# Algorithms — 30-03-2015

How to evaluate an algorithm

- Does it do what we want it to do?
- Does it work correctly according to the original specs of the task?
- Is there documentation that describes how to use it and how it works?
- Are procedures created in such a way that they perform logical sub-functions
- Is the code readable?
- Space complexity
- Time complexity

Evaluation can be loosely divided into two major phases:

- a priori estimates — performance analysis
- a posteriori testing — performance measurement

```
Algorithm abc(a,b,c) {
    return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

Time for the return statement is taken as one time step irrespective of the number of +, -, * & / inside it.

(1)

```
Algorithm Sum(a,n) {
    s = 0.0;
    for i = 1 to n
        s = s + a[i];
    return s;
}
```

Let us add a count to determine the number of steps, initialized to zero globally in algorithm 1.

A step can be 10 additions; it can be 20 multiplications; it can be 5 additions & 8 multiplications but not n additions or p+2 subtractions etc.

```
Algorithm Sum(a, n) {
    s = 0.0;
    count = 1 + count; // for the statement above
  for i = 1 to n {
        count = 1 + count; // for for
        s = s + a[i];
        count = 1 + count; // for statement above
  }
  count = 1 + count; // for last time of for
  count = 1 + count; // for return below
  return s;
}
```

(2)

Keeping the count only,

```
Algorithm Sum(a, n) {
    count = count + 1;
    for i = 1 to n count = count + 2
    count = count + 2;
}
```

(3)

So, initialized to zero, count will increase a total of $2n+3$. So, each invocation of algorithm 2 executes $2n+3$ steps.

```
Algorithm RSum(a, n) {
    if n <= 0 then return 0.0;
    else return RSum(a, n-1) + a[n];
}
```

(4)

Adding count to algorithm 4, we get

(5)

```
Algorithm RSum(a, n) {
    count = count + 1; // for if below
    if n <= 0 then {
        count = count + 1; // for return
        return 0.0;
    }
    else {
        count = count + 1; // for addition, function invocation
        return RSum(a, n-1) + a[n];
    }
}
```

Let $t_{RSum}(n)$ be the increase in the value of count when algorithm 5 terminates. ~~We see $t_{RSum}(0) = 2$.~~

We see $t_{RSum}(0) = 2$. When $n > 0$, count increases by 2 plus whatever increase results from the invocation of RSum from within the else clause. By the defn. of $t_{RSum}$, this additional increase is $t_{RSum}(n-1)$. So,

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

So,

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2 * 2 + t_{RSum}(n-2) \\ &\vdots \\ &= n * 2 + t_{RSum}(0) \\ &= 2n + 2, \quad n \geq 0. \end{aligned}$$

Compare algorithm 2 & 4; 2 goes thro. $2n+3$ steps and 4 goes through $2n+2$ steps. Both are linear time algorithms. Both have input size of $n+1$, 1 for array size. Input size is also called instance characteristics.

(6)

```
Algorithm Add (a,b,c, m, n) {
    for i = 1 to m
        for j = 1 to n
            c[i,j] = a[i,j] + b[i,j];
}
```

After introducing count statements,

```
Algorithm Add (a,b,c, m,n) {
    count = count + 1;
    for j = 1 to m {
        count = count + 1;
```

(7)

```
Algorithm Add(a,b,c,m,n) {
    for i = 1 to m {
        count = count + 1;
        for j = 1 to n {
            count = count + 1;
            c[i,j] = a[i,j] + b[i,j];
            count = count + 1;
        }
        count = count + 1;
    }
    count = count + 1;
}
```

Simplifying

