Asymptotic Analysis of Algorithms

1 "Big-Oh" Notation

A style of time complexity analysis that is very common in algorithm analysis is the so-called *asymptotic analysis.* Except on rare occasions, this is the type of analysis we do in this course. (You will be happy to know it is the easiest!) In asymptotic analysis we drop low order terms and multiplicative constants from the time complexity function $T(n)$. For example, if the function is $3 \cdot n^2 + 7 \cdot \log n$ we are only interested in the "n^2" part: $7 \cdot \log n$ is a low order term and 3 is "just" a multiplicative constant. Some special mathematical notation has been developed to express asymptotic analysis conveniently. To express *asymptotic upper bounds* on running times of algorithms, we introduce "big-oh" notation.

Definition 1 *Let $f : \mathbb{N} \to \mathbb{R}$.*

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

In words, $g \in O(f)$ (pronounced "gee is in big oh of ef") 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 $f(n)$ is an *asymptotic upper bound* for $g(n)$. Figure 1 illustrates graphically what it means for g to be in $O(f)$.

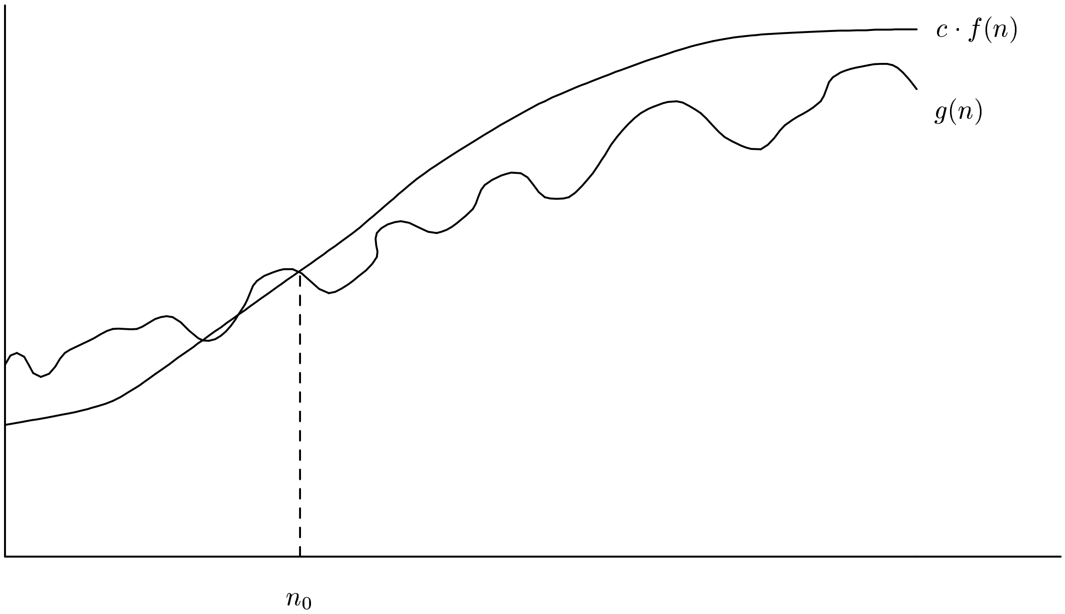

Figure 1. $g(n) \in O(f(n))$.

Example 1. $f(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} \in O(n^2)$. This is because $|f(n)| = 3 \cdot n^2 + 4 \cdot n^{3/2} \leq 3 \cdot n^2 + 4 \cdot n^2 = 7 \cdot |n^2|$ (remember, $n \in \mathbb{N}$). 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 $|f(n)| = (n-5)^2 = n^2 - 10 \cdot n + 25$. 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 $n \geq n_0$, $|f(n)| \leq c \cdot |n^2|$.

Exercise. Prove the following.
(i) $n \in O(n^2)$.
(ii) $3n + 1 \in O(n)$.
(iii) $\log_2 n \in O(\log_3 n)$.
(iv) $O(\log_2 n) \subseteq O(n)$. (Hint: $\log_2 n < n$ for all $n \geq 1$.)

(v) $O(n^k) \subseteq O(n^l)$ for any real constants $0 \leq k < l$.
(vi) $p(n) \in O(n^d)$, where p is any polynomial of degree d.
(vii) $n \notin O(\sqrt{n})$.
(viii) $O(\sqrt{n}) \not\subseteq O(\log n)$.

