Analysis of Algorithms

http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/

Book: http://ldc.usb.ve/~xiomara/ci2525/ALG_3rd.pdf

Background

Asymptotic Analysis

Worst, Average, and Best Cases of Algorithms

Asymptotic Analysis

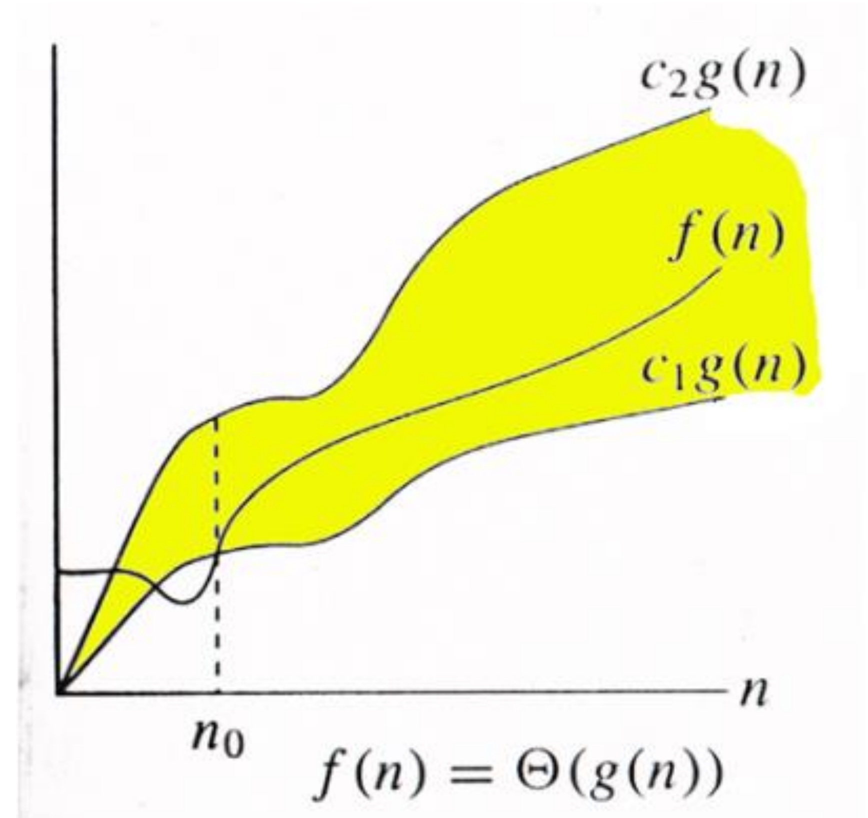
The main idea is to have a measure of efficiency of algorithms that

- 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 complexity of algorithms for asymptotic analysis

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1) Θ Notation

- The theta notation bounds a functions from above and below,
- so it defines exact asymptotic behavior
- A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants
- For example, consider the following expression



$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved

For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions

$\Theta((g(n))$ = { $f(n)$: there exist positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0 \}$$

The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 * g(n)$ and $c_2 * g(n)$ for large values of n ($n \geq n_0$)

The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0

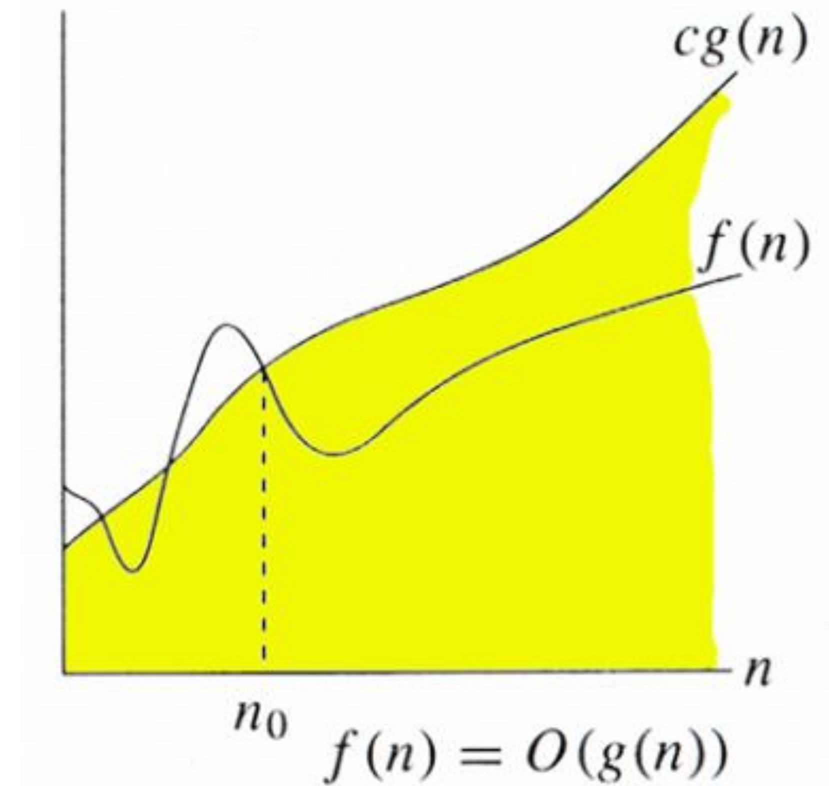
2) Big O Notation

- Big O notation defines an upper bound of an algorithm, it bounds a function only from above

