What is an algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

"Algorithm is a sequence of well defined, simple, unambiguous and effective statements, which when executed sequentially will produce the desired result in finite amount of time."

Properties of an algorithm

In the terms of:

Input: An algorithm takes zero or more inputs

Output: An algorithm results in one or more outputs (depends upon the program)

Unambiguity: instructions in an algorithm should be clear and simple.

Finiteness: the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.

An algorithm should be efficient and flexible.

All operations can be carried out in a finite amount of time.

Language Independent: The algorithm can be implemented in any language with the same output.

Why do we need algorithms?

It helps us to understand the given problem effectively by splitting it into small steps.

For example, let us consider making Maggie

Step 1: boil water

Step 2: add chopped onions, tomatoes and other vegetables to it

Step 3: next add Maggie masala

Step 4: then add noodles to it

Step 5: Taddaaahhhh!!! Time to eat

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2; we need to follow the specific order to make Maggie. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming. We will write an algorithm to subtract two numbers entered by the user.

Step 1: Start

Step 2: Declare three variables A, B, and difference.

Step 3: Input the values of A and B.

Step 4: Subtract the values of A and B and store the result in the Difference variable, i.e., difference=A+B.

Step 5: Print difference

Step 6: Exit

Writing an algorithm to find whether the given number is odd or even:

Algorithm

```
Step 1:  Start

Step 2:  [Take Input] Read: Number

Step 3:  Check:

If

Number%2 == 0 Then

Print: N is an Even Number.

Else

Print: N is an Odd Number.
```

Step 4: Exit

Analysis of complexity:

Programs derived from two algorithms for solving the same problem should both be

- Machine independent
- Language independent
- Realistic

The performance is measured using:

1. Space Complexity
2. Time Complexity

Space Complexity:

"The amount of space required to solve a problem and produce an output."

The space need by a program has the following components:

1. Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

2. Data space: Data space is the space needed to store all constant and variable values.

3. Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space.

"The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input."

Space complexity = space used by input+ extra space required

Time complexity:

The amount of time required to complete the execution.

The time complexity is mainly calculated by counting the number of steps to finish the execution.

```
Step 1:  Input A

Step 2:  Input B

Step 3:  set SUM=A+B

Step 4:  print sum
```

```
A=10;              // 1-time

B=20;              // 1-time

SUM=A+B;       // 1-time

Print ("%d", SUM); //1- time

            Total: 4- constant time
```

```
Step1:  Declared the variable "J" and sum and initialize sum with 0
Step2: Read the number of terms u wanted suppose it will stored in "n".
Step3:  begin the loop J=1 to J<n
            Sum=sum +J;
          Increment the value of J by 1
Step4: Display sum
```

```
J=1;            // 1-time
N=10;        //1-time
While (J<=N)   //N+1 times
 { Printf (" %d\t", J);  //N-times
J=J+1;             //N-Times}
 Total =1+1+ (N+1) +N+N        =3N+3
```

Linear loop:

For (J=0; J<100; J++)   //executes 100 times

$\hookrightarrow$ Time complexity is N

For (J=0; J<100; J=J+2)   //executes 50 times as it is j=j+2

$\hookrightarrow$ Time complexity is N/2

While (J<=N)   //executes N+1 times

Reason: while loop executes for N times but check the condition again therefore time complexity becomes N+1

Logarithmic loop:

```
For (J=1; J<1000; J=J*2)    //executes log (n) times
```

$\rightarrow$ Time complexity is log (n)

Nested loops:

```
For (I=0; I<M; I++)
{       For (J=0; J<N; J++) {
------  }
 }
```

$\rightarrow$ Time complexity is (M*N)

```
For (I=0; I<M; I++)              //this occurs M times
{       For (J=0; J<N; J=J*2) {   //this occurs in log (n)
------  }
 }
```

$\rightarrow$ Time complexity is (M*(log (n))

Complexity of Algorithms:

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the input data.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria.

Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem.

We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data

The complexity function f(n) for certain cases are:

1. Best Case: The minimum possible value of f(n) is called the best case.
2. Average Case: The expected value of f(n).
3. Worst Case: The maximum value of f(n) for any key possible input.

