TIME COMPLEXITY OF ALGORITHMS

Vassos Hadzilacos
University of Toronto

1 Measuring time complexity

The ***worst-case*** time complexity of an algorithm is expressed as a function

$$T : \mathbb{N} \to \mathbb{N}$$

where $T(n)$ is the maximum number of "steps" in any execution of the algorithm on inputs of "size" n. Intuitively, the amount of time an algorithm takes depends on how large is the input on which the algorithm must operate: Sorting large lists takes longer than sorting short lists; multiplying huge matrices takes longer than multiplying small ones. The dependence of the time needed on the size of the input is not necessarily linear: sorting twice the number of elements takes quite a bit more than just twice as much time; searching (using binary search) through a sorted list twice as long, takes a lot less than twice as much time. The time complexity function expresses that dependence. Note that an algorithm might take different amounts of time on inputs of the same size. We have defined the worst-case time complexity, which means that we count the *maximum* number of steps that any input of a particular size could take. For example, if the time complexity of an algorithm is $3 \cdot n^2$, it means that on inputs of size n the algorithm requires *up to* $3 \cdot n^2$ steps. To make this precise, we must clarify what we mean by "input size" and "step".

Input size. We can define the size of an input in a general way as the number of bits required to store the input. This definition is general but it is sometimes inconvenient because it is too low-level. More usefully we define the size of the input in a way that is problem-dependent. For example, when we are dealing with sorting algorithms, it may be more convenient to use the number of elements we want to sort as the measure of the input size. This measure ignores the size of the individual elements that are to be sorted. Sometimes there may be several reasonable choices for the size of input. For instance, if we are dealing with algorithms for multiplying square matrices, we may express the input size as the dimension of the matrix (i.e., the number of columns or rows), or we may express the input size as the number of entries in the matrix. In this case the two measures are related to each other (the latter is the square of the former). One conclusion from this discussion is that in order to properly interpret the function that describes the time complexity of an algorithm we must be clear about how exactly we measure the size of inputs.

Step. A step of the algorithm can be defined precisely if we fix a particular machine on which the algorithm is to be run. For instance, if we are using a machine with a Pentium processor, we might define a step to be one Pentium instruction. This is not the only reasonable choice: different instructions take different amounts of time, so a more refined definition might be that a step is one cycle of the processor's clock. In general, however, we want to analyse the time complexity of an algorithm without restricting ourselves to some particular machine. We can do this by adopting a more flexible notion of what constitutes a step. In general, we will consider a step to be anything that we can reasonably expect a computer to do in a fixed amount of time. Typical examples are performing an arithmetic operation, comparing two numbers, or assigning a value to a variable.

2 Asymptotic bound notation

Since, in the interest of generality, we measure time in somewhat abstractly defined "steps", there is little point in fretting over the precise number of steps. For instance, if by some definition of steps the time complexity of the algorithm is $5n^2$, by a different definition of steps it might be $7n^2$, and by yet another definition of steps it might be $n^2/2$. Thus, we would like to be able to ignore constant factors when expressing the time complexity of algorithms. If we are willing to be flexible about constant factors, we should also be willing to be flexible about "low-order" terms. So, for instance if the time complexity is $5n^2 + 17\log n$, and we are willing to drop the constant factor 5 of n^2, we should also be willing to drop the term $17\log n$ (since the $4n^2$ steps we are ignoring are many more than $17\log n$, for large enough values of n).

To express, in a mathematically meaningful manner, approximations that are oblivious to constant factors and low-order terms, computer scientists have developed some special notation about functions, known as the "big-oh", the "big-omega" and "big-theta" notation. If $k \in \mathbb{N}$, $\mathbb{N}^{\geq k}$ denotes be the set of natural numbers that are greater than or equal to k. $\mathbb{R}^{\geq 0}$ denotes the set of nonnegative real numbers and $\mathbb{R}^{>0}$ denotes the set of positive real numbers.

