1)A **ASYMPTOTIC NOTATIONS:**

- To choose best algorithm we use space complexity and time complexity.
- Asymptotic notation is another method of finding time complexity of an algorithm.
- Generally an algorithm that is asymptotically more efficient will be best choice.
- These are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For eg: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e the worst case.

when the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using Asymptotic notations.

There are 3 types of Asymptotic notations. They are

1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)

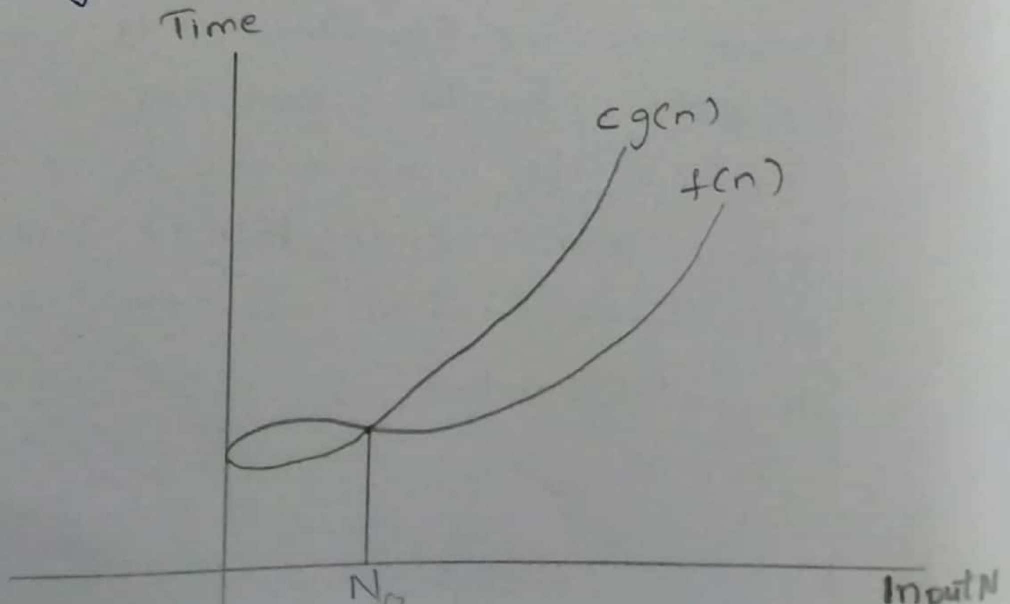
Big 'O' notation:

- It is a method of representing the upper bound of algorithm running time
- Let f(n) and g(n) are two non negative functions used for representing big O notations.

If $f(n) = O(g(n))$ if and only if $f(n) <= c * g(n)$ for all $n, n >= n_0$ where c, n_0 are positive constants. g(n) represents the upper bound of f(n).

$c > 0$ and $n_0 >= 1$.

Diagram:



Examples:

1) $3n+2 = O(n)$

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

$f(n) <= c*g(n)$

$3n+2 <= c*n$

$c = 4$

O(1): $g(n) = 1$

$3n+2 <= c*g(n)$

if $n=1$, $3*1+2 <= 4*1 \Rightarrow 5 <= 4$

which is false.

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

$3n+2 = c*g(n)$

if $n=2$, $3*2+2 <= 4*2 \Rightarrow 8 <= 8$

which is true

O(n^2): $g(n) = n$^2

$3n+2 <= c*g(n)$

if $n=2$, $3*2+2 = 4*2*2 \Rightarrow 8 <= 16$

which is true.

Hence O(1) is not possible.

O(n), O(n^2), ... are possible.

OMEGA 'Ω' notation:

- It is a method of representing the lower bound of algorithm running time.
- Let f(n) and g(n) are two non-negative functions used for representing omega notations.

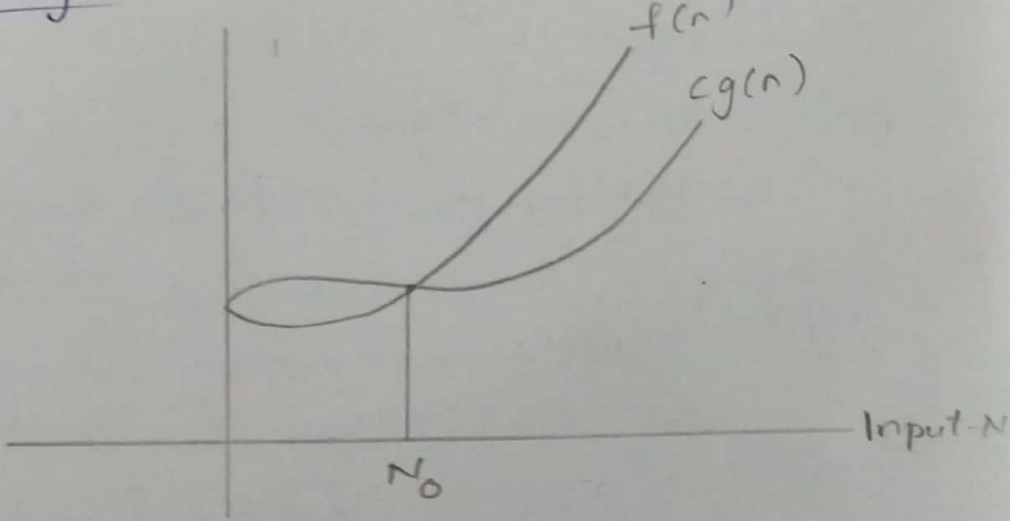
Scanned with CamScanner

If $f(n) = \Omega(g(n))$ if and only if $f(n) >= c * g(n)$

for all $n, n >= n_0$.

where c, n_0 are positive constants. $g(n)$ represents the lower bound of $f(n)$. $c > 0$ and $n_0 >= 1$.

