Complexity Analysis

Big – O notation

- It is most commonly used notation for specifying asymptotic complexity i.e rate of function growth.
- It refers to upper bound of functions.

Definition 1: $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.

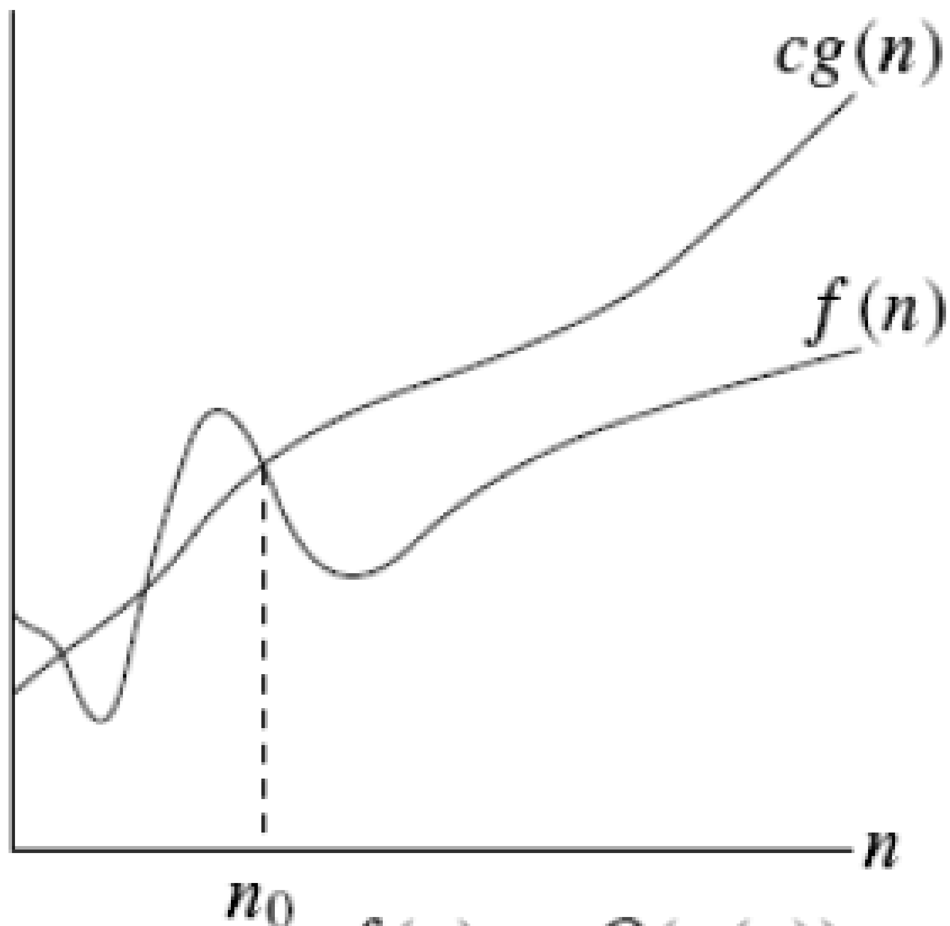
This definition reads: f is big-O of g if there is a positive number c such that f is not larger than cg for sufficiently large ns, that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

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

where $g(n) = n^2$

lim n->∞ f(n) / g(n) = c //\`c closer to 0\`

Graph for O Notation



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

Big–Oh Notation (O) :

>> $O(g(n)) = \{ f(n)$: there exist positive constants c and no such that $0 < f(n) < c*g(n)$ for all $n > no$ }.

>> It is asymptotic upper bound.

>>The function $f(n) = O(g(n))$ iff there exist positive constants c and no such that $f(n) \leq c * g(n)$ for all n, $n \geq no$.

>> The statement $f(n) = O(g(n))$ states only that g(n) is an upper bound on the value of f(n) for all n, $n \geq no$.

>> Eg :

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

$3n + 2 \leq 4n$ for all $n \geq 2$.

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

$3n + 3 \leq 4n$ for all $n \geq 3$.

3. $100n + 6 = O(n)$

$100n + 6 \leq 101n$ for all $n \geq 6$.

Basic rules for finding **Big - O**

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).

