Asymptotic Notations

Richard Hua

Semester 2, 2021

Why do we need them?

- Elementary operations.
- Running time of algorithms.
- Can get complicated.
- A lot of annoying details.

Need something to make our lives easier.

Observation

Observation:

function SUM(array $a[0 \cdots n-1]$)
    $k \leftarrow 0$
    for (int $j = 0$; $j < n$; $j++$)
        $k += a[j]$
    return k

If we only count arithmetic operations in the main loop, it does n additions.
If we include additions needed for the counter variable, then we need $2n + 1$ additions, when n is relatively large, this is approximately $2n$.

Observation

It looks like the running time of a program which implements an algorithm is often $cf(n)$ where c is a constant factor depending on a computer, language, operating system, etc.
From previous examples, we have also seen that even if we don't know the actual value of c, we are still able to answer some of the more important questions. For example, if the input size increases from n_1 to n_2, how does the relative running time of the program change?
Increase by a factor of $T(n_2)/T(n_1) = cf(n_2)/cf(n_1)$.
We need some mathematical notations to avoid having to say 'of the order of ...' or 'roughly proportional to ...', and to make this intuition precise and clear.

Asymptotic notations

Most common and standard tool to do this are 'Big Oh' (O), 'Big Theta' (Θ) and 'Big Omega' (Ω).
Note the letter 'O' is supposed to be the capital 'omicron' in the Greek alphabet. Since the Greek omicron and the English 'O' are indistinguishable in most fonts supported by modern system, it has been replaced by 'O' and we read $O()$ as 'Big Oh' rather than 'Big Omicron'.

Asymptotic notations

Important assumptions:

- Constant factors are ignored (not always a good idea in practice).
- Running time functions have non-negative values.
- Input size is non-negative.

Big Oh

Definition

Let $f(n)$ and $g(n)$ be non-negative-valued functions defined on non-negative integers n. Then $g(n)$ is $O(f(n))$ (read '$g(n)$ is Big Oh of $f(n)$') if there exists a positive real constant c and a positive integer n_0 such that $g(n) \leqslant cf(n)$ for all $n > n_0$.

Big Oh

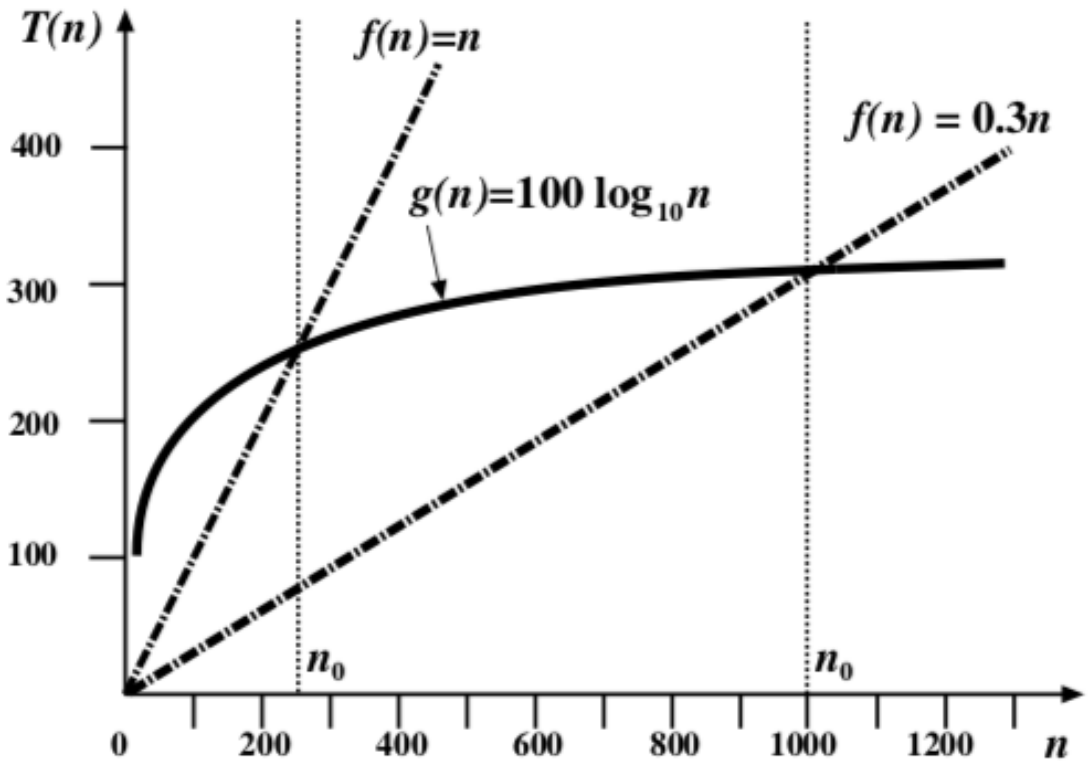

Figure: Big Oh property: $g(n)$ is $O(n)$.