```
Algorithm Add (a,b,c, m, n) {
    for i = 1 to m {
        count = count + 2; // 2m
        for j = 1 to n
            count = count + 2 // 2mn
    }
    count = count + 1; // once
}
```

So, increase in count is $2mn+2m+1$.
We observe, if $m>n$, then interchanging the two for loops will reduce step count to $2mn+2n+1$.
Input size of the algorithm is $2mn+2$ and input characteristics are $m$, $n$.

In many algorithms, step count from input characteristics is not so direct, for example, linear search. The step count depends on where the data is. In such cases, we define three types of step count — best case, worst case and average case.

Here also determining the exact step count is exceedingly difficult ~~task~~ task. Also, it may not be very useful for comparative purposes, unless of course, something like $3n+3$ versus $100n+20$. Even then it need not be exactly $100n$ but "about $80n$ or $75n$" etc would be adequate.

So, let us consider step counts as ~~$t_p \geq c_1 n^2$ or $t_Q$~~ $c_1 n^2 <= t_p(n) <= c_2 n^2$ or $t_Q(n,m) = c_1 n + c_2 m$ etc where $c_1, c_2 >= 0$.

In such cases we knows algorithm with complexity $c_1 n^2 + c_2 n$ will be slower than algorithm with complexity $c_3 n$ for sufficiently large values of $n$. For small values of $n$, either can be ~~faster~~ faster, since if $c_1 = 1$, $c_2 = 2$, $c_3 = 100$ then $c_1 n^2 + c_2 n <= c_3 n$ for $n <= 98$ and $c_1 n^2 + c_2 n > c_3 n$ for $n > 98$.

Again, if $c_1 = 1, c_2 = 2, c_3 = 1000$ then $c_1 n^2 + c_2 n <= c_3 n$ for $n <= 998$.

But, irrespective of values of $c_1$, $c_2$ and $c_3$, there will be an $n$ beyond which algorithm ~~of~~ with complexity $c_3 n$ will be faster than the one with complexity $c_1 n^2 + c_2 n$. This value of $n$ is called the break-even point which may even be zero making algorithm with $c_3 n$ always faster or as fast as the other. Finding analytically, of the break-even point is not possible, for that one must run it on a computer. It is sufficient to know that algorithms have complexities like $c_1 n^2 + c_2 n$ or $c_3 n$ for some $c_1, c_2, c_3 > 0$. There is little advantage in determining the $c$'s as effects of $n$ are much more important in the ~~long~~ long run.

With this motivation, we introduce some inexact but useful terminologies for time complexities of algorithms. In the following discussion, functions f and g are non-negative functions.

The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff
~~[illegible]~~
$\exists c$ & $n_0$ such that $f(n) \le c * g(n)$ $\forall n, n \ge n_0$.

Examples: $3n+2 = O(n)$ as $3n+2 \le 4n$ for all $n \ge 2$.

$3n+3 = O(n)$ as $3n+3 \le 4n$ for all $n \ge 3$.

$100n+6 = O(n)$ as $100n+6 \le 101n$ for all $n \ge 6$.

$10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 \le 11n^2$ for all $n \ge 5$.

$1000n^2+100n-6 = O(n^2)$ as $1000n^2+100n-6 \le 1001n^2$ for all $n \ge 100$

$6*2^n+n^2 = O(2^n)$ as $6*2^n+n^2 \le 7*2^n$ for all $n \ge 4$.

$3n+3 = O(n^2)$ as $3n+3 \le 3n^2$ for all $n \ge 2$

$10n^2+4n+2 = O(n^4)$ as $10n^2+4n+2 \le 10n^4$ for all $n \ge 2$.

$3n+2 \ne O(1)$ as $3n+2$ is not less than or equal to $c$ for any $c$ and all $n \ge n_0$

Similarly $10n^2+4n+2 \ne O(n)$ as $10n^2+4n+2$ is not less than or equal to $cn$ for any $c$ and all $n \ge n_0$

As seen through the examples, $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound of $f(n)$ for all $n, n \ge n_0$. But it does not say anything about how good this bound is. See that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^{n+1})$ and so on. To derive any information from $f(n) = O(g(n))$, $g(n)$ should be as small as possible a fn of n for which $f(n) = O(g(n))$. So, we would say $3n+3 = O(n)$, but ~~[illegible]~~ would not say $3n+3 = O(n^2)$, though that is also correct.

~~The relation is not reversible, i.e. $f(n) = O(g(n)) \not\Rightarrow g(n) = O(f(n))$.~~

We don't write $O(g(n)) = f(n)$, because = here is not "equals" rather it is to be read as "is".

The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist +ve constants $c$ and $n_0$ such that $f(n) \geq c * g(n)$ for all $n, n \geq n_0$.

Examples:

$3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$

$3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$

$100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$

$10n^2+4n+2 = \Omega(n^2)$ as $10n^2+4n+2 \geq n^2$ for $n \geq 1$.

$6+2^n+n^2 = \Omega(2^n)$ as $6+2^n+n^2 \geq 2^n$ for $n \geq 1$.

But then, like before,

$3n+3 = \Omega(1)$, $10n^2+4n+2 = \Omega(n)$, $10n^2+4n+2 = \Omega(1)$,

$6+2^n+n^2 = \Omega(n^{100})$, $6+2^n+n^2 = \Omega(1)$.

As in the case of O notation there are several function $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. To be useful, $f(n) = \Omega(g(n))$ should be such that $g(n)$ is as large as possible.

The function $f(n) = \theta(g(n))$ (read as "f of n is theta of g of n") iff there exist +ve constants $c_1, c_2$ and $n_0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$.

Examples:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$ and $3n+2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$.

$3n+3 = \theta(n)$

$10n^2+4n+2 = \theta(n^2)$

$6+2^n+n^2 = \theta(2^n)$

$10 * \log n + 4 = \theta(\log n)$

$3n+2 \neq \theta(1)$, $3n+3 \neq \theta(n^2)$, $10n^2+4n+2 \neq \theta(n)$.

The theta notation is more precise than both the $O$ and $\Omega$ notations.
The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound of $f(n)$.

Also notice that in all the examples for $O$, $\Omega$ and $\Theta$, coefficient of $g(n)$ is always 1, i.e. we never write $3n+3 = O(3n)$ or $10 = O(100)$ or $10n^2+4n+2 = \Omega(4n^2)$ although each of these statements is true. It is only practice to have the coefficient as 1.

The $O$, $\Omega$ and $\Theta$ are called asymptotic complexity and they do not require step count in an algorithm.

Exercise:

Show that the following equalities are correct:

$$5n^2 - 6n = \Theta(n^2)$$
$$n! = O(n^n)$$
$$2n^2 2^n + n\log n = \Theta(n^2 2^n)$$
$$\sum_{i=0}^{n} i^2 = \Theta(n^3)$$

Show that the following equalities are incorrect:

$$10n^2 + 9 = O(n)$$
$$n^2 \log n = \Theta(n^2)$$
$$n^2/\log n = \Theta(n^2)$$