Why is this useful? In many cases obtaining the exact time complexity function $T(n)$ of an algorithm is a very laborious task and yields a messy function. However, it is often relatively easy to find a much simpler function $f(n)$ that is an asymptotic upper bound for $T(n)$, prove that $T(n) \in O(f(n))$, and leave it at that (without bothering to find an exact expression for $T(n)$).

Of course, we want to find the best asymptotic upper bound on $T(n)$ possible. For example, if $T(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} + 5 \cdot \log n + 16$ then $T(n) \in O(n^2)$. But, $T(n)$ is also in the sets $O(n^3), O(n^4), \ldots, O(n^{100}), \ldots$ To say that $T(n) \in O(n^{100})$ is true but quite uninformative. We want to find a simple asymptotic upper bound that is as small as possible. The bound $f(n)$ is as small as possible for $T(n)$ if $T(n) \in O(f(n))$ and for all functions $g(n)$ such that $T(n) \in O(g(n))$, it is also the case that $f(n) \in O(g(n))$. In words, $f(n)$ is as small an asymptotic upper bound as possible for $T(n)$ if $f(n)$ is an asymptotic upper bound for $T(n)$ and all other functions which are asymptotic upper bounds for $T(n)$ are also asymptotic upper bounds for $f(n)$. For the $T(n)$ given above, $f(n) = n^2$ is as small an asymptotic upper bound as possible, and so is $g(n) = 53 \cdot n^2 + 16n$. We prefer to state $f(n)$ as an asymptotic upper bound on $T(n)$ rather than $g(n)$ because $f(n)$ is simpler. On the other hand, if $h(n) = n^2 \log n$ then $T(n) \in O(h(n))$, but $h(n)$ is not as small an asymptotic upper bound as possible (why?).

2 "Big-Omega" and "Big-Theta" Notation

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

Definition 2 *Let* $f : \mathbb{N} \to \mathbb{R}$.

$$\Omega(f) \stackrel{\text{def}}{=} \{\, g : \mathbb{N} \to \mathbb{R} \,|\, \text{There exist } c > 0 \text{ and } n_0 \geq 0 \text{ such that for all } n \geq n_0,\ |g(n)| \geq c \cdot |f(n)| \,\}.$$

In words, $g \in \Omega(f)$ (pronounced "gee is in big omega of ef") if for all sufficiently large n (i.e., for $n \geq n_0$) $|g(n)|$ is bounded from below by $|f(n)|$ — possibly multiplied by a positive constant. We say $f(n)$ is an *asymptotic lower bound* for $g(n)$. Analogous to the case for asymptotic upper bounds, we are interested in finding a simple asymptotic lower bound which is as large as possible.

Notice the symmetry between "big-oh" and "big-omega": $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$.

Definition 3 $\Theta(f) \stackrel{\text{def}}{=} O(f) \cap \Omega(f)$.

Thus, if $g(n) \in \Theta(f)$ (pronounced "gee is in big theta of ef") then asymptotically $g(n)$ and $f(n)$ grow at the same rate within a positive multiplicative constant. In terms of this notation our previous concepts simplify to: $f(n)$ is an asymptotic upper bound (lower bound) on $T(n)$ which is as small as possible (as large as possible) if and only if $T(n) \in \Theta(f(n))$.

Exercise. Prove the following.
(i) $n^2 \in \Omega(n)$.
(ii) $n! \in \Omega(2^n)$.
(iii) $\Omega(n \log n) \subseteq \Omega(n)$.
(iv) $n \notin \Omega(n \log n)$.
(v) $n \in \Theta(n)$.
(vi) $3n + 1 \in \Theta(n)$.
(vii) $\sum_{i=1}^{n} \log_2 i \in \Theta(n \log_2 n)$. (Hint: For $1 \leq i \leq n$, $\log_2 i \leq \log_2 n$. For $\lceil \frac{n}{2} \rceil \leq i \leq n$, $\log_2 i \geq (\log_2 n) - 1$).
(viii) $\log(n^2) \in \Theta(\log n)$.

(ix) $\log^2 n \notin \Theta(\log n)$. Note: $\log^2 n$ denotes $(\log n)^2$.
(x) $p(n) \in \Theta(n^d)$ where p is any polynomial of degree d.
(xi) $\Theta(n \log^2 n + \sqrt{n} \log n) \neq \Theta(n \log n)$.