Difference between little-oh and big-oh

f ∈ O(g) says, essentially

For **at least one** choice of a constant *k* > 0, you can find a constant *a* such that the inequality 0 <= f(x) <= k g(x) holds for all x > a.

f ∈ o(g) says, essentially

For **every** choice of a constant *k* > 0, you can find a constant *a* such that the inequality 0 <= f(x) < k g(x) holds for all x > a.

These both describe upper bounds, although somewhat counter-intuitively, Little-o is the stronger statement. There is a much larger gap between the growth rates of f and g if f ∈ o(g) than if f ∈ O(g).

Example 1

```
//linear
for(int i = 0; i < n; i++) {
    cout << i << endl;
}
```

- Ans: $O(n)$

Example 2

```
//quadratic
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++){
        //do swap stuff, constant time
    }
}
```

- Ans $O(n^2)$

Example 3

```
//quadratic
for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++){
        //do swap stuff, constant time
    }
}
```

- Ans: (n(n+1)/2). This is still in the bound of O(n^2)

Example 4

```
for(int i = 0; i < 2*n; i++) {
    cout << i << endl;
}
```

- At first you might say that the upper bound is O(2n); however, we drop constants so it becomes O(n)

Example 5

```
//linear
for(int i = 0; i < n; i++) {
    cout << i << endl;
}

//quadratic
for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++){
        //do constant time stuff
    }
}
```

- Ans : In this case we add each loop's Big O, in this case n+n^2. O(n^2+n) is not an acceptable answer since we must drop the lowest term. The upper bound is O(n^2). Why? Because it has the largest growth rate

Example 6

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < 2; j++){
        //do stuff
    }
}
```

- Ans: Outer loop is 'n', inner loop is 2, this we have 2n, dropped constant gives up O(n)

Example 7

```
for(int i = 1; i < n; i *= 2) {
    cout << i << endl;
}
```

- There are n iterations, however, instead of simply incrementing, 'i' is increased by 2*itself each run. Thus the loop is log(n).

Example 8

```
for(int i = 0; i < n; i++) { //linear
    for(int j = 1; j < n; j *= 2){ // log (n)
        //do constant time stuff
    }
}
```

- Ans: n*log(n)

Examples

$3 * n^2 + n/2 + 12 \quad \in \quad \mathbf{O(n^2)}$

$4*n*\log_2(3*n+1) + 2*n-1 \quad \in \quad \mathbf{O(n * \log n)}$

Typical Complexities

Complexity	Notation	Description
constant	O(1)	Constant number of operations, not depending on the input data size, e.g. n = 1 000 000 → 1-2 operations
logarithmic	O(log n)	Number of operations propor-tional of $\log_2(n)$ where n is the size of the input data, e.g. n = 1 000 000 000 → 30 operations
linear	O(n)	Number of operations proportional to the input data size, e.g. n = 10 000 → 5 000 operations

Typical Complexities

Complexity	Notation	Description
quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data, e.g. n = 500 → 250 000 operations
cubic	$O(n^3)$	Number of operations propor-tional to the cube of the size of the input data, e.g. n = 200 → 8 000 000 operations
exponential	$O(2^n)$, $O(k^n)$, $O(n!)$	Exponential number of operations, fast growing, e.g. n = 20 → 1 048 576 operations

Time Complexity and Speed

Complexity	10	20	50	100	1 000	10 000	100 000
O(1)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n*log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n^2)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	3-4 min
O(n^3)	< 1 s	< 1 s	< 1 s	< 1 s	20 s	5 hours	231 days
O(2^n)	< 1 s	< 1 s	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 s	hangs	hangs	hangs	hangs	hangs	hangs
O(n^n)	3-4 min	hangs	hangs	hangs	hangs	hangs	hangs

Practical Examples

- O(n): printing a list of n items to the screen, looking at each item once.
- O(log n): taking a list of items, cutting it in half repeatedly until there's only one item left.
- O(n^2): taking a list of n items, and comparing every item to every other item.

How to determine Complexities Example 1

- Sequence of statements

  statement 1;

  statement 2;

  ...

  statement k;

- total time = time(statement 1) + time(statement 2) + ... + time(statement k)

- If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: **O(1).**

Example 2

- if-then-else statements