Definition. Let $f : \mathbb{N}^{\geq k} \to \mathbb{R}^{\geq 0}$, for some $k \in \mathbb{N}$. $O(f)$ is the following set of functions from $\mathbb{N}^{\geq \ell}$ to $\mathbb{R}^{\geq 0}$, for any $\ell \in \mathbb{N}$:

$$O(f) \overset{\text{def}}{=} \{g : \text{there exist } c \in \mathbb{R}^{>0} \text{ and } n_0 \in \mathbb{N} \text{ such that for all } n \geq n_0,\ g(n) \leq c \cdot f(n)\}.$$

In words, $g \in O(f)$ if for all sufficiently large n (for $n \geq n_0$) $g(n)$ is bounded from above by $f(n)$ — possibly multiplied by a positive constant. We say that f is an ***asymptotic upper bound*** for g.

Example 1. $f(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} \in O(n^2)$. This is because $3 \cdot n^2 + 4 \cdot n^{3/2} \leq 3 \cdot n^2 + 4 \cdot n^2 \leq 7 \cdot n^2$. Thus, pick $n_0 = 0$ and $c = 7$. For all $n \geq n_0$, $f(n) \leq c \cdot n^2$.

Example 2. $f(n) = (n-5)^2 \in O(n^2)$. This is because $(n-5)^2 = n^2 - 10 \cdot n + 25$. Check (with elementary algebra) that for all $n \geq 3$, $n^2 - 10 \cdot n + 25 \leq 2 \cdot n^2$. Thus, pick $n_0 = 3$ and $c = 2$. For all $n \geq n_0$, $f(n) \leq c \cdot n^2$.

Example 3. $n^2 - 10n \notin O(n)$. We prove this by contradiction. Assume the contrary, i.e., that $n^2 - 10n \in O(n)$. Thus, there are constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $n^2 - 10n \leq cn$. Therefore, for all $n \geq n_0$, $n \leq c + 10$. Let $k = 1 + \max(n_0, c+10)$. Clearly, $k \geq n_0$ but it is not the case that $k \leq c + 10$, so we have derived a contradiction. This means that our original assumption, namely that $n^2 - 10n \in O(n)$, is wrong. Therefore, $n^2 - 10n \notin O(n)$.

Exercise. Prove the following:
1. $n \in O(n^2)$.
2. $3n + 1 \in O(n)$.
3. $\log_2 n \in O(\log_3 n)$.
4. $O(\log_2 n) \subseteq O(n)$. (Hint: $\log_2 n < n$ for all $n \geq 1$.)
5. $O(n^k) \subseteq O(n^\ell)$ for all constants $0 \leq k < \ell$.

There is a similar notation for asymptotic *lower* bounds, the "big-omega" notation.

Definition. Let $f : \mathbb{N}^{\geq k} \to \mathbb{R}^{\geq 0}$, for some $k \in \mathbb{N}$. $\Omega(f)$ is the following set of functions from $\mathbb{N}^{\geq \ell}$ to $\mathbb{R}^{\geq 0}$, for any $\ell \in \mathbb{N}$:

$$\Omega(f) \overset{\text{def}}{=} \{g : \text{there exist } d \in \mathbb{R}^{>0} \text{ and } m_0 \in \mathbb{N} \text{ such that for all } n \geq m_0,\ g(n) \geq d \cdot f(n)\}.$$

In words, $g \in \Omega(f)$ if for all sufficiently large n (for $n \geq m_0$) $g(n)$ is bounded from below by $f(n)$ — possibly multiplied by a positive constant. We say $f(n)$ is an ***asymptotic lower bound*** for $g(n)$.

Definition. $\Theta(f) \overset{\text{def}}{=} O(f) \cap \Omega(f)$.

Thus, if $g(n) \in \Theta(f)$ then $g(n)$ and $f(n)$ are within a constant factor of each other.

Exercise. Prove the following.

1. $n^2 \in \Omega(n)$.
2. $\Omega(n \log n) \subseteq \Omega(n)$.
3. $\sum_{i=1}^{n} \log_2 i \in \Theta(n \log_2 n)$. (Hint for (iii): For $\lceil n/2 \rceil \leq i \leq n$, $\log_2 i \geq (\log_2 n) - 1$).

The sets $O(f)$, $\Omega(f)$, and $\Theta(f)$ have the following useful properties, which you should prove:

- $g \in O(f)$ if and only if $f \in \Omega(g)$.
- $O(f) = O(g)$ if and only if $f \in O(g)$ and $g \in O(f)$.
- $O(f) = O(g)$ if and only if $f \in \Theta(g)$.
- If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
- If $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$ then $g_1 + g_2 \in O(\max\{f_1, f_2\})$.
- If $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$ then $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.