O-Notation.
Lecture Notes.
Kasper Green Larsen.

1 Introduction to O-Notation

This lecture note is intended as a quick introduction to O-notation, which is commonly used when analyzing algorithms. It should be familiar to computer scientists, but may be new to students from other study directions. Regardless of your background, the note may serve as a quick recap of the notation.

When analyzing algorithms or data structures, we are typically interested in understanding how their resource consumptions grow as a function of the input size. Thus we usually have a variable n denoting the input size, where n is assumed to be positive. As an example, consider an algorithmic problem where we receive n integers $x_1, \dots, x_n$ as input and must determine whether there is a triple x_i, x_j, x_k with $i < j < k$ such that $x_i + x_j + x_k = 0$. A simple algorithm for solving this problem just checks all candidate triples (i, j, k). Pseudo-code for this algorithm would be something like:

```
For $i = 1, \dots, n$:
    For $j = i + 1, \dots, n$:
        For $k = j + 1, \dots, n$:
            If $x_i + x_j + x_k = 0$: Return **Yes**.
Return **No**.
```

We would like to state the running time of this algorithm. Clearly, it is proportional to the number of triples tested. There are precisely:

$$\binom{n}{3} = \frac{n!}{3!(n-3)!} = \frac{n(n-1)(n-2)}{6} = n^3/6 - n^2/6 - n^2/3 + n/3.$$

such triples. So the running time is $n^3/6 - n^2/2 + n/3$. This expression is unnecessarily complicated for understanding the behaviour of the algorithms running time as a function of n. What we actually care about is just the rate of growth, i.e. we don't care about the constant factors $6, 2$ and 3. Moreover, as n gets large enough, the term $n^3/6$ will dominate $n^2/2$ and $n/3$ as it grows strictly faster as a function of n.

O-Notation. To simplify such expressions and only communicate the rate of growth, we introduce O-Notation. Formally, for two functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, we may write

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

if the following is satisfied:

- There exists constants $n_0 \in \mathbb{R}$ and $c > 0$, such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Said in words, we may write that $f(n) = O(g(n))$ if from some point and onwards ($n \geq n_0$), the function f remains bounded by a constant c times $g(n)$.

Let us use this definition to simplify the statement of the running time $f(n) = n^3/6 - n^2/2 + n/3$ of the above algorithm. First, we notice that

$$n^3/6 = O(n^3).$$

This can be seen by observing that $n^3/6 \leq 1 \cdot n^3$ for all $n \geq 0$. That is, we have chosen $c = 1$ and $n_0 = 0$ in the definition of O-notation. By a similar argument, we could argue that $n/3 = O(n)$. We could even combine $n^3/6 - n^2/2$ and see that:

$$n^3/6 - n^2/2 = O(n^3).$$

This follows by observing that $n^3/6 - n^2/2 \leq n^3/6 \leq 1 \cdot n^3$ for all $n \geq 0$. Indeed, any negative term in a sum can always be dropped when introducing O-notation.

Sums. O-notation is convenient when a function f is a sum of multiple terms. Indeed, for any constant number of terms:

$$f(n) = f_1(n) + f_2(n) + \cdots + f_k(n),$$

if we have that $f_i(n) = O(g(n))$ for all i, then $f(n) = O(g(n))$. To see this, notice that by definition, we have $f_i(n) \leq c_i g(n)$ for $n \geq n_i$ where c_i and n_i are constants. Thus if we define $n_0 = \max\{n_1, \ldots, n_k\}$ and $c = k \max\{c_1, \ldots, c_k\}$ then for any $n \geq n_0$, we also have $n \geq n_i$ and therefore:

$$\begin{aligned}
f(n) &= f_1(n) + f_2(n) + \cdots + f_k(n) \\
&\leq c_1 g(n) + \cdots + c_t g(n) \\
&\leq \max_i c_i g(n) + \cdots + \max_i c_i g(n) \\
&= k \max_i c_i g(n) \\
&= ckg(n).
\end{aligned}$$

By definition of O-notation, we thus have $f(n) = O(g(n))$.