Diagram:



Example:

1) $3n+2 = \Omega(n)$

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

$f(n) >= c * g(n) \Rightarrow 3n+2 >= c*n$, $c = 2$

$\Omega(1)$: $g(n) = 1$, $3n+2 >= c*g(n)$

If $n = 1$, $3*1+2 >= 2*1 \Rightarrow 5 >= 2$

whis is true.

$\Omega(n)$: $g(n) = n$, $3n+2 >= c*g(n)$

If $n = 2$, $3*2+2 >= 2*2 \Rightarrow 8 >= 4$

which is true

$\Omega(n^2)$: $g(n) = n^2$, $3n+2 >= c*g(n)$

If $n = 3$, $3*3+2 >= 2*3*3 \Rightarrow 11 >= 18$

which is false

Hence $\Omega(1), \Omega(n)$ is possible

$\Omega(n^2), \Omega(n^3)$ .. are not possible

THETA 'Θ' notation:

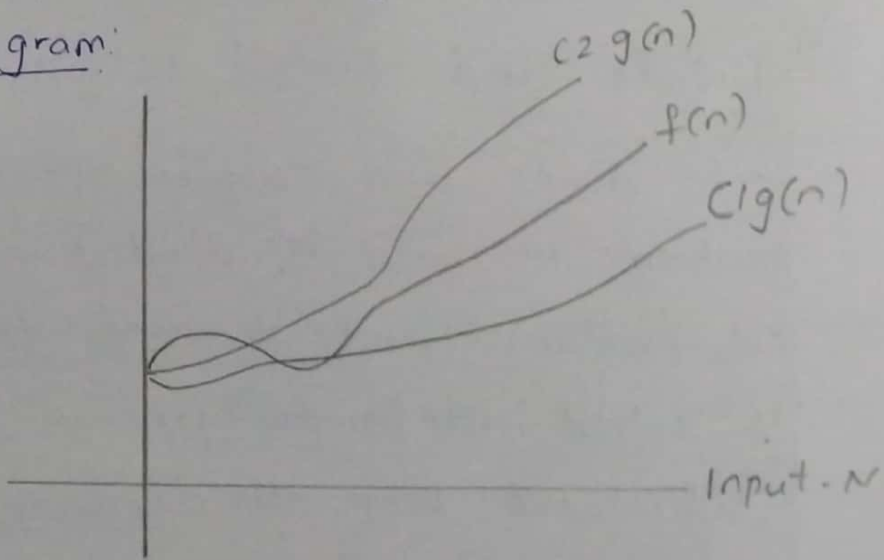
- It is a method of representing both lower and upper bound of algorithm running time.
- Let f(n) and g(n) are two non-negative functions used for representing theta notations

If $f(n) = \Theta(g(n))$ if and only if $c1 * g(n) <= f(n) <= c2 * g(n)$ for all $n, n >= n_0$.

where $c1, c2, n_0$ are positive constants. g(n) represents the average boundary level of f(n)

$c1 > 0, c2 > 0$ and $n_0 >= 1$.

Diagram:



Examples:

1) $3n + 2 = \Theta(n)$

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

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

$c1 * n <= 3n + 2 <= c2 * n$

$c1 = 3$ and $c2 = 4$

$\Theta(1)$:

$g(n)=1$, $3*n <= 3n+2 <= 4*n$

If $n=1$, $3*1 <= 3*1+2 <= 4*1$

$3 <= 5 <= 4$ which is false

$\Theta(n)$: $g(n)=n$, $3*n <= 3n+2 <= 4*n$.

If $n=2$, $3*2 <= 3*2+2 <= 4*2$

$6 <= 8 <= 8$ which is true

$\Theta(n^2)$:

$g(n)=n^2$, $3*n*n <= 3n+2 <= 4*n*n$

If $n=2$, $3*2*2 <= 3*2+2 <= 4*2*2$

$12 <= 8 <= 16$ which is false

hence $\Theta(n)$ is possible.

$\Theta(1)$, $\Theta(n^2)$, ... are not possible.

2)A **Divide and Control Method:**

- In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.
- when we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution

of an original problem.

Divide-and-conquer approach is given in a three-step process.

Divide/Break: This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub problem is further divisible. At this stage sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve: This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

- Most common usage
  - Break up problem of size n into two equal parts of size 1/2n.
  - Solve two parts recursively
  - Combine two solutions into overall solution in linear time
- Consequence
  - Brute force: n^2
  - Divide and conquer: n log n.

