Lecture 1 : Revision on Big-O analysis

Justin Pearson — Based on Slides by Pierre Flener

November 1, 2022

Lecture Plan

- Revision on Algorithm analysis.
  - Counting the number of steps
  - Asymptotic analysis
- Big-O, $O(f(n))$, Big-Theta $\Theta(f(n))$, and big-Omega $\Omega(f(n))$.

Next Lecture:

- Some examples:
  - Merge Sort
  - Quick Sort
- The master theorem.

- The most important take away from this lecture is understanding the definition of $O(f(n))$, $\Theta(f(n))$ and $\Omega(f(n))$.
- The definition is quite subtle, but once you get the hang of it you will have a nice way of expressing complexity results.

Runtime Equations

Consider the following function (defined functionally, but easy to translate to Python), which returns the sum of the elements of the given integer list:

```
fun sumList [] = 0
  | sumList (x::xs) = x + sumList xs
```

The runtime T of this function depends on the given list. We will assume that we are using fixed precision numbers, and so the runtime is a function of the length of the list.

Runtime Equations

Assuming that [pattern matching and] the + operation takes the same time t_{add} regardless of the two numbers being added, we can see that only the length n of the list matters to T.

We can express $T(n)$ recursively mirroring the function definition.

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{\text{add}} & \text{if } n > 0 \end{cases}$$

where t_0 (the time of [pattern matching and] returning 0) and t_{add} are constants (that is, they do not depend on n).

Solving Recurrences

The expression for $T(n)$ is called a recurrence. We can use it for computing runtimes (given actual values of the constants t_0 and t_{add}), but it is difficult to work with.

For example $T(3) = T(2) + t_{\text{add}} = (T(1) + t_{\text{add}}) + t_{\text{add}} = T(0) + t_{\text{add}} + t_{\text{add}} + t_{\text{add}} = t_0 + 3 \cdot t_{\text{add}}$

Solving Recurrences

We prefer a closed form (that is, a non-recursive equation), if possible.

Equivalent definition of $T(n)$, for all $n \geq 0$:

$$T(n) = n \cdot t_{\text{add}} + t_0$$

Much simpler! But: How do we get there? Can we prove it?

Deriving Closed Forms

There is no general way of solving recurrences.
Recommended method:

First guess the answer, and then prove it by induction!

Suggestions for making a good guess:

- If the recurrence is similar (upon variable substitution) to one seen before, then guess a similar closed form.
- Expansion Method: Detect a pattern for several values. Example:

$$
\begin{array}{rcl}
T(0) & = & t_0 \\
T(1) & = & T(0) + t_{\text{add}} = 1 \cdot t_{\text{add}} + t_0 \\
T(2) & = & T(1) + t_{\text{add}} = 2 \cdot t_{\text{add}} + t_0 \\
T(3) & = & T(2) + t_{\text{add}} = 3 \cdot t_{\text{add}} + t_0
\end{array}
$$

Proof by Induction

Let:

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{\text{add}} & \text{if } n > 0 \end{cases} \tag{1}$$

Theorem: $T(n) = n \cdot t_{\text{add}} + t_0$, for all $n \geq 0$.

Proof

Basis: If $n = 0$, then $T(n) = t_0 = 0 \cdot t_{\text{add}} + t_0$, by the recurrence.

Induction: Assume $T(n) = n \cdot t_{\text{add}} + t_0$ for some $n \geq 0$. Then:

$$\begin{aligned} T(n+1) &= T(n) + t_{\text{add}}, \text{ by the recurrence} \\ &= (n \cdot t_{\text{add}} + t_0) + t_{\text{add}}, \text{ by the assumption above} \\ &= (n+1) \cdot t_{\text{add}} + t_0, \text{ by arithmetic laws} \quad \square \end{aligned}$$

Back to Algorithm Analysis

The equation $T(n) = n \cdot t_{\text{add}} + t_0$ is a useful, but approximate, predictor of the actual runtime of `sumList`.

Even if t_0 and t_{add} were measured accurately, the actual runtime would vary with every change in the hardware or software environment. There might be effects with due to the cache, there might be other processes running or whatever.

The actual values of t_0 or t_{add} are not really interesting, we are more interested in how the function grows with n.

Back to Algorithm Analysis

Looking at the equation

$$T(n) = n \cdot t_{\mathrm{add}} + t_0$$

The only interesting part of the equation is the term with n. The runtime of `sumList` is (within constant factor t_{add}) proportional to the length n of the list.

Calling `sumList` with a list twice as long will approximately double the runtime.

Asymptotic notation

There is a precise mathematical notation for describing how functions behave up to constant factors and ignoring lower order terms. There are three definitions to understand (formal definitions soon):

- Big-O $O(f(n))$ is a set of functions that never grow faster than $f(n)$ up to a constant factor for sufficiently large n.
- Big-Omega $\Omega(f(n))$ is a set of functions that always grow faster than $f(n)$ up to a constant factor for sufficiently large n.
- Big-Theta $\Theta(f(n))$ is a set of functions that are bounded above and below by $f(n)$ by a constant factor for sufficiently large n.

The $O(f(n))$ notation

Important: $O(f(n))$ is a set.

We say that $g(n) \in O(f(n))$ if there exists a positive constant k and n_0 such that for all $n \geq n_0$ we have that:

$$g(n) \leq k \cdot f(n)$$

The $O(f(n))$ notation

Let's prove that $t_{\text{add}} \cdot n + t_0 \in O(n)$. The proof strategy is to look at the definition of $O(n)$ and try to find the relevant constants.

