What is Running Time Analysis?

It is the process of determining how processing time of a problem increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. Example:

- Size of an array
- Polynomial degree
- Number of elements in matrix
- Number of bits in the binary representation of the input
- Vertices and edges in the graph

How to Compare Algorithms?

To compare algorithms, let us define a few objective measures

Execution Times? Not a good measure as execution time are specific to a particular computer

Number of statements executed? Not a good measure as the number of statements varies with programming languages as well as with the style of individual programmer

Ideal Solution? Let us assume that we express the running time of a given algorithm as a function of input size n(i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc. We measure the total number of basic operations (additions, subtractions, increments, multiplications, divisions,modulo etc.) performed as a function of input size.

What is the Rate of Growth?

The rate at which running time increases as a function of input is called rate of growth.

Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you over there and asks you what you are buying, then in general you say buying a car. This is because the cost of the car is high compared to the cost of the bicycle (approximately the cost of the bicycle to the cost of the car).

Total cost = cost of car + cost of bicycle

Total cost ~ cost of car (approximation)

Similarly,

For a given function, ignore the low order terms that are relatively insignificant.

$n^4 + 2n^3 + 100n + 500 \sim n^4$ (for large value of input size, n)

Commonly used rate of growths

Time Complexity	Name
1	Constant
logn	Logarithmic
n	Linear
nlogn	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Types of analysis

To analyse the given algorithm, we need to know with which inputs does the algorithm take less time and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions:one for the case where it takes less time and another for the case where it takes more time.

Three types of analysis are generally performed:

• **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.

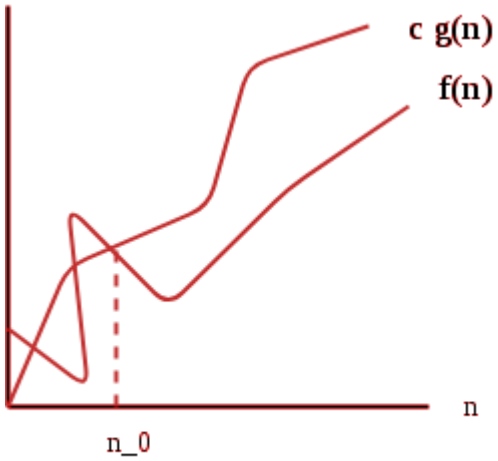
• **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.

• **Average case:** The average case gives an idea about the average running time of the given algorithm.

Asymptotic Notations

We aim to identify upper and lower bounds by doing worst case, average case and best case analysis. To represent the upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function f(n).

- **Big-O Notation**
  This notation gives the **tight upper bound** of the given function. Generally, it is represented as f(n)=O(g(n)). That means at larger values of N the upper bound of a f(n) is g(n).
  For example: if f(n) = $n^4 + 2n^3 + 100^n + 500$ is the given algorithm then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.



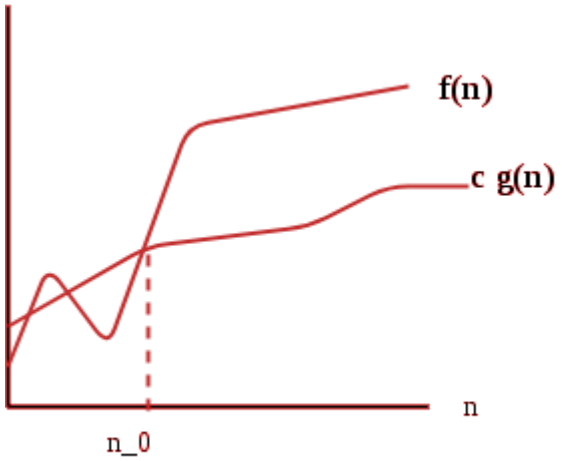
f(n) = O(g(n))

O-notation is defined as
O(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= f(n) <= cg(n) for all n > n_0 }. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give the smallest rate of growth g(n) which is greater than or equal to the given algorithm's rate of growth.

Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0 the rate of growth could be different. n_0 is called the threshold for the given function.

- **Omega-Ω Notation**
  Similar to the O discussion, this notation gives the **tighter lower bound** of the given algorithm and we represent it as f(n) = Ω(g(n)). That means, at larger values of n, the tighter lower bound of f(n) is g(n).

  For example, if f(n) = $100n^2+10n+50$, g(n) is $\Omega(n^2)$.