```
 if (condition)
{
        sequence of statements 1
}
else
{
        sequence of statements 2
}
```

- Here, either sequence 1 will execute, or sequence 2 will execute.
- Therefore, the worst-case time is the slowest of the two possibilities: **max(time(sequence 1), time(sequence 2)).**
- For example, if sequence 1 is **O(N)** and sequence 2 is **O(1)** the worst-case time for the whole if-then-else statement would be **O(N)**.

Example 3

- for loops

```
for (i = 0; i < N; i++)
{
        sequence of statements
}
```

- The loop executes N times, so the sequence of statements also executes N times.
- Since we assume the statements are **O(1)**, the total time for the for loop is **N * O(1)**, which is **O(N)** overall.

Example 4

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < M; j++)
    {
        sequence of statements ;
    }
}
```

- 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 complexity is **O(N * M).**
- In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is **O(N^2).**

Example 5

```
for (i = 0; i < N; i++)
 {
     for (j = i+1; j < N; j++)
     {
             sequence of statements;
      }
 }
```

Value of i	Number of iterations
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

- So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1.
- We've seen that formula before: the total is $O(N^2)$.

Complexity Examples

```
int FindMaxElement(int array[])
{
    int max = array[0];
    for (int i=0; i<n; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
        }
    }
    return max;
}
```

- Runs in O(n) where n is the size of the array
- The number of elementary steps is ~ n

Complexity Examples (2)

```
long FindInversions(int array[])
{
    long inversions = 0;
    for (int i=0; i<n; i++)
        for (int j = i+1; j<n; i++)
            if (array[i] > array[j])
                inversions++;
    return inversions;
}
```

- Runs in $O(n^2)$ where n is the size of the array
- The number of elementary steps is $\sim$ $n*(n+1) / 2$

Complexity Examples (3)

```
decimal Sum3(int n)
{
    decimal sum = 0;
    for (int a=0; a<n; a++)
        for (int b=0; b<n; b++)
            for (int c=0; c<n; c++)
                sum += a*b*c;
    return sum;
}
```

- Runs in cubic time $O(n^3)$
- The number of elementary steps is $\sim n^3$

Complexity Examples (4)

```
long SumMN(int n, int m)
{
    long sum = 0;
    for (int x=0; x<n; x++)
        for (int y=0; y<m; y++)
            sum += x*y;
    return sum;
}
```

- Runs in quadratic time `O(n*m)`
- The number of elementary steps is ~ `n*m`

Complexity Examples (5)

```
long SumMN(int n, int m)
{
    long sum = 0;
    for (int x=0; x<n; x++)
        for (int y=0; y<m; y++)
            if (x==y)
                for (int i=0; i<n; i++)
                    sum += i*x*y;
    return sum;
}
```

- Runs in quadratic time $O(n*m)$
- The number of elementary steps is

$$\sim n*m + \min(m,n)*n$$

Big - Ω notation

- It refers to lower bound of functions.

