Analysis of Algorithms

(CLRS 2.2, 3)

1 Algorithm analysis

- Why? We want to predict how the algorithm will behave (e.g. running time) on arbitrary inputs, and how it will compare to other algorithms
- Of course, the most accurate way is: implement, run and measure (Can you think of cons with this?).
- We want to understand the pattern behind behaviour of algorithm $\Longrightarrow$ theoretical analysis
- To analyze an algorithm theoretically, we break down the algorithm in high-level primitive operations (independent of programming language and platform), such as:
  - assigning a variable
  - performing an arithmetic operation
  - comparing two numbers, etc
- A primitive operation corresponds to a low-level machine instruction
- The time of the operations differ a little, by a small constant; basically an operation may take beween a few clock cycles to a hundred clock cycles or so.
- For the purpose of analysis, we consider that all operations take the same amount of time.
- To analyize: we count the number of operations. This will be our estimate of running time.
- The formal name for this is the RAM model:

Random-access machine (RAM) model:

- Instructions executed sequentially one at a time
- All instructions take unit time:
  - Load/Store
  - Arithmetics (e.g. $+, -, *, /$)
  - Logic (e.g. $>$)
- Main memory is infinite

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
  - other processes
- But RAM model is good enough to give realistic results.

- We are interested in running time as a function of *input size*—usually denoted by n.
  - E.g. running time of a sorting algorithm is expressed function of n=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 are interested in:

- **Best-case running time:** The shortest running time for any input of size n. The algorithm will never be faster than this.
- **Worst-case running time:** The longest running time for *any* input of size n. The algorithm will never be slower than this.

Sometimes we might also be interested in:

- **Average-case running time:** The average running time over.....
  - Be careful: average over what? Must assume an input distribution.

- Unless otherwise specified, we are interested in worst-case running time (wcrt). Wcrt gives us a guarantee.
- For some algorithms, worst-case occur fairly often.
- Average case 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.

2 Asymptotic analysis

Unless otherwise specified, we are interested in the w.c.r.t of an algorithm. Sometimes we might want to analyze the b.c.r.t.

E.g.: Algorithm X best case is $3n+5$ (linear), and worst case is $2n^2+5n+7$ (quadratic). (note: these are made-up numbers).

For Insertion-sort we can do a precise analysis and find that the worst-case is $k_3n^2 + k_4n + k_5$. The effort to compute all terms and the constants in front of the terms is not really worth it, because for large input the running time is dominated by the term n^2. Another good reason for not caring about constants and lower order terms is that the RAM model is not completely realistic anyway (not all operations cost the same).

$$k_3n^2 + k_4n + k_5 \sim n^2$$

Basically, we look at the running time of an algorithm when the input size n is large enough so that constants and lower-order terms do not matter. This is called **aymptotic analysis of algorithms**.

Now we would like to formalize this idea (It is easy to see that $n+2 \sim n$, or that $4n^2+3n+10 \sim n^2$. But how about more complicated functions? say $n^n + n! + n^{\log\log n} + n^{1/logn}$).

$\Downarrow$

We want to express **rate of growth** of a function:

- the dominant term with respect to n
- ignoring constants in front of it

$k_1n + k_2 \sim n$
$k_1n \log n \sim n \log n$
$k_1n^2 + k_2n + k_3 \sim n^2$

We also want to formalize that e.g. an $n \log n$ algorithm is better than an n^2 algorithm.

$\Downarrow$

Growth of rate of a function: O-notation, Ω-notation, Θ-notation. You have probably seen it intuitively defined but now we will now define it more carefully.

Assume we have two functions $f, g : N \longrightarrow R$ that represent running times. 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 $g \Leftrightarrow f \in \Theta(g) \Leftrightarrow f = g$

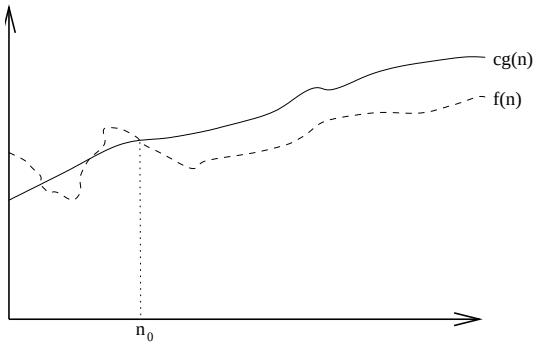
More formal definitions follow.

