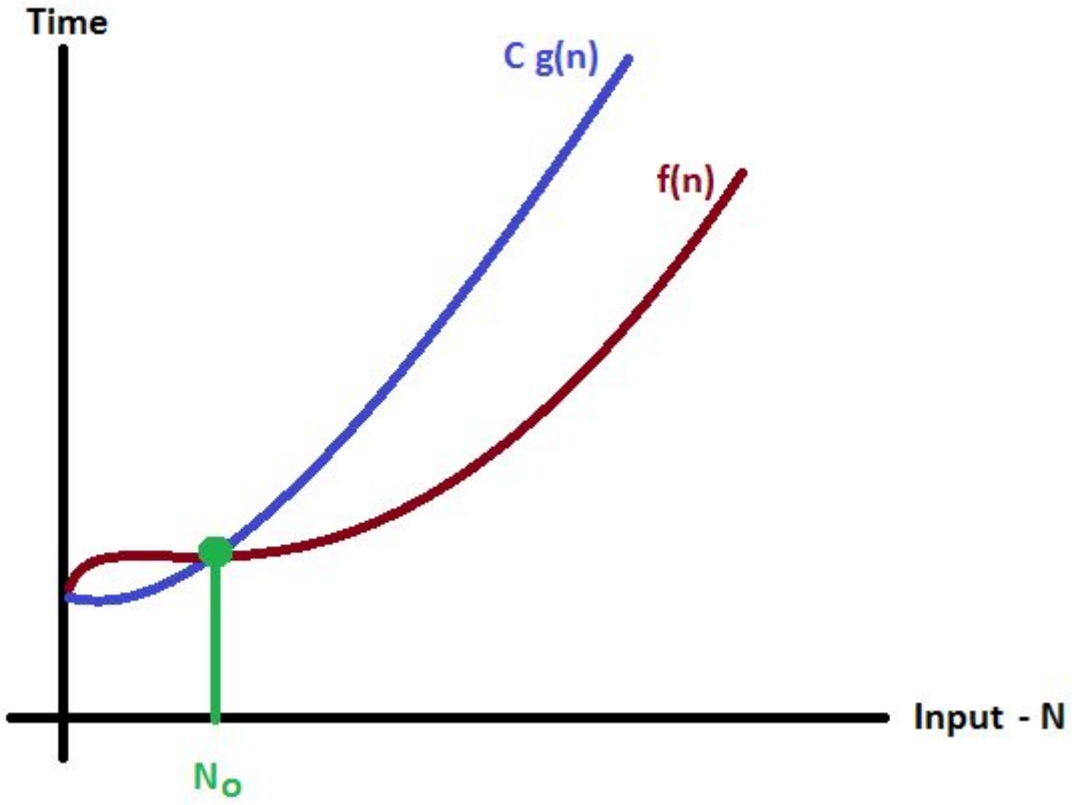
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