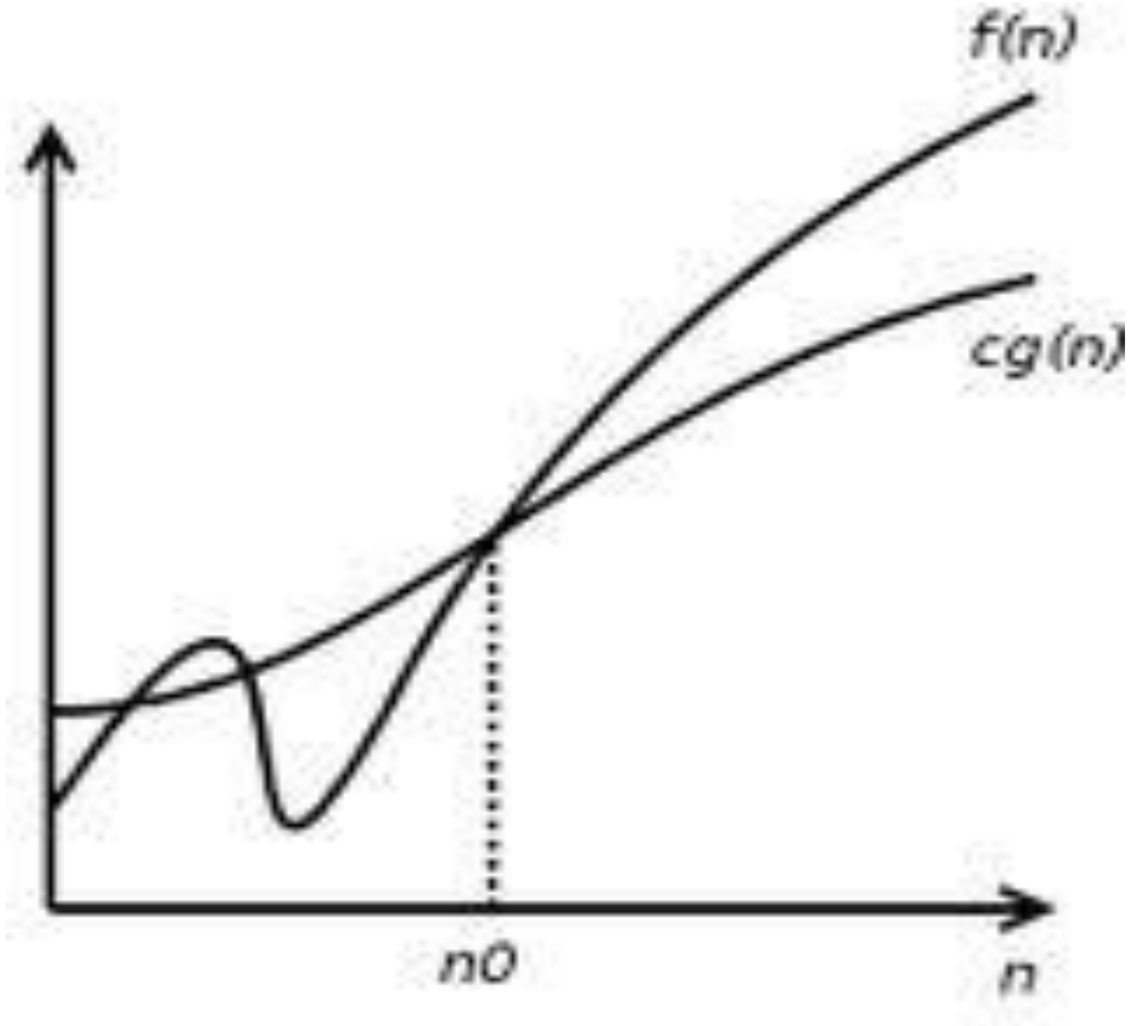
Definition 2: The function $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n)$ for all $n \geq N$.

This definition reads: f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all ns. In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g.

- **Example :**
  - 5n^2 is **Ω(n)** because 5n^2 ≥ 5n for n ≥ 1.

lim n->∞   f(n) / g(n) = c   //`c closer to ∞`

Graph for Omega Notation



Omega Notation (Ω)

>> $\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n0 such that $0 < c * g(n) < f(n)$ for all $n > n0 \}$

>> Asymptotic lower bound.

>> The function $f(n) = \Omega(g(n))$ iff there exist positive constants c and n0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n0$.

>> The statement $f(n) = \Omega(g(n))$ states only that g(n) is only a lower bound on the value of f(n) for all n, $n \geq n0$.

>> E.g.

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

$3n + 2 \geq 3n$ for all $n \geq 1$.

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

$3n + 3 \geq 3n$ for all $n \geq 1$.

3. $100n + 6 = \Omega(n)$

$100n + 6 \geq 100n$ for all $n \geq 0$.

Θ notation

- It refers to tight bound of functions.

Definition 3: $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$.

- Informally, if f(n) is Θ(g(n)) then both the functions have the same rate of increase.

