Analysis of Algorithms

[Reading: CLRS 2.2, 3]

Laura Toma, csci2200, Bowdoin College

- Why analysis? We want to predict how the algorithm will behave (e.g. running time) on arbitrary inputs, and how it will compare to other algorithms. We want to understand its bottlenecks and the patterns behind its behaviour.
- Theoretical analysis: we break down the algorithm in instructions, and count the number of instructions. Note that the result depends crucially on what exactly we consider to be an *instruction*.
- What is an *instruction* ? An instruction should be primitive (in the sense that it corresponds to a machine instruction on a real computer; e.g. cannot count "sorting" as one operation) and at the same time high-level enough (so that it's independent of the specific processor and platform).
- More formally, to define an instruction, we need to define a model of a generic computer. This is called the RAM model of computation:

  Random-access machine (RAM) model of computation:
  - Contains all the standard instructions available on any computer:
    - Load/Store, Arithmetics (e.g. $+, -, *, /$), Logic (e.g. $>$), jumps
  - All instructions take the same amount of time
  - Instructions executed sequentially one at a time
- To analyse an algorithm, we implicitly assume the RAM model of computation (all instructions take the same) and we count the number of instructions. This will be our estimate of running time.

  The **running time** of an algorithm is **the number of instructions it executes in the RAM model of computation**.
- RAM model not completely realistic, e.g.
  - main memory not infinite (even though we often imagine it is when we program)
  - not all memory accesses take same time (cache, main memory, disk)
  - not all arithmetic operations take same time (e.g. multiplications expensive)
  - instruction pipelining, etc
- But RAM model gives realistic results in most cases.

- The running time is expressed as a function of the *input size*
    - E.g: running time of a sorting algorithm is expressed as a function of n, the number of elements in the input array.
- Not always possible to come up with a precise expression of running time for a specific input.
    - E.g. Running time of insertion sort on a specific input depends on that input; we can;t nail it down without knowing what the specific input is.

  We are interested in the smallest and the largest running time for an input of size n:

- **Best-case running time:** The shortest possible running time for *any* input of size n.
- **Worst-case running time:** The longest possible running time for *any* input of size n.

  Sometimes we might also be interested in:

- Average-case running time: The average running time over a set of inputs of size n.
    - Must be careful: Need to assume an input distribution.
- Unless otherwise specified, we are interested in the worst-case running time. Wcrt gives us a guarantee.
- For some algorithms, worst-case occur fairly often.
- Average case is often as bad as worst case (but not always!).
- Case study: Express the best-case and worst case running times of bubble-sort, selection sort and insertion sort, and give examples of inputs that trigger best case and worst-case behaviour, respectively.

Asymptotic analysis

- Assume we've done a careful analysis, counting all instructions, and found that:

  Algorithm X best case is $3n + 5$ (linear), and worst case is $2n^2 + 5n + 7$ (quadratic).
- The effort to compute all terms and the constants in front of the terms is not worth it for two reasons:
    1. these constants are not accurate because the RAM model is not completely realistic (not all operations cost the same).
    2. for large input the running time is dominated by the higher order term
- For these reasons, when analysing algorithms, we are not interested in all the constants and all the lower order terms, but in the overall *rate of growth* of the running time. This is called *aymptotic analysis* (because it's done when $n \to \infty$).
- The rate of growth is expressed using O-notation, Ω-notation, Θ-notation. You have probably seen it intuitively defined but now we will now define it more carefully.

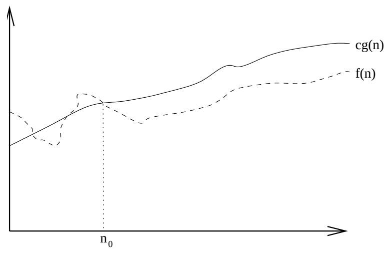
- Notation: Let $f(n), g(n)$ be non-negative functions (think of them as representing the the worst-case running time of two algorithms).

O-notation (pronounced: Big-O)

We say that f is $O(g(n))$ if, for sufficiently large n, the function $f(n)$ is bounded above by a constant multiple of $g(n)$. Put differently, if f is upper bounded by g in the asymptotic sense, i.e. as n goes to ∞. More precisely,

$$\boxed{f \text{ is } O(g(n)) \text{ if } \exists\ c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n)\ \forall n \geq n_0}$$

We think of $f(n) \in O(g(n))$ as corresponding to $f(n) \leq g(n)$.