Asymptotic analysis

As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space. So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity; we can decide which data structure is the best for an algorithm.

The various asymptotic notations are:

1. O (Big Oh notation)   { f(n) <= c*g(n) where n>=no }

2. Ω (Big Omega notation)   { f(n) >= c*g(n) where n>=no }

3. θ (Theta notation)   { 0<=c1*g(n) <=f(n) <= c2*g(n) where n>=no }

Where f(n) is your algorithmic runtime.

g(**n**) is an arbitrary time complexity you are trying to relate to your algorithm.

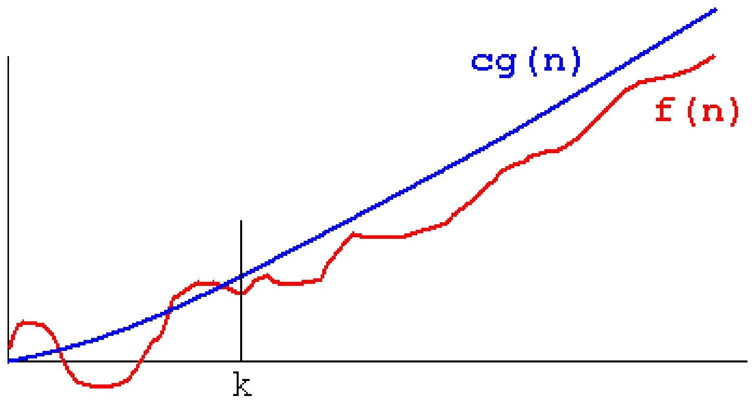
Big-Oh notation

Big O notation is an asymptotic notation that 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 the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

Definition: –

Given function f(n) and g(n) , we say that f(n) is O(g(n)) if there are exist positive constants c and n0 such that f(n) ≤ cg(n) for n ≥ n0 .



The growth rate of f(n) is less than or equal to the growth rate of g(n).

g(n) is an upper bound on f(n)

Rules for finding Big-oh notation:

1. If f(n) is a polynomial of degree d, then f(n) is O(n^d)

   – Drop lowest term

   – Drop constant factors

Example: f(n)=2n^3+3n^2+1 then f(n) = O(n^3)

2. Use the simplest possible class of function – Say "2n is O (n)"

Instead of "2n is O (n2)"

3. Use the simplest expression of the class – Say "3n+5 is O (n)" instead of "3n+5 is O (3n)"

4. If T1 (n) = O (f(n)) and T2 (n) = O(g(n)), then

   T1 (n) + T2 (n) = max (O(f(n)), O(g(n))),

   T1 (n) * T2 (n) = O (f (n) * g (n))

Example 1:

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

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

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

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

$$\boxed{2n+3<=5n}$$

Some more examples: f(n) -> lower bound

1.2n2 + 5n – 6 = O(2^n) //always 2^n grows more than f(n)

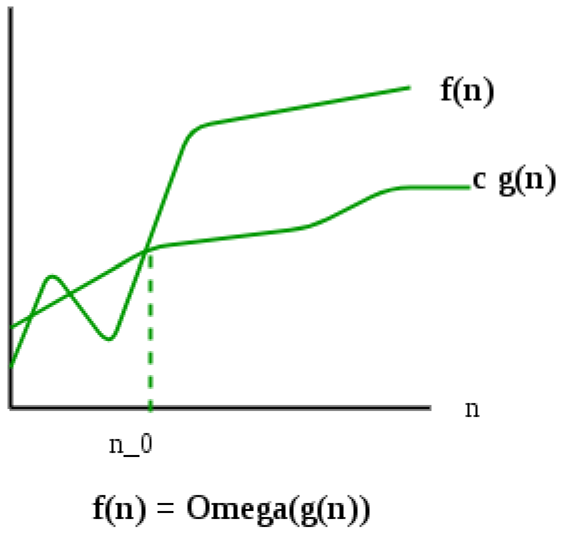
2. 2n2 + 5n – 6 = O(n^3) //cubic grows more than quadratic

3. 2n2 + 5n – 6 = O(n^2) //under some specific values of c it works

4. 2n2 + 5n – 6 ≠ O(n) //quadratic grows more than linear

Omega Notation (Ω)

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



Definition:

Given function f(n) and g(n) , we say that f(n) is Ω(g(n)) if there are exist positive constants c and n0 such that f(n) ≥ cg(n) for all n≥ n0

Let's consider a simple example.

