Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

```
    Let f (n) = an²+bn+c
```

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

Asymptotic notation:

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

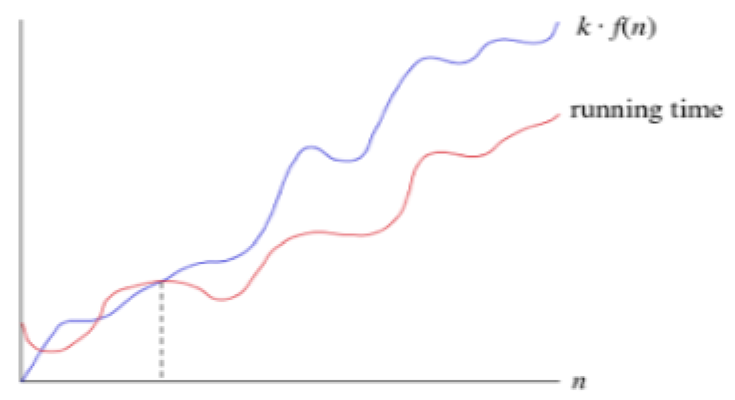
Asymptotic Notations:

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. Big-oh notation: Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) **for** n>n0n>n0 in all **case**

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



ASYMPTOTIC UPPER BOUND

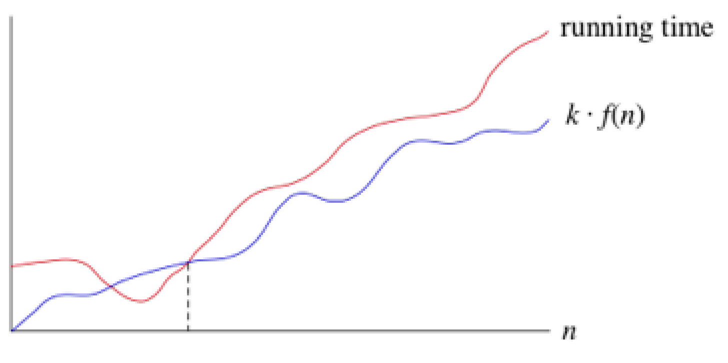
For Example:

1. 1. $3n+2=O(n)$ as $3n+2 \leq 4n$ **for** all $n \geq 2$
2. 2. $3n+3=O(n)$ as $3n+3 \leq 4n$ **for** all $n \geq 3$

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega () Notation: The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

```
F (n) ≥ k* g (n) for all n, n≥ n0
```



ASYMPTOTIC LOWER BOUND

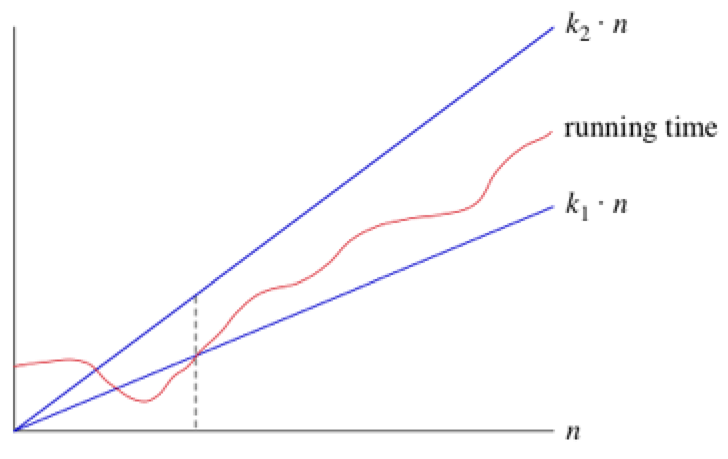
For Example:

```
  f (n) =8n²+2n-3≥8n²-3
        =7n²+(n²-3)≥7n² (g(n))
Thus, k1=7
```

Hence, the complexity of **f (n)** can be represented as Ω (g (n))

3. Theta (θ): The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

```
  k1 * g (n) ≤ f(n)≤ k2 g(n)for all n, n≥ n0
```



ASYMPTOTIC TIGHT BOUND

For Example:

```
3n+2= θ (n) as 3n+2≥3n and 3n+2≤ 4n, for n
    k1=3,k2=4, and n0=2
```

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

Analyzing Algorithm Control Structure

To analyze a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and therefore, each instruction must be analyzed separately to analyze overall performance. However, there are some algorithm control structures which are present in each programming code and have a specific asymptotic analysis.

Some Algorithm Control Structures are:

1. Sequencing
2. If-then-else
3. for loop
4. While loop

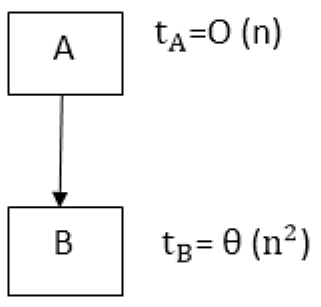
---

1. Sequencing:

Suppose our algorithm consists of two parts A and B. A takes time t_A and B takes time t_B for computation. The total computation "$t_A + t_B$" is according to the sequence rule. According to maximum rule, this computation time is (max (t_A,t_B)).