Sums of term with varying magnitude. If we return to our example with the function $f(n) = n^3/6 - n^2/2 + n/3$, we see that all of the terms are $O(n^3)$. Indeed $n/3 \leq 1 \cdot n^3$ for all $n \geq 1$, thus we arrive at:

$$f(n) = O(n^3),$$

which is a much simpler expression and more clearly illustrates that the running time of the algorithm grows as n^3 as a function of the input size. When one is comfortable with O-notation, one realized that in any sum of terms, only the quickest growing matter. For instance, in the expression:

$$f(n) = n/100 + 100\sqrt{n}$$

We actually have that $100\sqrt{n}$ is bigger than $n/100$ for quite a while. However, as n grows big enough (in this case for $n/100 > 100\sqrt{n} \Leftrightarrow n > 10^8$, the term $n/100$ dominates. Thus $f(n) = O(n/100)$, which is again $O(n)$. We could also start out by observing that $100\sqrt{n} \leq 100n$ for all $n \geq 1$, i.e. $100\sqrt{n} = O(n)$ by choosing the constant $c = 100$ and $n_0 = 1$ in the definition of O-notation.

Constants don't matter. In our example $f(n) = n^3/6 - n^2/2 + n/3$, all constant factors are less than 1. In general, we might have much bigger constant like $f(n) = 100n^2$. Also in this case, the definition of O-notation allows us to drop the 100 since $100n^2 \geq 100 \cdot n^2$ for $n \geq 1$. That is, we may use $c = 100$ in the definition of O-notation. If we have several terms that grows at the same rate, i.e. $f(n) = n/10 + 5n$, then using the rules for sums, we can again conclude $f(n) = O(n)$, i.e. only the rate of growth matters, not the constants and not the number of such terms (as long as there is a constant number of terms in the expression).

Products. O-notation is also convenient when understanding products of functions. If $f(n)$ and $g(n)$ are both non-negative functions for $n \geq n_0$ and $f(n) = O(h(n))$ and $g(n) = O(\ell(n))$ then

$$f(n)g(n) = O(h(n)\ell(n)).$$

For example, for $f(n) = 10\sqrt{n}$ and $g(n) = 2n$, we have $f(n) = O(\sqrt{n})$ and $g(n) = O(n)$ and therefore $f(n)g(n) = O(n^{3/2})$.

To see that this is a valid rule, let c_1 and n_1 be the constants such that $f(n) \leq c_1 h(n)$ for $n \geq n_1$ and let c_2 and n_2 be the constants such that $g(n) \leq c_2 \ell(n)$ for $n \geq n_2$. Let $c = c_1 c_2$ and $n_3 \max\{n_0, n_1, n_2\}$, then for $n \geq n_3$, we have

$$f(n)g(n) \leq c_1 h(n) c_2 \ell(n) = c_1 c_2 h(n) \ell(n).$$

Base of Logs. One observation that may come in handy is that the base of logarithms - as long as they are constants greater than 1 - do not matter in O-notation. That is, $\log_2 n, \ln n$ and $\log_{10} n$ are all equivalent. To see this, let $a, b > 1$ be any constants and observe that:

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \cdot \log_b n.$$

The term $\log_b a$ is always positive when $a, b > 1$ and is clearly a constant when both a and b are constants. Thus we have that $\log_a n = O(\log_b n)$ by using the constant $1/\log_b a$ in the definition of O-notation. We will thus often drop the base of the logarithm when using O-notation and simply write

$$f(n) = O(\log n).$$

Careful with Exponents. One place to be careful when using O-notation is when constant appear in exponents. For instance, consider the expression $2^{2\log_2 n}$. If we had only had $2\log_2 n$, it would be true that $2\log_2 n = O(\log_2 n)$, but it is not true that $2^{2\log_2 n} = O(2^{\log_2 n}) = n$. Indeed $2^{2\log_2 n} = n^2$. Thus be aware that we can only ignore multiplicative constant factors, not constant factors in exponents.

Tightness of Bounds. Note that the definition of O-notation does not require that when we write $f(n) = O(g(n))$, we have that f grows at the rate of g. Indeed, it is perfectly valid to write that $n = O(n^4)$. However, when stating resource consumptions of algorithms, we generally wish to give a tight an upper bound in the O-notation as possible.

Examples. Let us conclude with a few examples of O-notation. If you wish, you may see it as an exercise to prove formally that these are valid statements:

- $4n^2 = O(n^2)$.
- $n/100 + \log_2 n = O(n)$.
- $10n \log_3 n = O(n \log n)$.
- $2^{2\log_2 n+1} = O(n^2)$.
- $n^2(n/10 + 2n^3) = O(n^5)$.