Big Oh

If $g(n)$ is $O(f(n))$ then an algorithm with running time $g(n)$ runs for large n at least as fast, to with in a constant factor, as an algorithm with running time $f(n)$.
Usually the term 'asymptotically' is used in this context to describe behaviour of functions for large values of n.
This term means that $g(n)$ for large n may approach closer and closer to $cf(n)$.
$O(f(n))$ specifies an **asymptotic upper bound** for $g(n)$.

Asymptotic notations

Sometimes, the 'Big Oh' property is denoted $g(n) = O(f(n))$.
More precisely, we should say $g(n) \in O(f(n))$.
$O(f(n))$ is a set of functions which are increasing, in essence, with the same or lesser rate as n tends to infinity.

Big Omega

Definition

The function $g(n)$ is $\Omega(f(n))$ if there exists a positive real constant c and a positive integer n_0 such that $g(n) \geqslant cf(n)$ for all $n > n_0$.

'Big Omega' is complementary to 'Big Oh' and generalizes the concept of 'lower bound' in the same way as 'Big Oh' generalizes the concept of 'upper bound'.
$g(n)$ is $O(f(n))$ if and only if $f(n)$ is $\Omega(g(n))$.

Big Theta

Definition

The function $g(n)$ is $\Theta(f(n))$ if there exist two positive real constants c_1 and c_2 and a positive integer n_0 such that $c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)$ for all $n > n_0$.

$f(n)$ and $g(n)$ are of the same order.
$f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. i.e. $f(n)$ is both an asymptotic upper and lower bound for $g(n)$.
The 'Big Theta' property means that $f(n)$ and $g(n)$ have asymptotically tight bounds and are in some sense equivalent for our purposes.

Some more notes on asymptotic notations

The exact function, g, is not very important because it can be multiplied by an arbitrary constant.
The relative behaviour of two functions is compared only asymptotically with large n.
If the constants are very large, the asymptotic behaviour loses practical interest.

Question

Is ignoring the constant factor a good idea?

Example

function SUM(array $a[0 \cdots n-1]$)
    $k \leftarrow 0$
    for (int $j = 0$; $j < n$; $j++$)
        $k += a[j]$
    return k

Exercise

Show that SUM is an $O(n)$ algorithm.

Example

Let's prove the more general case: all linear functions $g(n) = an + b$; $a > 0$, is $O(n)$.
Proof: $g(n) \leqslant an + |b| \leqslant (a + |b|)n$ for all $n \geqslant 1$.

Example

Recall the exercise where we ignored the base of a log function.
For each $m > 1$, $g(n) = \log_m(n)$ has the same rate of increase as $\lg(n)$.
$\log_m(n) = \log_m(2) \lg(n)$ for all $n > 0$.
We often ignore the log base when using asymptotic notations.
e.g. $\lg n = \Theta(\log n)$.

Exercise

Exercise

Show that $f(n) = \sqrt{34}n^3 + 80n \lg n + 53$ *is* $O(n^3)$.

Scaling

Lemma

For all constants $c > 0$, cf is $O(f)$.

Proof: The relationship $cf(n) \leqslant cf(n)$ obviously holds for all $n \geqslant 0$.

Transitivity

Lemma

If h is $O(g)$ and g is $O(f)$, then h is $O(f)$.

Proof: We have h is $O(g)$ so $h \leqslant c_1 g$ for all $n > n_1$. Since g is $O(f)$ we also have $g \leqslant c_2 f$ for all $n > n_2$.
So $h \leqslant c_1 g \leqslant c_1 c_2 f$ when $n \geqslant \max\{n_1, n_2\}$.

Rule of sums

Lemma

If g_1 is $O(f_1)$ and g_2 is $O(f_2)$, then $g_1 + g_2$ is $O(max\{f_1, f_2\})$.

We have $g_1 \leqslant c_1 f_1$ for all $n > n_1$ and $g_2 \leqslant c_2 f_2$ for all $n > n_2$.
For all $n \geqslant \max\{n_1, n_2\}$,
$g_1 + g_2 \leqslant c_1 f_1 + c_2 f_2 \leqslant \max\{c_1, c_2\}(f_1 + f_2)$.
Note that $f_1 + f_2 \leqslant 2\max\{f_1, f_2\}$, so $g_1 + g_2 \leqslant c\max\{f_1, f_2\}$ where $c = 2\max\{c_1, c_2\}$.