2.1 O-notation (pronounced: Big-O)

Let $f(n)$ be a non-negative function, say the worst-case running time of a certain algorithm. 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 $g(n)$. More precisely,

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

- f is upper bounded by g in the asymptotic sense, i.e. as n goes to ∞.
- $O(\cdot)$ is used to asymptotically *upper bound* a function.
- 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 $n > 1$.
- $k_1n^2+k_2n+k_3$ is $O(n^2)$ because $k_1n^2+k_2n+k_3 < (k_1+|k_2|+|k_3|)n^2$ and for $c > k_1+|k_2|+|k_3|$ and $n \geq 1$, $k_1n^2 + k_2n + k_3 < cn^2$.
- $k_1n^2 + k_2n + k_3$ is $O(n^3)$ as $k_1n^2 + k_2n + k_3 < (k_1 + k_2 + k_3)n^3$
- $f(n) = 100n^2$, $g(n) = n^2$
  - $f(n) \in O(g(n))$
  - $g(n) \in O(f(n))$
- $f(n) = n, g(n) = n^2$
  - $f(n) \in O(g(n))$
- $20n^2 + 10n \lg n + 5$ 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:

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

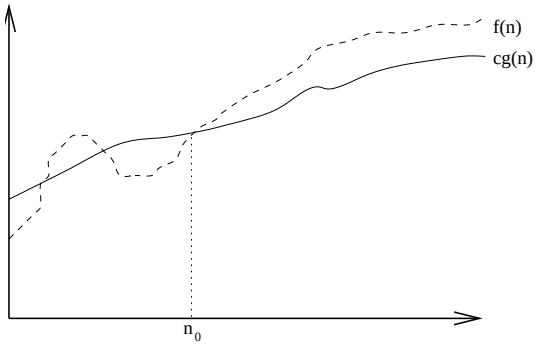
We can prove that an algorithm has running time $O(n^3)$, and then we analyse more carefully and manage to show that the running time is in fact $O(n^2)$. The first bound is correct, but the second one is better. We want the "tightest" possible bound for the running time.

2.2 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 $g(n)$. More precisely,

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

- $\Omega(\cdot)$ is used to asymptotically *lower bound* a function.
- 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 $n > 0$.
- $k_1n^2 + k_2n + k_3 \in \Omega(n^2)$.
- $k_1n^2 + k_2n + k_3 \in \Omega(n)$
- $f(n) = an^2 + bn + c$, $g(n) = n^2$
  - $f(n) \in \Omega(g(n))$
  - $g(n) \in \Omega(f(n))$
- $f(n) = 100n^2$, $g(n) = n^2$
  - $f(n) \in \Omega(g(n))$
  - $g(n) \in \Omega(f(n))$
- $f(n) = n, g(n) = n^2$
  - $g(n) \in \Omega(f(n))$

Note that $\Omega(\cdot)$ gives a lower bound of f, but not necessarilly tight:

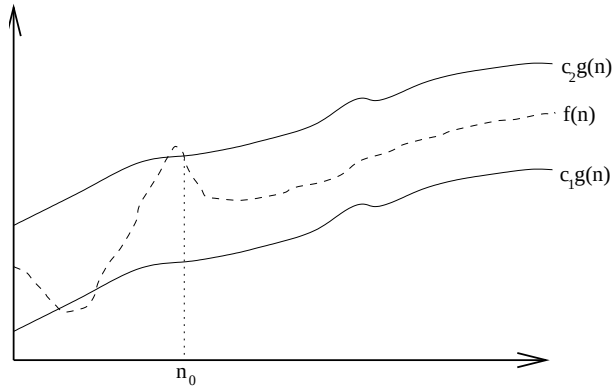
- $n \in \Omega(n)$, $n^2 \in \Omega(n)$, $n^3 \in \Omega(n)$, $n^{100} \in \Omega(n)$

2.3 Θ-notation (Big-Theta)

Θ() is used to give asymptotically tight bounds. Essentially, if we can show that a running time $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$, then $g(n)$ is a tight bound for $f(n)$. That is, $f(n)$ grows like $g(n)$ to within a constant factor.

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

- $\Theta(\cdot)$ is used to asymptotically *tight bound* a function.
- We think of $f(n) \in \Theta(g(n))$ as corresponding to $f(n) = g(n)$.



Examples:

- $k_1 n^2 + k_2 n + k_3 \in \Theta(n^2)$
- $6n \log n + \sqrt{n} \log^2 n \in \Theta(n \log n)$
- $an^2 + bn + c \in O(n^2), an^2 + bn + c \in \Omega(n^2) \longrightarrow an^2 + bn + c \in \Theta(n^2)$
- $n \neq \Theta(n^2)$
- $6n \lg n + \sqrt{n} \lg^n = \Theta(n \lg n)$

Asymptotic tight bounds are nice to find, because they characterize the running time of an algorithm precisely up to constant factors.

3 Growth and limits

The growth of two functions f and g can be found by computing the limit $lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)}$. 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 $f = c \cdot g \Longrightarrow f = \Theta(g)$.

This will be useful when doing exercises.

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

5 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 normally write $f(n) = O(g(n))$.

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

- When we say "the running time is $O(n^2)$" we mean that the worst-case running time is $O(n^2)$.
- When we say "the running time is $\Omega(n^2)$", we mean that the *best case* running time is $\Omega(n^2)$.
- 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 $\Theta(g)$).

6 Algorithms matter!

Sort 10 million integers on

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

7 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:
  - $\lg^k n = (\lg n)^k$
  - $\lg \lg n = \lg(\lg n)$
  - $a^{\log_b c} = c^{\log_b a}$
  - $a^{\log_a b} = b$
  - $\log_a n = \frac{\log_b n}{\log_b a}$
  - $\lg b^n = n \lg b$
  - $\lg xy = \lg x + \lg y$
  - $\log_a b = \frac{1}{\log_b a}$