Lecture 10: Big-Oh

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

January 27, 2014

So far we have talked about $O()$ informally, as a way of capturing the worst-case computation time of an algorithm. We are now going to delve more in detail into the definition of $O()$, as well as how we compute it for different pieces of code. Note that $O()$ is used in general to measure both time as quell as memory requirements.

1 Examples

Whenever we assess running time, we will do it as a function of the **size of the input** provided to the algorithm. We are mainly interested in the "shape" of the running time, as a function of the

We have noted before that basic operations, such as assignments, arithmetic operations, comparisons etc are assumed to take a constant amount of time, because this time does not vary regardless of the input; we denote this as $O(1)$. For example, the code:

int $x = 11$;
int $y = 10$;
int $z = x + y$;
if $(z > 10)$ $z = z - 1$;

is $O(1)$. No matter how we modify the values, these operations will always take the same amount of time. It is true that in practice, the 4 instructions above will take slightly longer than only one such instruction. But the $O()$ notation will ignore such details, and focus on the "big picture": while the running time of the 4 instructions is the sum of the running times for each of them, it still is constant, regardless of the actually values we give to these variables.

Now let us consider a for loop going on an array a of size n:

for $(i = 0; i < n; i++)$
  $a[i] = 1$;

The loop executes n times, and during each execution, we run a constant-time operation. Hence, the total running time is $n * c$ where c is a constant, and it *grows linearly* with the size of the array a. Hence, the complexity is $O(n)$.

Now consider another loop:
for ($i = 0$; $i < n$; $i+ = 2$)
$a[i] = 1$;

This loop executes $n/2$ times, so the total running time is $n/2 * c$, where c is a constant, but from the point of view of $O()$, this is still $O(n)$, because the running time still grows linearly with n.

2 Intuition and definition

We have seen now that $O()$ only takes into account the "fastest-growing" components of the actual running time, and that constants do not matter. Now we give a formal definition of $O()$ and some results which help us compute it.

We want to use the $O()$ notation in order to say whether a function f grows slower than a function g. If so, we want to say that $f(n)$ is "of order" $g(n)$ (denoted $O(g(n))$). So far, we have been using this mainly to talk about computation time, but we will also use it for memory usage in the future. In general, we can define this concept by talking about general functions, even without having a computational application in mind.

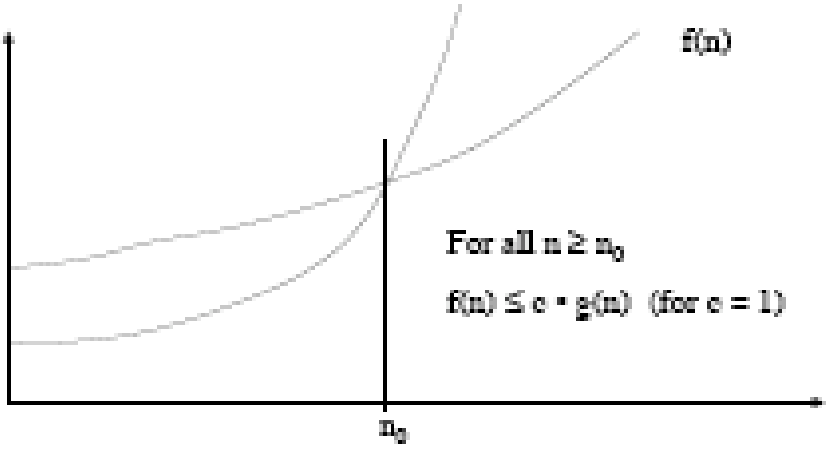

Figure 1: Function $f(n)$ grows slower than $g(n)$

To get some intuition, consider the example in Figure 1. Function f is growing slower than g (the top line), but this does not mean that f is always smaller. However, after some point n_0, it really is always smaller. In general, we can allow g to be multiplied by a positive constant c: the effect of this will not affect the order of growth. For example, if f is a quadratic function and g is a linear function, intuitively no matter how much we multiply g, f will overtake it eventually. This motivates the following definition:
Definition (Big-Oh): Function $f(n)$ is $O(g(n))$ if and only if there exist a constant $c > 0$ and a constant natural number n_0 such that:

$$\forall n \geq n_0, f(n) \leq cg(n)$$

Note that c and n_0 must be constants (i.e., they do not depend on n).

We can indeed think of $O(g(n))$ as the *set* of all functions $f(n)$ that are $O(g(n))$:

$$O(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0, f(n) \leq cg(n)\}$$

Hence, we can use the notation $f(n) \in O(g(n))$ to denote the fact the $f(n)$ is $O(g(n))$. As we will see below, these sets are infinite.

3 Proving $O()$ relationships between functions

In order to prove that $f \in O(g)$, we can use directly the definition. More precisely, we have to find constants n_0 and c such that $f(n) \leq cg(n), \forall n \geq n_0$.