Examples:

- $\frac{1}{3}n^2 + 3n$ is $O(n^2)$ because $\frac{1}{3}n^2 + 3n \leq \frac{1}{3}n^2 + 3n^2 = (\frac{1}{3} + 3)n^2$; holds for $c = 1/3 + 3 = 3.3$ and $n > 1$.
- $f(n) = 100n^2$, $g(n) = n^2$: $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$
- $f(n) = n, g(n) = n^2$: $f(n) \in O(g(n))$
- $20n^2 + 10n \lg n + 5$ is $O(n^3)$
- 2^{100} is $O(1)$ ($O(1)$ denotes constant time)
- Note that $O(\cdot)$ expresses an upper bound of f, but not necessarilly tight:

$$n \in O(n), n \in O(n^2), n \in O(n^3), n \in O(n^{100})$$

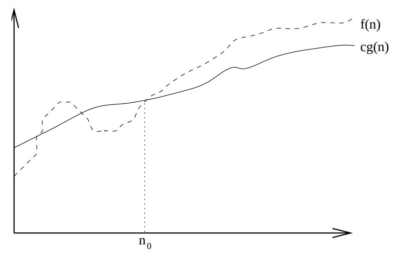
We can prove that an algorithm has running time $O(n^3)$, and then we analyse more carefully and manage to show that the running time is in fact $O(n^2)$. The first bound is correct, but the second one is better. We want the "tightest" possible bound for the running time.

Ω-notation (big-Omega)

We say that f is $\Omega(g(n))$ if, for sufficiently large n, the function $f(n)$ is bounded below by a constant multiple of $g(n)$. Put differently, if f is lower bounded by g in the asymptotic sense, i.e. as n goes to ∞. More precisely,

$$\boxed{f(n) \text{ is } \Omega(g(n)) \text{ if } \exists\ c > 0, n_0 > 0 \text{ such that } cg(n) \leq f(n)\ \forall n \geq n_0\}}$$

We think of $f(n) \in \Omega(g(n))$ as corresponding to $f(n) \geq g(n)$.



Examples:

- $\frac{1}{3}n^2 + 3n \in \Omega(n^2)$ because $\frac{1}{3}n^2 + 3n \geq cn^2$; true if $c = 1/3$ and $n > 0$.
- $f(n) = an^2 + bn + c$, $g(n) = n^2$: $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(f(n))$
- $f(n) = 100n^2$, $g(n) = n^2$: $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(f(n))$
- $f(n) = n, g(n) = n^2$: $g(n) \in \Omega(f(n))$
- Note that $\Omega(\cdot)$ gives a lower bound of f, but not necessarilly tight:

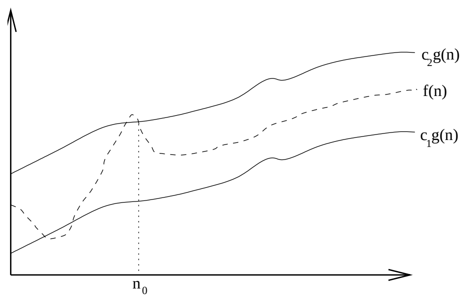
$$n \in \Omega(n), n^2 \in \Omega(n), n^3 \in \Omega(n), n^{100} \in \Omega(n)$$

0.1 Θ-notation (Big-Theta)

$\Theta()$ is used to express asymptotically *tight* bounds. We say that $g(n)$ is a tight bound for $f(n)$, or $f(n) is \Theta(g(n))$, if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$. That is, $f(n)$ grows like $g(n)$ to within a constant factor.

$$\boxed{f(n) \text{ is } \Theta(g(n)) \text{ if } f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n))}$$

We think of $f(n) \in \Theta(g(n))$ as corresponding to $f(n) = g(n)$.



Examples:

- $6n\log n + \sqrt{n}\log^2 n \in \Theta(n\log n)$
- $n \neq \Theta(n^2)$
- $6n\lg n + \sqrt{n}\lg^n = \Theta(n\lg n)$

When analysing the worst or best case running time of an algorithm we aim for asymptotic tight bounds because they characterize the running time of an algorithm precisely up to constant factors.