Example

In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

$\Rightarrow$3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

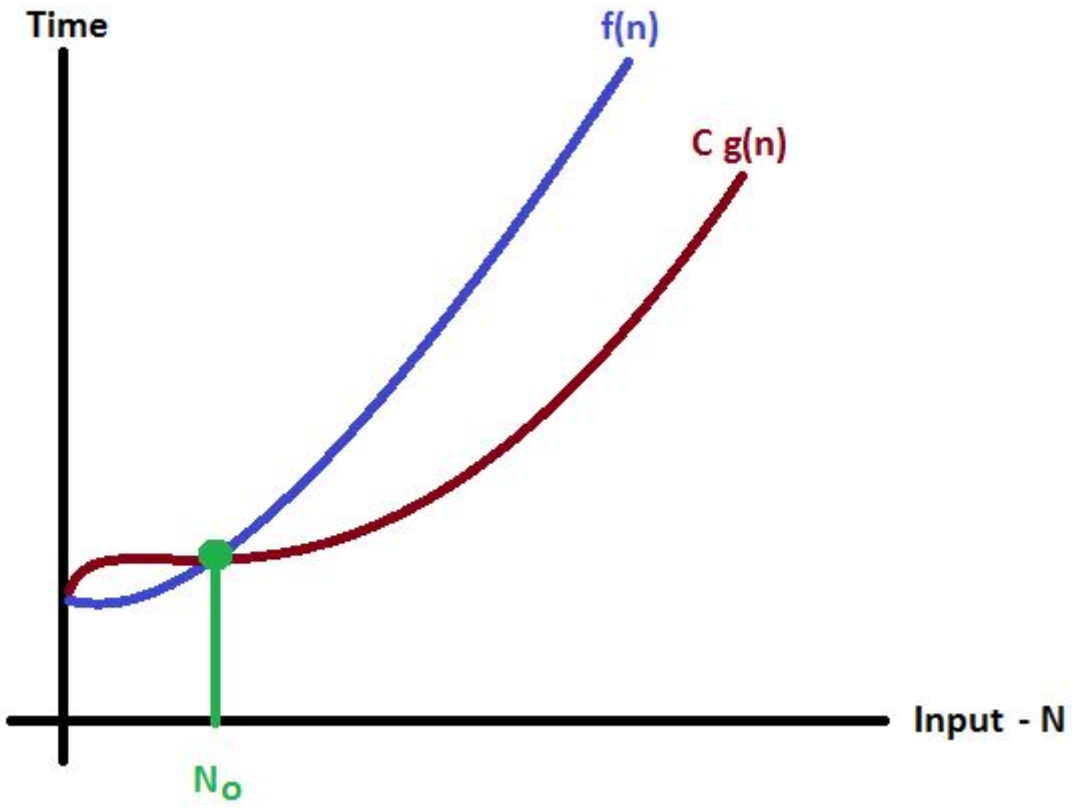
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω (g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

$\Rightarrow$3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**. By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

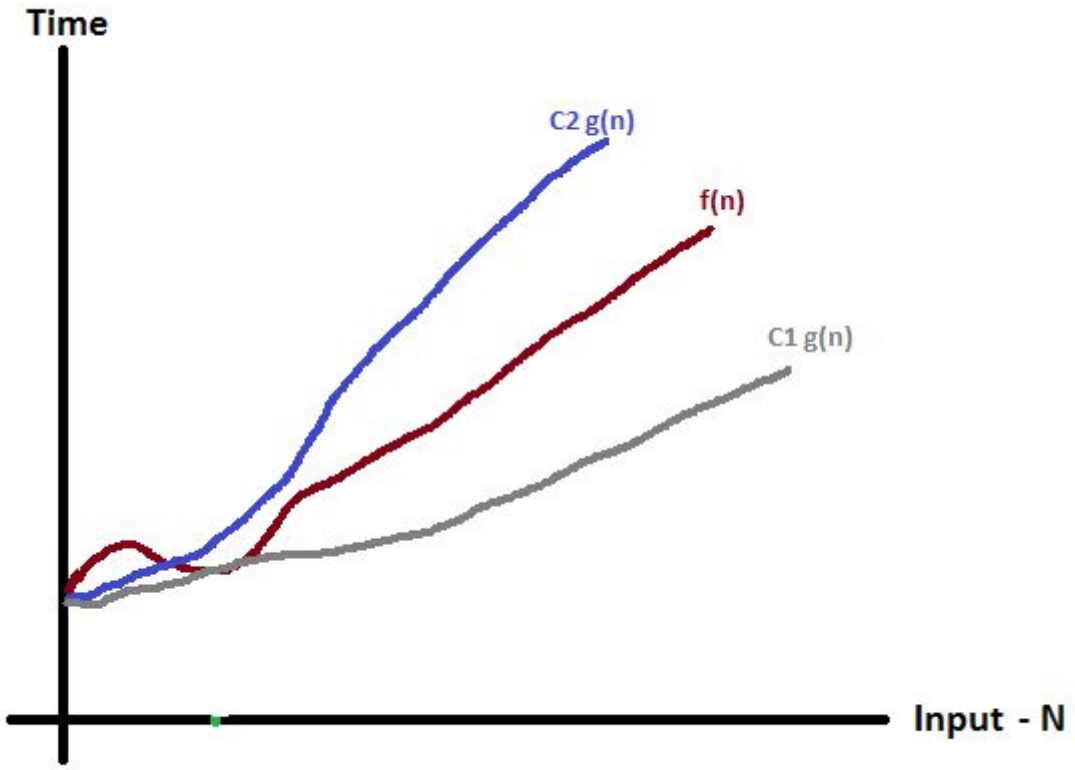
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) >= C_2 g(n) for all n >= n_0, C_1, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- In above graph after a particular input value n_0, always $\mathbf{C_1\ g(n)}$ is less than **f(n)** and $\mathbf{C_2\ g(n)}$ is greater than **f(n)** which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) >= C_2 g(n)** for all values of **C_1, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) >= $\Rightarrow C_2$ g(n)