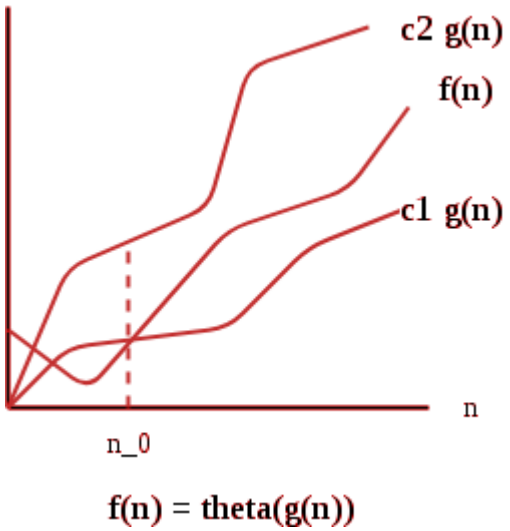


f(n) = Omega(g(n))

Ω-notation is defined as
Ω(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= cg(n) <= f(n) for all n > n_0 }.
g(n) is an asymptotic lower bound for f(n). Our objective is to give the largest rate of growth g(n) which is less than or equal to the given algorithms' rate of growth.

- **Theta-Θ Notation**
  This notation decides whether the upper and the lower bound of the given function(algorithm) are the same. The average running time of an algorithm is always **between the lower bound and the upper bound**. If the upper bound(O) and the lower bound(Ω) give the same result, then the Θ notation will also have the same rate of growth. As an example, let us assume that f(n) = 10n+ n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in the best case is g(n) = O(n).

  In this case, the rate of growth in the best case and worst case are the same. As a result the average case will also be the same. For a given function(algorithm), if the rates of growth for O and Ω are not same, then the rate of growth for Θ case may not be the same.



Θ-notation is defined as
Θ(g(n)) = {f(n): there exists positive constants c and n_0 such that 0 <= c_1g(n) <= f(n) <= c_2g(n) for all n > n_0 }. g(n) is an asymptotic tight bound for f(n). Θ(g(n)) is the set of functions with the same order of growth as g(n).

Note: In the analysis, we generally focus on the upper bound(O) because knowing the lower bound(Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound(O) and lower bound(Ω) are the same .

Guidelines for Asymptotic Notations

There are some general rules to help us determine the running time of an algorithm.

- Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations .

```
//   executes n times
for i from 1 to n
      m = m + 2;       //   constant time c
      i++;
```

Total time = (a constant c) x n = cn = O(n)

- Nested loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//   outer loop executed n times
for i from 1 to n {
     //   inner loop executed n time
     i++;
     for j from 1 to n {
          k = k+1;    //   constant time
          j++
     }
}
```

Total time = c x n x n = cn^2 = $O(n^2)$

- Consecutive Statements: Add the time complexities of each statement.

```
x = x + 1;                 //  constant time
//   executed n times
for i from 1 to n {
      m = m + 2;       //   constant time c
      i++;
}
//   outer loop executed n time
for i from 1 to n {
     //   inner loop executed n time
     for j from 1 to n {
          k = k + 1;   //   constant time
          j++;
      }
}
```

Total time = $c_0 + c_1n + c_2n^2 = O(n^2)$

- If-then-else statements: Worst case running time: the test, plus either the then part or the else part (whichever is the largest)

```
//   condition check: constant time
if(length() == 0)
     return false;
else {
     // else part: (constant + constant) * n
     for n = 0 to n< length {
          //   another if: constant + constant (no else part)
          if( !list[n].equals(otherList.list[n]) )
               return false;
     }
}
```

Total time = $c_0 + (c_1 + c_2) * n = O(n)$

- Logarithmic Complexity: Algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction(usually by 1/2).

```
for i from 1 to n
    i = i * 2
```

If we observe carefully, the value of i is doubling every time. Initially i=1, in the next step i=2, and in subsequent steps i=4,8 and so on. Let us assume that the loop is executing some k times. At the kth step $2^k = n$, and at (k+1)th step we come out of the loop.

Taking logarithm on both sides

$log(2^k) = logn$

$klog2 = logn$

$k = logn$

Total time = O(logn)

Recurrence relations and Master's Theorem

Recurrence Relation: A recurrence relation is an equation that recursively defines a sequence where the next term of the sequence is a function of the previous terms. In other words we express the n^{th} term of the sequence(F_n) as a function of the previous terms F_i(i<n).
For example: Fibonacci Series: $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$.
In Fibonacci Series the n^{th} term can be expressed as the sum of previous 2 terms. The base case for n <= 1 is also defined above. So the second term is $F_2 = F_0 + F_1 = 1$, $F_3 = 2$, $F_4 = 3$ and so on.

There are various algorithms whose running time can be described in terms of a recurrence relation, and solving the recurrence relation gives us the time complexity of the algorithm.
For example: Let us consider the recursive version of binary search to find the position of an element in a sorted array.
In **Binary search** we are given a sorted array of elements and we aim to find the target element. We do this by comparing the target with the middle element and compress the search space to half of its original size in every pass. If the target element is greater than the middle element we move our left pointer to the middle position else we move the right pointer to the middle position.