3 Useful Rules for "Big-Oh" Notation

The sets $O(f)$, $\Omega(f)$, and $\Theta(f)$ have the following useful properties.
(i) $g \in O(f)$ if and only if $f \in \Omega(g)$.
(ii) $O(f) = O(g)$ if and only if $f \in O(g)$ and $g \in O(f)$.
(iii) $O(f) = O(g)$ if and only if $f \in \Theta(g)$.
(iv) If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.

Exercise. Prove the above properties.

The following rules are useful for analysis of the running times of algorithms. (An example will be given in the next section to show how these rules can be used.)
Rule of Sums I: If $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$ then $g_1 + g_2 \in O(\max\{|f_1|, |f_2|\})$.

Rule of Sums II: Let $f_i, g_i : \mathbb{N} \to \mathbb{R}$, for $i \in \mathbb{N}$, be such that for some $c > 0$ and $n_0 \geq 0$, $|g_i(n)| \leq c \cdot |f_i(n)|$, for all $i \in \mathbb{N}$ and $n \geq n_0$. Let $k : \mathbb{N} \to \mathbb{N}$, and define $G(n) = \sum_{i=1}^{k(n)} g_i(n)$ and $F(n) = \sum_{i=1}^{k(n)} |f_i(n)|$. Then, $G \in O(F)$.

Rule of Products: If $g_1 \in O(f_1)$ and $g_2 \in O(f_2)$ then $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.

Exercise. Prove the above three rules.

Rule of Sums I is useful for finding an asymptotic upper bound on the time complexity of an algorithm which is composed of the execution of one algorithm followed by the execution of another algorithm. Rule of Sums II and Rule of Products are useful for determining the time complexity of loops.

By applying the Rule of Sums I or the Rule of Products repeatedly a *constant number of times* k, we obtain the following generalisation: Let $g_i \in O(f_i)$ for $i = 1, 2, \ldots, k$. Then

$$\begin{aligned} \sum_{i=1}^{k} g_i &\in O(\max\{|f_1|, |f_2|, \ldots, |f_k|\}) \\ \prod_{i=1}^{k} g_i &\in O(\prod_{i=1}^{k} f_i). \end{aligned}$$

These generalizations can be proved easily by mathematical induction on k.

It is very important to understand that in these generalisations, k *must* be a constant. The generalisations of the two rules obtained by repeatedly applying them a non-constant number of times are *not* valid. To see this, let $g(n) = 2$ for all $n \in \mathbb{N}$. Thus, $g(n) \in O(1)$. If we applied the rule of products n times with respect to $g(n)$ we would obtain $\prod_{i=1}^{n} g(n) \in O(1^n)$, i.e., $2^n \in O(1)$, which is patently false. What went wrong is that we are applying the rule of products a non-constant number of times.

4 Example of the time complexity analysis of a simple algorithm

Selection sort is an algorithm for sorting a array. It works roughly as follows: First it finds the smallest element in the array and interchanges it with the first element of the array; then it finds the second smallest element and interchanges it with the second element of the array, etc. Let A be the array filled with keys that is to be sorted. The procedure *swap*(a, b) interchanges the value of a with the value of b.

```
        Selection Sort(A : array[1..n])

        begin
(1)         for i := 1 to n - 1 do
                begin
(2)                 smallest := i
(3)                 for j := i to n do
(4)                     if A[j] < A[smallest] then smallest := j
(5)                 swap(A[i], A[smallest])
                end
        end
```

We first calculate the time for the i^{th} iteration of the main loop (which starts at line (1)). Then we obtain the time complexity of the entire algorithm by summing up the time complexities of all the iterations ($i = 1, 2, \ldots, n-1$). In the i^{th} iteration, we have to account for:

line (2) — takes constant ($O(1)$) time

line (3) — takes time equal to the number or repetitions of the loop times the time for line (4). The time for line (4) is in $O(1)$. Thus, by the Rule of Products, the time for line (3) is in $O(n-i+1)$.

line (5) — takes constant time.

Thus, the i^{th} iteration of the main loop is the total time for lines (2), (3) and (5), which, by the Rule of Sums I, is $\leq c \cdot (n-i+1)$, for some constant c (what do you think c should be?). By Rule of Sums II, the time complexity $T(n)$ of the entire algorithm is in

$$O\left(\sum_{i=1}^{n-1}(n-i+1)\right).$$

But,

$$\sum_{i=1}^{n-1}(n-i+1) = n + (n-1) + (n-2) + \cdots + 2 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1.$$

Thus, $T(n) \in O(n^2)$.