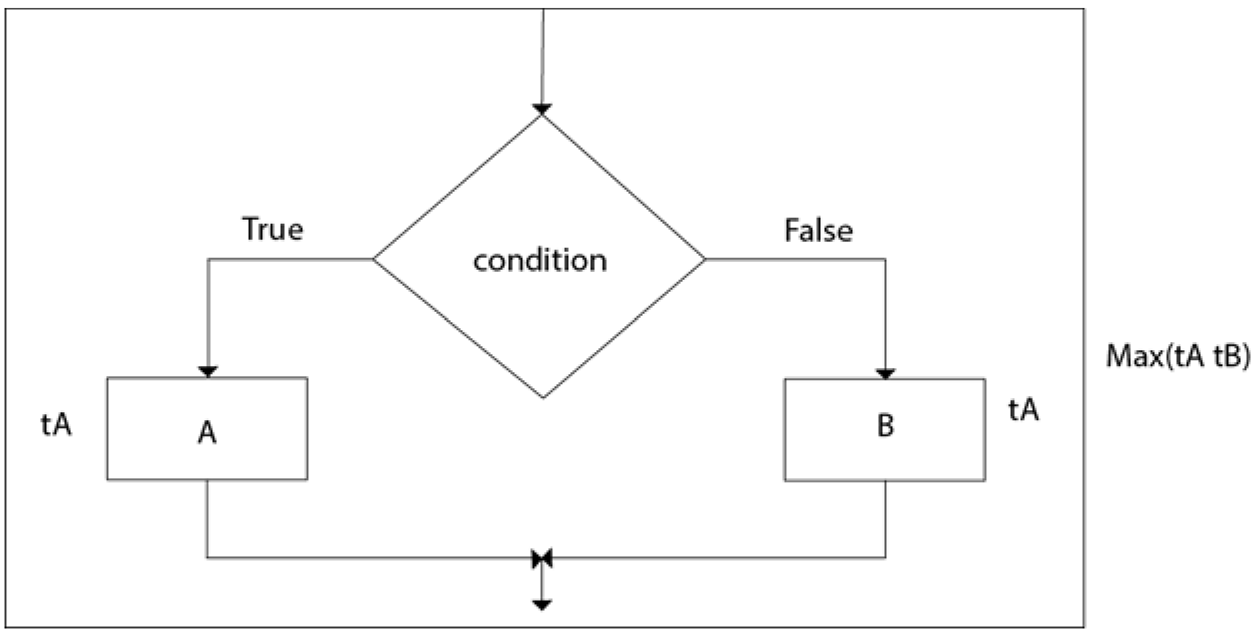
Example:

```
Suppose tA =O (n) and tB = θ (n²).
Then, the  total computation time can be calculated as
```



```
Computation Time = tA + tB
 = (max (tA,tB)
 = (max (O (n), θ (n²)) = θ (n²)
```

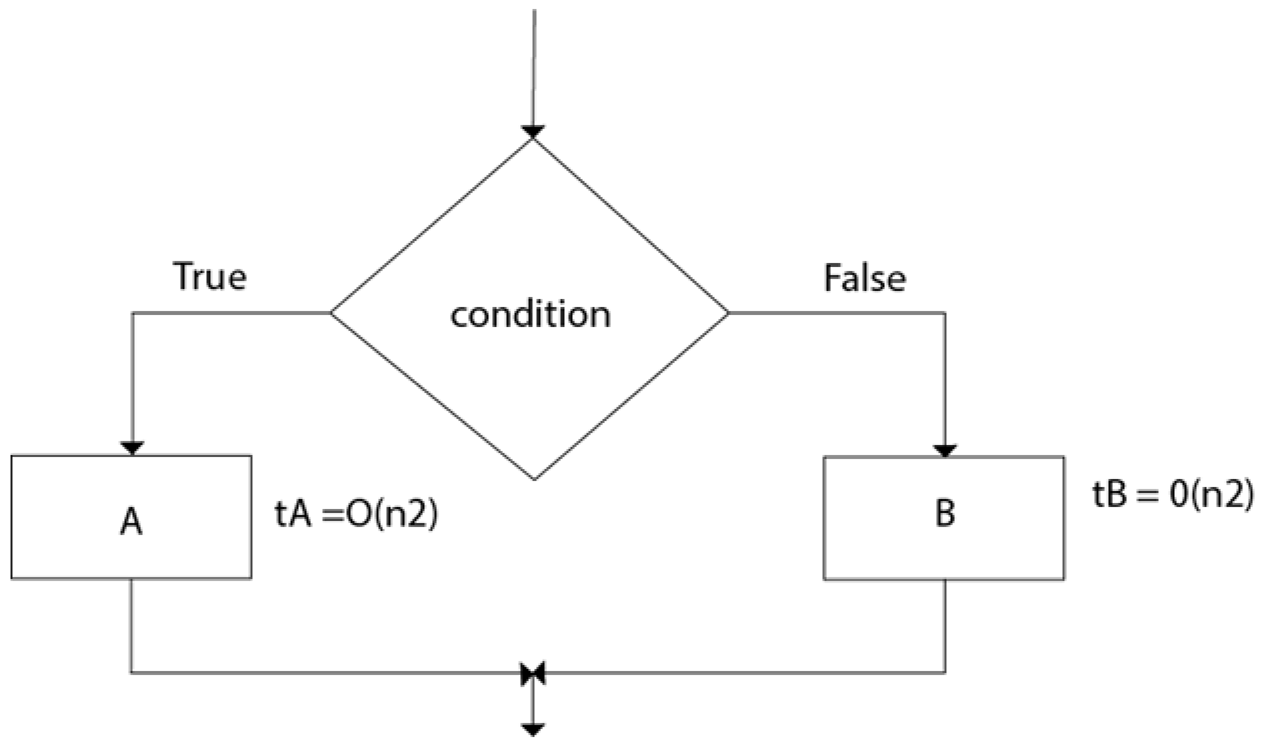
2. If-then-else:



The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max (t_A,t_B).

Example:

```
Suppose tA = O (n²) and tB = θ (n²)
Calculate the total computation time for the following:
```



```
Total Computation = (max (tA,tB))
                  = max (O (n²), θ (n²) = θ (n²)
```

3. For loop:

The general format of for loop is:

1. For (initialization; condition; updation)
2.
3. Statement(s);

Complexity of for loop:

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the **inner** loop execute a total of N * M times. Thus, the total complexity for the two loops is O (N2)

Consider the following loop:

```
1. for i ← 1 to n
2. {
3.        P (i)
4. }
```

If the computation time t_i for (P_I) various as a function of "i", then the total computation time for the loop is given not by a multiplication but by a sum i.e.

```
1. For i ← 1 to n
2. {
3.          P (i)
4. }
```

Takes $\sum_{i=1}^{n} t_i$ time, i.e.$\sum_{j=1}^{n} \theta(1)=\theta\sum_{i=1}^{n} \theta(n)$

If the algorithms consist of nested "for" loops, then the total computation time is

```
For i ← 1 to n
 {
                              $\sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij}$
      For j ←  1 to n
    {
      P (ij)
    }
 }
```

Example:

Consider the following "for" loop, Calculate the total computation time for the following:

1. For i ← 2 to n-1

```
2.      {
3.          For j ← 3 to i
4.          {
5.                      Sum ← Sum+A [i] [j]
6.                 }
7.              }
```

Solution:

The total Computation time is:

$$\sum_{i=2}^{n-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{n-1} \sum_{j=3}^{i} \theta\,(1)$$

$$\sum_{i=2}^{n-1} \; \theta(i)$$

$$= \theta(\textstyle\sum_{i=2}^{n-1} i) = \theta\left(\frac{m^2}{2}\right) + \theta(m)$$

$$= \theta(m^2)$$

4. While loop:

The Simple technique for analyzing the loop is to determine the function of variable involved whose value decreases each time around. Secondly, for terminating the loop, it is necessary that value must be a positive integer. By keeping track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analyzing "while" loop is to treat them as recursive algorithms.

Algorithm:

```
1.  1. [Initialize] Set k: =1, LOC: =1 and MAX: = DATA [1]
2.  2. Repeat steps 3 and 4 while K≤N
3.  3.  if MAX<DATA [k],then:
4.      Set LOC: = K and MAX: = DATA [k]
5.  4. Set k: = k+1
6.     [End of step 2 loop]
7.  5. Write: LOC, MAX
8.  6. EXIT
```

Example:

The running time of algorithm array Max of computing the maximum element in an array of n integer is O (n).

Solution:

```
1.    array Max (A, n)
2.  1. Current max ← A [0]
3.  2. For i ←  1 to n-1
4.  3. do if current max < A [i]
5.  4. then  current max ← A [i]
6.  5. return current max.
```

The number of primitive operation t (n) executed by this algorithm is at least.

1. 2 + 1 + n +4 (n-1) + 1=5n
2. 2 + 1 + n + 6 (n-1) + 1=7n-2

The best case T(n) =5n occurs when A [0] is the maximum element. The worst case T(n) = 7n-2 occurs when element are sorted in increasing order.

We may, therefore, apply the big-Oh definition with c=7 and n_0=1 and conclude the running time of this is O (n).