- **Example**:

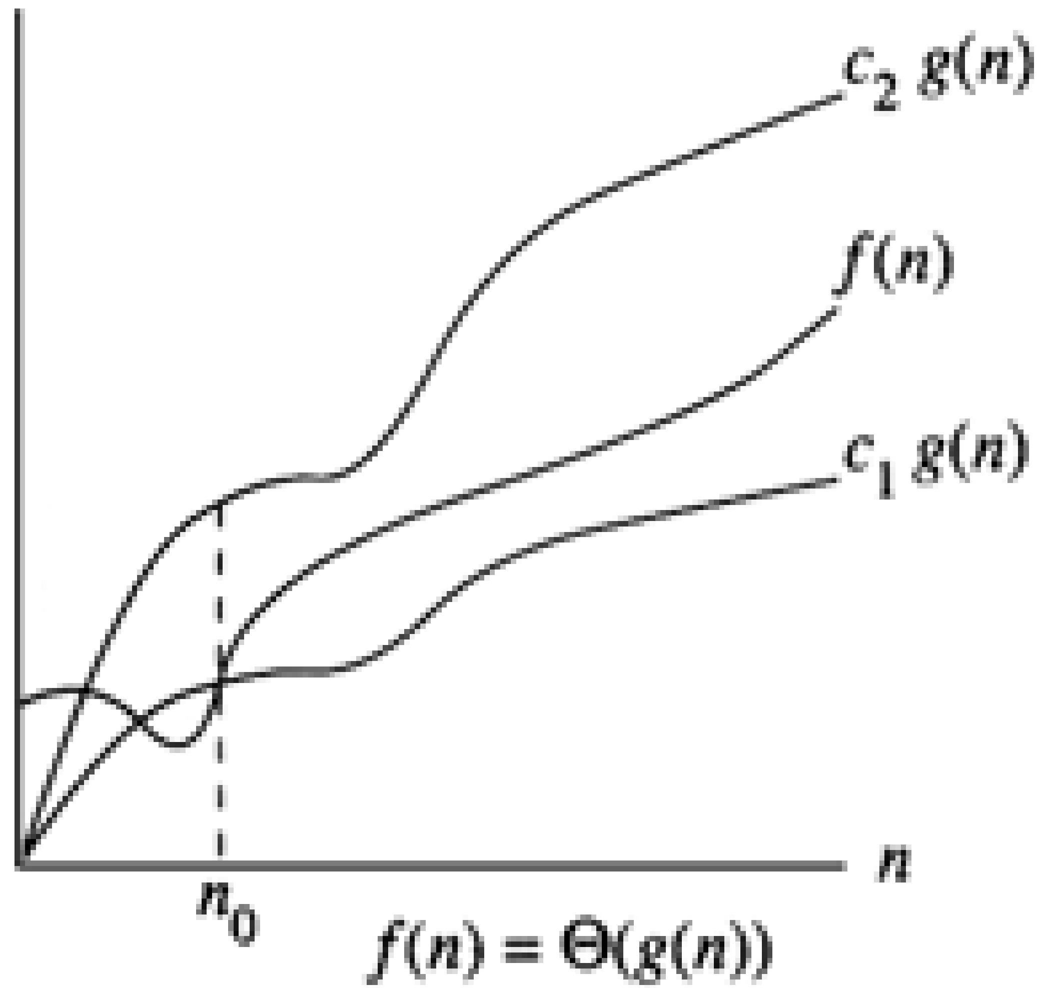
  The same rate of increase for

  $f(n) = n + 5n^{0.5}$ and $g(n) = n$

  because

  $n \leq (n + 5n^{0.5}) \leq 6n$ for $n > 1$

Theta notation



Theta Notation (Θ)

>> $\Theta(g(n)) = \{ f(n)$: there exist positive constants c1 and c2 and n0 such that $0 < c1 * g(n) < f(n) < c2 * g(n)$ for all $n > n0 \}$

>> The function $f(n) = \Theta(g(n))$ iff there exist positive constants C1, C2 and n0 such that $C1 * g(n) \leq f(n) \leq C2 * g(n)$ for all n, $n \geq n0$.

>> E.g.

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

$3n + 2 \geq 3n$ for all $n \geq 2$.

$3n + 2 \leq 4n$ for all $n \geq 2$.

So, C1 = 3, C2 =4 & n0 =2.

>> The statement $f(n) = \Theta(g(n))$ iff g(n) is both an upper and lower bound on the value of f(n).