- For example, in the case of Insertion Sort, it takes linear time in best case and quadratic time in worst case

- ∴ time complexity of Insertion Sort is

$$O(n^2)$$



- Note that $O(n^2)$ also covers linear time

- If **Θ** notation is used to represent time complexity of Insertion Sort, two statements should be used for best and worst cases:

  1. The worst case time complexity of Insertion Sort is $\Theta(n^2)$

  2. The best case time complexity of Insertion Sort is $\Theta(n)$

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm

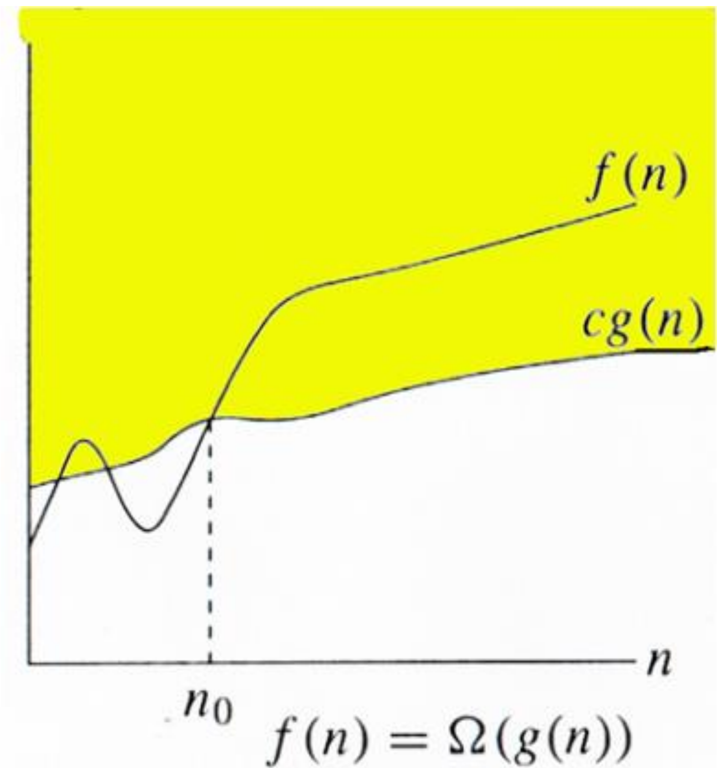
$O(g(n))$ = { $f(n)$: there exist positive constants c and $n0$ such that

$$0 <= f(n) <= cg(n) \text{ for all } n >= n0\}$$

3) Ω Notation

- Just as Big O notation provides an asymptotic upper bound on a function, **Ω** notation provides an asymptotic lower bound

- Notation can be useful when we have lower bound on time complexity of an algorithm



For a given function g(n), we denote by Ω(g(n)) the set of functions

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that

$$0 <= cg(n) <= f(n) \text{ for all } n >= n0 \}.$$

Let us consider the same Insertion Sort example here

The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in the worst case and sometimes in the average case

A good example is the Analysis of Bubble Sort Algorithm

Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$ $\therefore$ $3n + 8 = O(n)$ with $c = 4$ and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$ $\therefore$ $n^2 + 1 = O(n^2)$ with $c = 2$ and $n0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$ $\therefore$ $n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n > 1$ $\therefore$ $2n^3 - 2n^2 = O(n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \le n$, for all $n \ge 1$ $\therefore$ $n = O(n)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \le 410$, for all $n > 1$ $\therefore$ $410 = O(1)$ with $c = 1$ and $n_0 = 1$

Guidelines to Asymptotic Analysis

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

1) **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=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total time = a constant c × n = c n = O(n).

2) **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=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements**: Add the time complexities of each statement.

```
x = x +1; //constant time
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

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

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

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity**: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i=1; i<=n;)
    i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially i = 1, in 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 k^{th} step $2^k = n$, and at $(k + 1)^{th}$ step we come out of the loop. Taking logarithm on both sides, gives

$$log(2^k) = logn$$
$$klog2 = logn$$
$$k = logn \qquad \text{//if we assume base-2}$$

Total time = O(logn).