Analysis of Algorithms: introductory idea

Sometimes, there is more than one way to solve a problem. We need to learn how to compare the performance of different algorithms and choose the best one to solve a particular problem. In Analysis of algorithms, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size. This analysis is machine independent i.e. the machine dependent constants can always be ignored after certain values of input size.

To summarize, time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the size of the input.

We can have three cases to analyze an algorithm:

1) Worst Case

2) Average Case

3) Best Case

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

Best Case Analysis (Not so useful, rarely done)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

If not mentioned, in general we always consider worst case analysis to measure efficiency of an algorithm.

Abhishek Dey
Assistant Professor
Department of Computer Science
Bethune College, Kolkata

How to measure efficiency

The main idea of analysis of algorithm is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time (or space) complexity of algorithms.

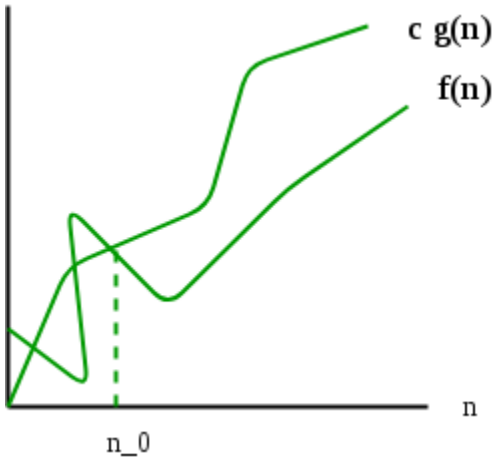
Big O Notation:

The Big O (or Big –Oh) notation defines an upper bound of an algorithm; it bounds a function from above (O stands for order of). Suppose, operation of an algorithm can be described by a function f(n), n is the input size. Time complexity of this algorithm is O(g(n)) signifies that, the algorithm always performs less than or equal to $c.n^2$ (c is a positive real constant) operations for increasing values of n. This stability is obtained only after some input size n_0.

Mathematically this is expressed as,

$f(n) \in O(g(n))$ if there exists a positive integer n_0 and a real positive constant c, such that

$f(n) \leq c.g(n) \; \forall \; n \geq n_0$



The fastest possible running time for any algorithm is O(1), commonly referred to as Constant Running Time. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable. In actual cases, the performance (Runtime) of an algorithm depends on input size n.

Abhishek Dey
Assistant Professor
Department of Computer Science
Bethune College, Kolkata

Some examples:

O(1):

Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function. For example swap() function has O(1) time complexity.

A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
  for (i = 1; i <= 2000; i++)
{
      // some O(1) expressions
 }
```

O(n):

Time Complexity of a loop is considered as O(n) if the loop variables is incremented or decremented by a constant amount. For example following functions have O(n) time complexity.

```
for (i = 1; i <= n; i += 2)
{
// some O(1) expressions
}

for ( i = n; i > 0; i -= 5)
{
// some O(1) expressions
}
```

Abhishek Dey
Assistant Professor
Department of Computer Science
Bethune College, Kolkata

$O(n^c)$:

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
 for (i = 1; i <=n; i += 1)
{
        for (j = 1; j <=n; j += 2)
        {
                // some O(1) expressions
        }
}
```

```
for (i = n; i >0; i -= 1)
 {
        for (j = 1; j <=n; j += 2)
        {
                // some O(1) expressions
        }
 }
```

$O(\log_c n)$:

Time Complexity of a loop is considered as $O(\log_c n)$ if the loop variables is divided or multiplied by a constant amount c.

```
 for (i = 1; i <=n; i *= c)
{
   // some O(1) expressions
 }
```

```
 for (i = n; i > 0; i /= c)
{
   // some O(1) expressions
 }
```

Abhishek Dey
Assistant Professor
Department of Computer Science
Bethune College, Kolkata

The algorithms can be classified as follows from the best-to-worst performance (Running Time Complexity):

A logarithmic algorithm i.e. O(logn) - Runtime grows logarithmically in proportion to n.

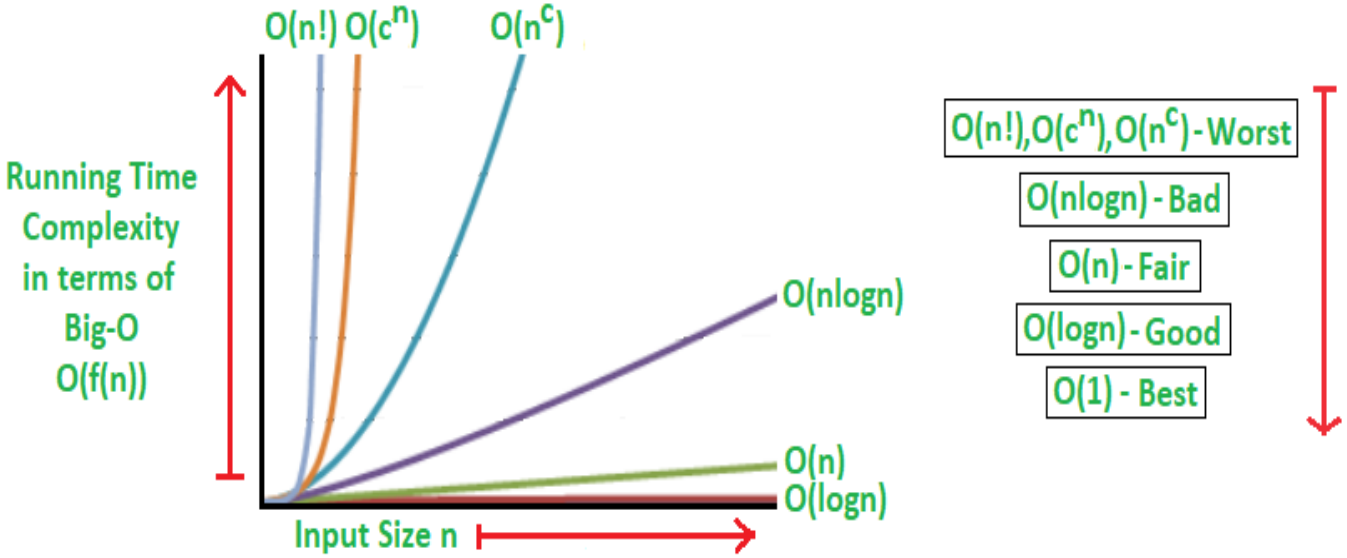
A linear algorithm i.e. O(n) - Runtime grows directly in proportion to n.

A superlinear algorithm i.e. O(nlogn) - Runtime grows in proportion to n.

A polynomial algorithm i.e. $O(n^c)$ - Runtime grows quicker than previous all based on n.

An exponential algorithm i.e. $O(c^n)$ - Runtime grows even faster than polynomial algorithm based on n.

A factorial algorithm i.e. O(n!) - Runtime grows the fastest and becomes quickly unusable for even small values of n.



Reference books and websites:

1. Data Structures Using C by Reema Thareja
2. Data Structures Through C In Depth by S.K.Srivastava and Deepali Srivastava
3. https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/
4. https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Abhishek Dey
Assistant Professor
Department of Computer Science
Bethune College, Kolkata