Example 1: Prove that $5 + 3n^2 \in O(1 + n^2)$.
Proof: The intuition for this example is given in Figure 2. We need to pick some $c \geq 3$. For example, let us pick $c = 5$. Also, let us choose $n_0 = 1$. Then we have that for any $n \geq n_0$:

$$\begin{aligned} 5 + 3n^2 &\leq 5 + 5n^2 \text{ because } n \geq 1 \\ &= 5(1 + n^2) = c(1 + n^2) \end{aligned}$$

Since we found n_0 and c, this concludes the proof.

Example 2: Prove that $n \sin(n) \in O(n)$.
Proof: Note that $\sin(n) \leq 1, \forall n$, so $n \sin(n) \leq n, \forall n$. Hence, we can pick $c = 1$ and $n_0 = 1$ and the definition applies.

Example 3: Prove that $5n + 13 \in O(n)$.
Proof: Pick $c = 6$ and $n_0 = 13$, and the definition applies.

In order to prove that $f \notin O(g)$, we need to show that, for any c and n_0 we choose, there will be some value $n \geq n_0$ so that $f(n) > cg(n)$.

Example 4: Prove that $n^2 \notin O(n)$.
Proof: Let us pick an arbitrary value c and an arbitrary n_0. Consider the function $n^2 - cn = n(n - c)$. This is a quadratic function, which will be positive for any n outside the $[0, c]$ interval. Hence, we can pick any $n > \max(n_0, c)$, and we will have $n^2 - cn > 0$, i.e. $f(n) > cg(n)$. This concludes the proof.

4 Useful results

Theorem (Sum rule): Let f_1 and f_2 be two functions such that $f_1(n) \in O(g(n))$ and $f_2(n) \in O(g(n))$. Then $f_1(n) + f_2(n) \in O(g(n))$

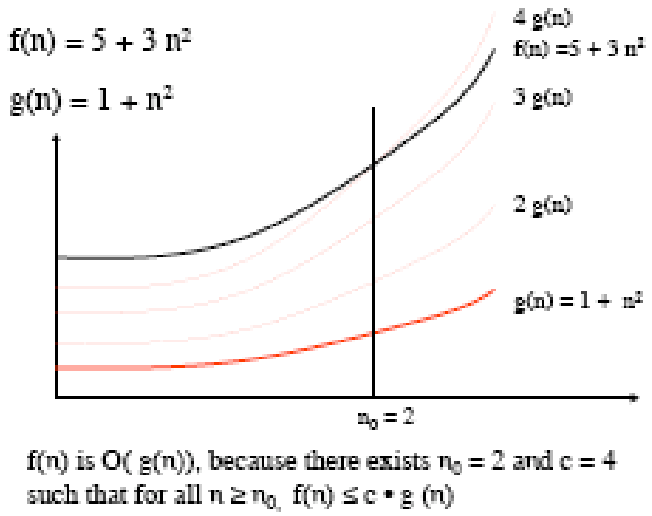

Figure 2: Visualizing the first example

Proof: From the definition we have:

$$f_1(n) \in O(g(n)) \implies \exists n_1, c_1 \text{ such that } f_1(n) \leq c_1 g(n), \forall n \geq n_1$$

$$f_2(n) \in O(g(n)) \implies \exists n_2, c_2 \text{ such that } f_2(n) \leq c_2 g(n), \forall n \geq n_2$$

Let $n_0 = \max(n_1, n_2)$. So for any $n \geq n_0$, both of the inequalities above hold. By adding them up, we have:

$$f_1(n) + f_2(n) \leq (c_1 + c_2) g(n) \forall n \geq n_0$$

which means that $f_1(n) + f_2(n) \in O(g(n))$.

In code, this situation occurs when we have a *sequence* of operations, e.g. two for loops following each other, a for loop followed by a function call, etc. This result basically means that **the time complexity of a sequence of operations will be dominated by the longest operation in the sequence**.

Theorem (Constant factors rule): If $f(n) \in O(g(n)$ then $kf(n) \in O(g(n))$ for any *positive constant* k.
Proof: From the definition, we have:

$$\forall n \geq n_0, f(n) \leq cg(n) \implies kf(n) \leq kcg(n)$$

Choosing the same n_0 and constant kc, the definition holds.

In terms of running time, this means that repeating a piece of code for a constant number of times does not change its complexity; so loops that execute a constant number of times do not increase the worst-case complexity.

Theorem (Product rule): If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$
Proof: From the definition we have:

$$f_1(n) \in O(g_1(n)) \Longrightarrow \exists n_1, c_1 \text{ such that } f_1(n) \leq c_1 g_1(n), \forall n \geq n_1$$

$$f_2(n) \in O(g_2(n)) \Longrightarrow \exists n_2, c_2 \text{ such that } f_2(n) \leq c_2 g_2(n), \forall n \geq n_2$$

Let $n_0 = \max(n_1, n_2)$. So for any $n \geq n_0$, both of the inequalities above hold. By multiplying them, we have:

$$f_1(n) \cdot f_2(n) \leq (c_1 c_2) g_1(n) \cdot g_2(n), \forall n \geq n_0$$

which means that $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.

In code, this situation corresponds to **nesting**, e.g. nested for loops, or a function call inside a loop, or recursive functions. This result means that nesting will increase the complexity of the code.