```
/*
      array from leftidx(0) to rightidx(arr.length-1) is considered
*/
function binarySearch(arr, leftidx , rightidx , target)
       //   base case : element not found
       if  leftidx > rightidx
                return -1

        middle = (leftidx + rightidx) / 2
        if  arr[middle] equals target
                return middle

        else if arr[middle] > target
                return binarySearch(arr , leftidx , middle - 1 , target)

        else
                return  binarySearch(arr, middle + 1, rightidx, target)
```

The recurrence relation for the above recursive function can be defined as -
Let T(n) denote the time taken by an algorithm to execute for input size 'n'. Hence, **T(n) = T(n/2) + 1** where T(1) = 1.

How to solve for T(n) ?
Solving the above recurrence relation and in general the recurrence relations for the algorithms following the divide and conquer paradigm can be easily done using the Master's Theorem.

Master's Theorem:
Master's theorem can be used to solve the recurrence relations of the type:
T(n) = a.T(n/b) + f(n) a>=1, b>1

Here,

- **n:** The size of the problem.
- **a:** The number of subproblems in the recursion.
- **n/b:** The size of each subproblem.(It is assumed that the size of all the subproblems are the same)
- **f(n):** The cost of work done outside the recursive calls, which basically includes the cost of dividing the problem and merging the solutions of the subproblems.

Note: Here 'a' and 'b' are constants and f(n) is asymptotically a positive function. In other words, for sufficiently large input size 'n', f(n) > 0 and T(n) is a monotonically increasing function.

The solution of recurrence relations of the form T(n) = a.T(n/b) + f(n) as given by Master's theorem where the whole idea is based upon the comparison of f(n) and n^($\log_b$a) and determining which of them is the dominating factor -

- **Case 1:** If f(n) = O(n^($\log_b$a - ε)), for some ε > 0, then T(n) = Θ(n^($\log_b$a)).
  This case can be interpreted as the worst case of f(n) is n^($\log_b$a - ε), which is less than n^($\log_b$a) so n^($\log_b$a) takes more time and dominates.
- **Case 2:** If f(n) = Θ(n^($\log_b$a)), then T(n) = Θ(n^($\log_b$a) * logn).
  If f(n) is also Θ(n^($\log_b$a)), then the time taken will be Θ(n^($\log_b$a)*logn).
- **Case 3:** If f(n) = Ω(n^($\log_b$a + ε)), for some ε > 0, and if a.f(n/b) <= c.f(n) for some c < 1 and all sufficiently large 'n', then T(n) = Θ(f(n)).
  Since the best case of f(n) is n^($\log_b$a - ε), so the best case of f(n) is greater than n^($\log_b$a), hence f(n) dominates.

Coming back to our problem of solving for T(n) for binary search, where T(n) = T(n/2) + 1, here the recurrence relation satisfies all the conditions of the master's theorem, where a = 1 and b = 2.

=> Calculating $\log_b a = \log_2 1 = 0$, so n^($\log_b$a) = n^(0) = 1.
=> Since f(n) = 1 = n^($\log_b$a)
=> Case 2: f(n) = Θ(n^($\log_b$a))
=> Hence, T(n) = Θ(logn)

Limitations of Master's Theorem

- We cannot use Master's theorem if T(n) is not monotone, for example T(n) = sin(n).
- f(n) must be a polynomial
- If a is not a constant, for example a = 2*n, or b cannot be expressed as a constant, for example T(n) = T(√n), then the master's theorem cannot be applied.

Let us now look at a few examples of Master theorem applications

a) $T(n) = 8T(n/2) + 1000n^2$

   Here,
   a = 8
   b = 2
   $f(n) = 1000n^2 = \Theta(n^2)$
   $\log_b a = \log_2 8 = 3 > 2$
   ie. f(n) < n^$\log_b$a-ϵ , where, ϵ is a constant.
   Case 1 implies here.
   Thus, $T(n) = f(n) = \Theta(n^{\log_b(a)}) = \Theta(n^3)$

b) $T(n) = 2T(n/2) + \Theta(n)$

Here,
a = 2
b = 2
f(n) = n
$\log_b a = \log_2 2 = 1 = 1$
ie. f(n) = n^$\log_b a$
Case 2 implies here.
Thus, $T(n) = f(n) = \Theta(n^{\log_2(2)} \log n) = \Theta(n \log n)$

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

Here,
a = 3
b = 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, $T(n) = f(n) = \Theta(n^2)$