Rate of growth and limits

Another way to think of rate of growth is to calculate the limit of the functions:

$$lim_{n\longrightarrow\infty}\frac{f(n)}{g(n)}$$

Using the definition of O, Ω, Θ it can be shown that :

- if $lim_{n\longrightarrow\infty}\frac{f(n)}{g(n)} = 0$: then intuitively $f < g \Longrightarrow f = O(g)$ and $f \neq \Theta(g)$.
- if $lim_{n\longrightarrow\infty}\frac{f(n)}{g(n)} = \infty$: then intuitively $f > g \Longrightarrow f = \Omega(g)$ and $f \neq \Theta(g)$.
- if $lim_{n\longrightarrow\infty}\frac{f(n)}{g(n)} = c, c > 0$: then intuitively $f = c \cdot g \Longrightarrow f = \Theta(g)$.

Thus comparing f and g in terms of their growth can be summarized as:

- f is below $g \Leftrightarrow f \in O(g) \Leftrightarrow f \leq g$
- f is above $g \Leftrightarrow f \in \Omega(g) \Leftrightarrow f \geq g$
- f is both above and below $g \Leftrightarrow f \in \Theta(g) \Leftrightarrow f = g$

Growth rate of standard functions

- Polynomial of degree d is defined as:

$$p(n) = \sum_{i=1}^{d} a_i \cdot n^i = \Theta(n^d)$$

  where $a_1, a_2, \ldots, a_d$ are constants (and $a_d > 0$).

Any polylog grows slower than any polynomial.

$$\log^a n = O(n^b), \forall a > 0$$

Any polynomial grows slower than any exponential with base $c > 1$.

$$n^b = O(c^n), \forall b > 0, c > 1$$

Asymptotic analysis, continued

- Upper and lower bounds are symmetrical: If f is upper-bounded by g then g is lower-bounded by f. Example: $n \in O(n^2)$ and $n^2 \in \Omega(n)$

- The correct way to say is that $f(n) \in O(g(n))$ or $f(n)$ is $O(g(n))$. Abusing notation, people often write $f(n) = O(g(n))$.

$$3n^2 + 2n + 10 = O(n^2), n = O(n^2), n^2 = \Omega(n), n \log n = \Omega(n), 2n^2 + 3n = \Theta(n^2)$$

- Example: Insertion-sort:
  - Best case: $\Omega(n)$
  - Worst case: $O(n^2)$
  - Therefore the running time is $\Omega(n)$ and $O(n^2)$.
  - We can actually say that worst case is $\Theta(n^2)$ because there exists an input for which insertion sort takes $\Omega(n^2)$. Same for best case.
  - But, we cannot say that the running time of insertion sort is $\Theta(n^2)$!
- Use of O-notation makes it much easier to analyze algorithms; we can easily prove the $O(n^2)$ insertion-sort time bound by saying that both loops run in $O(n)$ time.
- We often use $O(n)$ in equations and recurrences: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$).
- We use $O(1)$ to denote constant time.
- Not all functions are asymptotically comparable! There exist functions f, g such that f is not $O(g)$, f is not $\Omega(g)$ (and f is not $\Theta(g)$).

Algorithms matter!

Sort 10 million integers on

- 1 GHZ computer (1000 million instructions per second) using $2n^2$ algorithm.
  - $\frac{2\cdot(10^7)^2 \ \textit{inst.}}{10^9 \ \textit{inst. per second}} = 200000$ seconds ≈ 55 hours.
- 100 MHz computer (100 million instructions per second) using $50n \log n$ algorithm.
  - $\frac{50\cdot 10^7 \cdot \log 10^7 \ \textit{inst.}}{10^8 \ \textit{inst. per second}} < \frac{50\cdot 10^7\cdot 7\cdot 3}{10^8} = 5\cdot 7\cdot 3 = 105$ seconds.

Review of Log and Exp

- Base 2 logarithm comes up all the time (from now on we will always mean $\log_2 n$ when we write $\log n$ or $\lg n$).
  - Number of times we can divide n by 2 to get to 1 or less.
  - Number of bits in binary representation of n.
  - Note: $\log n << \sqrt{n} << n$
- Properties:
  - $\lg^k n = (\lg n)^k$
  - $\lg \lg n = \lg(\lg n)$
  - $a^{\log_b c} = c^{\log_b a}$
  - $a^{\log_a b} = b$
  - $\log_a n = \frac{\log_b n}{\log_b a}$
  - $\lg b^n = n \lg b$
  - $\lg xy = \lg x + \lg y$
  - $\log_a b = \frac{1}{\log_b a}$