Data Abstraction:

Data Abstraction is the reduction of a particular body of data to a simplified representation of the whole.

Here we discuss on the topic Control Abstraction for Divide & Conquer

Divide and conquer approach is a three step approach to design a algorithm for a given problem.

a) Divide b) Conquer c) Combine.

Algorithm:

```
Algorithm DAC(P)
{
  if small(P) then
      return S(P);
  else
  {
   divide P into smaller instances
                    P1, P2, .. Pk
```

apply DAC to each of these problems

return Combine (DAC(P1), DAC(P2) .. DAC(Pk));

}

}

Analysis of Divide and Conquer:

We can compute the runtime of DAC as follows:

STEP 1:

Given problem size is n and if n is small then directly we compute the solution without applying DAC. So that the time required is $T(1)$.

STEP 2:

If the size of 'p' is 'n' and if it is not smaller then divide the problem into k sub problems n1, n2 .... nk. Each of size is 1/b where b is a constant

Hence the time required to compute the given problem is

$$T(n) = T(n1/b) + T(n2/b) + \ldots + T(nk/b) + f(n).$$

where $f(n)$ represents the time required to combine the sub-problems.

Substitation method:

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

$$a = 2, b = 2, f(n) = n$$

$$T(n) = 2T(n/2) + n;$$

By substitution method,

$$T(n) = 2\{2T((n/2)/2) + (n/2)\}$$

$$= 4T(n/4) + 2n$$

Again by substitution

$$= 4(2T((n/4)/2) + (n/2))$$

$$= 8T(n/8) + 3n$$

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

Let $3 = i$

Generalised form $T(n) = 2^I T(n/2^i) + in$

Let $n = 2^i$

$$\log n = \log 2^i$$

$$\log n = i \log 2.$$

$$\log n = i$$

substitute $n = 2^I$; $i = \log n$

$$T(n) = nT(n/n) + \log n^n$$

$$= nT(1) = n \log n.$$

$$T(n) = O(n \log n).$$

STRASSON'S MATRIX MULTIPLICATION

Given two square matrices A and B of size nxn each, find their multiplication matrix.

Traditional Method:

It is a simple way to multiply two matrices.

```
void multiply (int A[][N], int B[][N],
                                int C[][N])
{ for(int i=0; i<N; i++)
  {
      for(int j=0; j<N; j++)
      {      C[i][j] = 0;
          for (int k=0; k<N; k++)
          {
              C[i][j] += A[i][k]*B[k][j].
          }
      }
  }
}
```

* Time complexity is $O(N^3)$.

- **Divide and Conquer:**

Following is simple Divide and Conquer method to multiply two square matrices.

- Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram.
- Calculate following values recursively. $ae+bg$, $af+bh$, $ce+dg$ and $cf+dh$.

$$\underbrace{\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]}_{A} \times \underbrace{\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]}_{B} = \underbrace{\left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]}_{C}$$

A,B and C are square matrices of size $N \times N$

a, b, c and d are submatrices of A, of size $N/2 \times N/2$

e, f, g and h are submatrices of B of size $N/2 \times N/2$.

- In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions.

- Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as $T(N) = 8T(N/2) + O(N^2)$

- From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above Traditional method.

- In the above divide and conquer method the main component for high time complexity is 8 recursive calls

- The idea of strassen's method is to reduce the number of recursive calls to 7.

- Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above diagram.

- But in strassen's method, the four sub-matrices of result are calculated using following formulae.

$P1 = a(f-h)$ $\quad$ $P2 = (a+b)h$

$P3 = (c+d)e$ $\quad$ $P4 = d(g-e)$

$P5 = (a+d)(e+h)$ $\quad$ $P6 = (b-d)(g+h)$

$P7 = (a-c)(e+f)$

The $A \times B$ can be calculated using above seven multiplications

Following are values of four sub-matrices of result C.

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} P5+P4-P2+P6 & P1+P2 \\ \hline P3+P4 & P1+P5-P3-P7 \end{array}\right]$$

$\qquad A \qquad\qquad B \qquad\qquad C$

A, B and C are square matrices of size $N \times N$.

a, b, c and d are submatrices of A, of size $N/2 \times N/2$.

e, f, g and h are submatrices of B, of size $N/2 \times N/2$

P1, P2, P3, P4, P5, P6 and P7 are submatrices of size $N/2 \times N/2$.

- Time Complexity Analysis of Strassen's Method.
- Addition and Subtraction of two matrices takes $O(N^2)$ time.
- so time complexity can be written as

$T(N) = 7T(N/2) + O(N^2)$ from Master's Theorem, time complexity of above method is $O(N^{\log 7})$ which is approximately $O(N^{2.8074})$

Scanned with CamScanner

• Generally strassen's Method is not preferred for practical applications for following reasons.

— The constants used in strassen's method are high and for a typical application Traditional method works better.

- for sparse matrices, there are better methods especially designed for them.

- The submatrices in recursion take extra space.

- Because of the limited precision of computer arithmetic on noninteger values larger errors accumulate in strassen's algorithm than in Naive method