C_1 n <= 3n + 2 >= C_2 n^2

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 1**.

By using Big - Theta notation we can represent the time compexity as follows...

3n + 2 = Θ(n)

Some Properties of Asymptotic Order of Growth

Transitivity :

- ^ $f(n) \in O(g(n))$ and $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
- ^ $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$
- ^ $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$

Reflexivity ^:

- $f(n) \in O(f(n))$
- ^ $f(n) \in \Theta(f(n))$
- ^ $f(n) \in \Omega(f(n))$

Symmetry and Transpose Symmetry ^

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$
- ^ $f(n) \in O(g(n))$ if and only if $g(n) \in O(f(n))$

Analysis of Recursive Algorithms

What is a recursive algorithm?

Example: Factorial

$n!$ = 1•2•3...n and 0! = 1 (called initial case)

So the recursive definition $n!$ = n•(n-1)!

Algorithm *F*(*n*)

if n = 0 **then** return 1 // base case

else F(n-1)•n // recursive call

Basic operation? multiplication

during the recursive call Formula for multiplication

$M(n) = M(n\text{-}1) + 1$

is a recursive formula too. This is typical.

We need the initial case which corresponds to the base case

$M(0) = 0$

There are no multiplications

Substitution Method

Solve by the method of *backward substitutions*

$M(n) = M(n\text{-}1) + 1$

$= [M(n\text{-}2) + 1] + 1 = M(n\text{-}2) + 2$ substituted $M(n\text{-}2)$ for $M(n\text{-}1)$

$= [M(n\text{-}3) + 1] + 2 = M(n\text{-}3) + 3$ substituted $M(n\text{-}3)$ for $M(n\text{-}2)$

... a pattern evolves

$= M(0) + n$

$= n$

Therefore $M(n) \varepsilon \Theta(n)$

Procedure for Recursive Algorithm

1. Specify problem size
2. Identify basic operation
3. Worst, best, average case
4. Write recursive relation for the number of basic operation. Don't forget the initial conditions (IC)
5. Solve recursive relation and order of growth

MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation* (in the innermost loop).
3. Check whether the *number of times the basic operation is executed* depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case,and, if necessary, best-case *efficiencies* have to be investigated separately.
4. Set up a *sum* expressing the number of times the algorithm's basic operation is executed
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its *order of growth*.

Assignment-I

1. Write a program using recursion for
   a) Calculating the factorial of a given number b) Calculating the greatest common divisor of a given number.
2. Write the algorithm for matrix addition and find its complexity.
3. Solve the recurrence relation by expansion T(n)=T(n/2)+1

T(n)=1

Assume that n is an integer n>=1

4. What is the difference between circular linked list

doubly link list. Mention the applications of each type of list.

5. What is a dequeue? Give an option between a linear array and circular array, which one will you choose to implement an array. Justify your answer.