Rule of products

Lemma

If g_1 is $O(f_1)$ and g_2 is $O(f_2)$, then $g_1 g_2$ is $O(f_1 f_2)$.

Exercise

Prove the lemma above.

Limit rule

Lemma

Suppose $\lim_{n\to\infty} f(n)/g(n)$ *exists (may be* ∞*), and denote the limit by* L*. Then*

- *if* $L = 0$*, then* f *is* $O(g)$ *and* f *is not* $\Omega(g)$*;*
- *if* $0 < L < \infty$ *then* f *is* $\Theta(g)$*;*
- *if* $L = \infty$ *then* f *is* $\Omega(g)$ *and* f *is not* $O(g)$*.*

Exercise

Prove the lemma above.

Limit rule

Lemma

Suppose $\lim_{n\to\infty} f(n)/g(n)$ *exists (may be* ∞*), and denote the limit by* L*. Then*

- *if* $L = 0$*, then* f *is* $O(g)$ *and* f *is not* $\Omega(g)$*;*
- *if* $0 < L < \infty$ *then* f *is* $\Theta(g)$*;*
- *if* $L = \infty$ *then* f *is* $\Omega(g)$ *and* f *is not* $O(g)$*.*

Note that if the limit doesn't exist, it doesn't mean there is no asymptotic relations between f and g.
Example, let $f = n$ and $g = (2 + (-1)^n)n$.
Clearly, g is $O(f)$, but the limit doesn't converge as $n \to \infty$

Exercise

Let $T(n) = f(n)g(n) + h(n)$ where $f(n)$ is $\Theta(n)$; $g(n)$ is $O(n \log n)$, and $h(n)$ is $\Omega(n^3)$. Which of the following statements are true?

A. $T(n)$ is $\Theta(n^3)$
B. $T(n)$ is $O(n^2 \log n)$
C. $T(n)$ is $\Theta(n^2 \log n)$
D. $T(n)$ is $O(n^3)$
E. $T(n)$ is $\Omega(n^3)$

Exercise

Exercise

Show that $T(n) = 10n^3 - 5n + 15$ *is not* $O(n^2)$.

Proof: Suppose $T(n)$ is $O(n^2)$. So we must have
$10n^3 - 5n + 15 \leqslant cn^2$ for all $n > n_0$.
$10n - 5/n + 15/n^2 \leqslant c$ for all $n > n_0$ so
$n - 0.5n^{-1} + 1.5n^{-2} \leqslant 0.1c$.
Let $k = \lceil 0.1c \rceil + 1$, then for all $n > k + 1 > 2$. We have
$0.5n^{-1} < 0.5/(k+1) < 1$ and $1.5n^{-2} < 1.5/(k+1)^2 < 1$.
$n - 0.5n^{-1} + 1.5n^{-2} > n - 1 > k \geqslant 0.1c$.

Asymptotic notations

The Big-Θ asymptotic notation is what's called an **equivalence relation** in mathematics.

- Reflexive. e.g. for all f, f is $\Theta(f)$.
- Symmetric. e.g. for all f and g, if f is $\Theta(g)$ then g is $\Theta(f)$
- Transitive.

Time complexity

Definition

A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the **time complexity** of the algorithm. (informal)

An algorithm is called **polynomial time** if its running time $T(n)$ is $O(n^k)$ where k is some fixed positive integer.
A computational problem is considered **intractable** iff no deterministic algorithm with polynomial time complexity exists for it.
Many problems are classed as intractable only because a polynomial time solution is unknown.

Time complexity

Definition

A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the **time complexity** of the algorithm. (informal)

In practice, quadratic and cubic algorithms cannot be used if the input size exceeds tens of thousands of items and exponential algorithms should be avoided whenever possible.
By optimizing our program, compiler, language etc., we are effectively only changing the hidden constant c slightly.

Time complexity

Definition

A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the **time complexity** of the algorithm. (informal)

Definition

Suppose we have an algorithm for a specific problem with time complexity $\Theta(f(n))$ and there is no algorithm with time complexity $\Theta(g(n))$ for any function g that grows more slowly than f when $n \to \infty$. Then we say that the algorithm is **asymptotically optimal** (or just **optimal**) for the given problem. (informal)

Useful but often quite hard to do in practice.

Additional notes

- Limitations of our theoretical framework.
- How do we check our analysis is correct? Hard to do sometimes.
- Large constants should be taken into account in practice.
- With very large input size, even asymptotic analysis may break down.
- Elementary operations used to be very accurate in predicting actual program runtimes. But not so anymore.
- The basic distinction between linear, quadratic, exponential time, etc. is still very clear (i.e. asymptotic notation give us a very good way of comparing different algorithms).