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

f(n)>=c.g(n)

To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

2n+3>=c*n

Suppose c=1

2n+3>=n (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.

Example 2:

f(n) = 10n^2 + 4n + 2

Let us take g(n) = n2

c = 10 & n0 = 0

Let us check the above condition

10n2 + 4n + 2 ≥ 10n2 for all n ≥ 0

The condition is satisfied.

Hence f(n) = Ω(n^2).

Some more examples:

1.2n2 + 5n – 6 $\neq$ Ω(2^n)

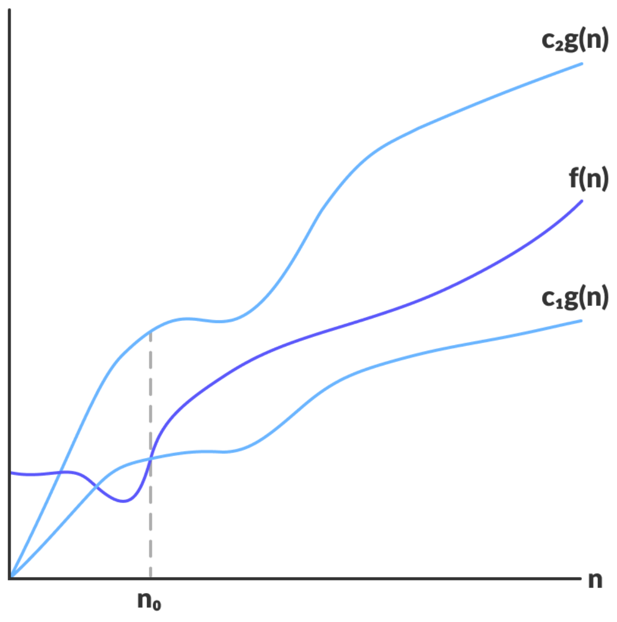
2. 2n2 + 5n – 6 $\neq$ Ω(n^3)

3. 2n2 + 5n – 6 = Ω(n^2)

4. 2n2 + 5n – 6 = Ω(n)

θ (Theta notation)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.



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

Definition:

f(n) = θ (g(n)), if there exist positive constants c1 , c2 , and n0 such that 0 ≤ c1 *g(n)≤ f (n) ≤c2 *g(n) for all n ≥ n0 }

Problems:

5n^2 is Θ(n^2)

f(n) = 5n^2 , g(n) = n^2

It is true for the values: c1 = 5, c2 = 6 and n0 >= 1

Note: f(n) is Θ(g(n)) if it is Ω(n2) and O(n2).

Theta (θ) Examples:

1. $2n2 + 5n – 6 \neq \Theta(2n)$ // not big=oh
2. $2n2 + 5n – 6 \neq \Theta(n3)$ //not omega
3. $2n2 + 5n – 6 = \Theta(n2)$ //both big-oh and omega so it is theta
4. $2n2 + 5n – 6 \neq \Theta(n)$ //not big-oh

Master theorem:

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(\boldsymbol{n}/b) + f(n)$$

where "$a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function".

There are 3 cases:

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

MASTER THEOREM EXAMPLES:

1. T(n) = 3T(n/2) + n2

Solution:

T(n) = 3T(n/2) + n2

Here,

a = 3

n/b = n/2

$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

2. T(n) = 4T (n/2) + n2

Solution:

In the same process as of (1); we can solve this

Case -2

3. $T(n) = 2^n T(n/2) + n^n$

Solution:

Does not apply, “a” should be constant but in this question “a” is not constant

4. T (n) = 64T (n/8) − n2 log n

Solution:

Does not apply, since f(n) is not positive

5. T (n) = 2T(n/2) + n/log n

Solution:

Does not apply (non-polynomial difference between f(n) and $n^{\log_b a}$)

Polynomial difference means: $f(n)/ n^{\log_b a} = n^c$ (for any real number c.)