Set $n_0 = 1$, then we need to find a constant k such that for all $n \geq 1$ we have that

$$t_{\text{add}} \cdot n + t_0 \leq k \cdot n$$

Set $k = t_{\text{add}} + t_0$ then $k \cdot n = t_{\text{add}} \cdot n + t_0 \cdot n$ and

$$t_{\text{add}} \cdot n + t_0 \leq t_{\text{add}} \cdot n + t_0 \cdot n$$

is true from simple arithmetic.

General Proof strategy

You need to find values of n_0 and k that make the inequality true:

$$g(n) \leq k \cdot f(n)$$

You do not have to find the tightest value of k and the smallest value of n_0.

Anything will do. Sometimes it is good to pick a larger n_0, because the function is a bit too erratic for small values of n.

The $\Omega(f(n))$ Notation

Important: $\Omega(f(n))$ is a set.

We say that $g(n) \in \Omega(f(n))$ if there exists a positive constant k and n_0 such that for all $n \geq n_0$ we have that:

$$k \cdot f(n) \leq g(n)$$

The $\Omega(f(n))$ notation

Let's prove that $t_{\text{add}} \cdot n + t_0 \in \Omega(n)$. The proof strategy is to look at the definition of $\Omega(n)$ and try to find the relevant constants.

Set $n_0 = 1$, then we need to find a constant k such that for all $n \geq 1$ we have that

$$k \cdot n \leq t_{\text{add}} \cdot n + t_0$$

This is much easier than before : Set $k = t_{\text{add}}$ then we have that

$$t_{\text{add}} \cdot n \leq t_{\text{add}} \cdot n + t_0$$

is true from simple arithmetic.

The $\Theta(f(n))$ Notation

Important: $\Theta(f(n))$ is a set.

We say that $g(n) \in \Theta(f(n))$ if there exists positive constants c_1, c_2 and n_0 such that for all $n \geq n_0$ we have that

$$c_1 f(n) \leq g(n) \leq c_2 f(n)$$

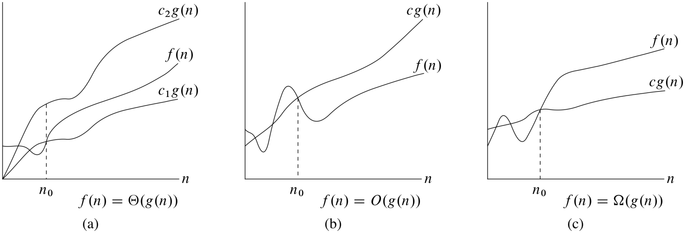
The $\Theta(f(n))$ Notation

Note that

$$g(n) \in \Theta(f(n)) \Leftrightarrow g(n) \in \Omega(f(n)) \wedge g(n) \in O(f(n))$$

The definition of $\Theta(f(n))$ requires two constants c_1 and c_2 (see the previous slide, you get one constant from your proof of membership in $\Omega(f(n))$ and one constant from your proof of membership in $O(f(n))$

Illustrating the definitions of Θ,Ω and O



$\Theta(f(n))$ Example

To prove that $t_{\text{add}} \cdot n + t_0 \in \Theta(n)$ set $c_1 = t_{\text{add}}$, $c_2 = t_{\text{add}} + t_0$, and $n_0 = 1$ then for all $n \geq n_0$ we have:

$$c_1 n \leq t_{\text{add}} \cdot n + t_0 \leq c_2 n$$

Another Example

$n^2 + 5 \cdot n + 10 \in \Theta(n^2)$.

Proof: We need to choose constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that

$$0 \leq c_1 \cdot n^2 \leq n^2 + 5 \cdot n + 10 \leq c_2 \cdot n^2$$

for all $n \geq n_0$. Dividing by n^2 (assuming $n > 0$) gives

$$0 \leq c_1 \leq 1 + \frac{5}{n} + \frac{10}{n^2} \leq c_2$$

The "sandwiched" term, $1 + \frac{5}{n} + \frac{10}{n^2}$, gets smaller as n grows.
It peaks at 16 for $n = 1$, so we can pick $n_0 = 1$ and $c_2 = 16$.
It drops to 6 for $n = 2$ and becomes close to 1 for $n = 1000$.
It never gets less than 1, so we can pick $c_1 = 1$. $\square$

Some observations

- $f(n) \in \Theta(f(n))$ (why?
- $\Theta(n^2 + n) = \Theta(n^2)$.
- $\Theta(n^3 + n^2 + n) = \Theta(n^3)$.

You can simplify a Θ, Ω or O set by only considering the dominating term. It is common practice to state the complexity result as simply as possible and ignore all the smaller terms.

Keeping Complexity Functions Simple

We can simplify complexity functions by:

- Setting all constant factors to 1.
- Dropping all lower-order terms.

Since $\log_b n = \frac{1}{\log_c b} \cdot \log_c n$, where $\frac{1}{\log_c b}$ is a constant factor (when the bases b and c are constants), it does not matter if we write $\log n$ or $\ln n$ or $\log_b n$. Computer scientists use $\log_2$ while mathematicians use $\log_e = \ln$.

Warning

A lot of people write $g(n) = \Theta(f(n))$ were here we write $g(n) \in \Theta(f(n))$. This is fine as long as you know that it is not equality.

If you forget that it is really set membership then you derive all sorts of contradictions:

$$n^2 + n = \Theta(n^2 + n) = \Theta(n^2)$$

$$n^2 = \Theta(n^2)$$

and so

$$n^2 + n = n^2$$

Next Lecture

- Some examples of recurrence relations from sorting algorithms
- Solving recurrence relations and the master theorem.