Analysis of Algorithms

Motivation

A good algorithm is
correct,
efficient, and
easy to implement.

- answering "how much time/space does this algorithm take?" and "can we do better?" requires a measure of the time/space requirements

Key Points

We want to **compare algorithms**, not programs.

- the elapsed time of a running program depends on many factors unrelated to the algorithm
  - speed of computer
  - computer architecture
  - choice of language, skill/cleverness of programmer, compiler optimizations
- implementing and debugging a program is time consuming
  - requires too many details

RAM Model of Computation

Assumptions –

- each simple operation takes exactly one time step
  - arithmetic, boolean, logical operations; `=`; `if`; subroutine calls
    - ⚠ `=`, `if` is the assignment or branch itself, not the evaluation of expressions or the execution of the body of a branch
    - ⚠ subroutine call is just the call and return, not the execution of the subroutine body
- each memory access takes exactly one time step

- expressions and blocks are not simple operations
- loops are not simple operations
  - composed of (many) simple operations
  - time required is the sum of the time required for each simple operation

Key Points

Those assumptions are actually false with respect to real computers.

Even though our analyses will be based on a model of computation that is **not** how real computers work, all is not lost –

- still meaningful
  - it is difficult to find a case where it gives misleading results
- simplifies analysis
  - allows for reasoning about algorithms in a language- and machine-independent manner

Key Points

We are actually more interested in **how quickly the running time of an algorithm increases as the size of the input increases** than in how long the algorithm will take on a particular input instance.

- still meaningful
  - a single input instance may not be all that informative anyway
  - any algorithm will is fine when the input is small – it's what happens for big inputs that matters
- simplifies analysis
  - means we don't need to count precisely – can focus on how the number of steps depends on aspects of the input
  - can consider (only) best and worst-case bounds
    - fewer cases to consider, and easier to work with an input instance with specific properties

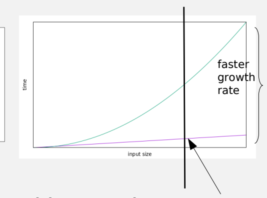
Key Points

We are actually even more interested in **categorizing algorithms into a few common classes** than determining specific growth rate functions.

- still meaningful
  - the differences within one class are far less than the differences between classes
- simplifies analysis
  - can drop constant factors and lower order terms (eliminating distracting bumps)
  - can analyze algorithm at a higher level of abstraction (pseudocode or even natural language description rather than code)

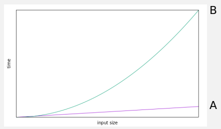
Understanding Limitations

Alice and Bob each implement different algorithms for solving a particular problem. When they run their programs, they find that the one with the slower growth rate takes longer. What could be going on?



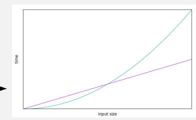
- be careful not to confuse growth rate with speed
  - the *speed* refers to the running time for a particular input
    - faster speed = takes less time
  - the *growth rate* refers to how quickly the running time increases
    - slower growth rate means the running time doesn't increase as quickly – the running time is smaller/shorter/faster for longer
  - the question is how an algorithm with a slower growth rate could take *more* time on an input than one with a faster growth rate

Understanding Limitations

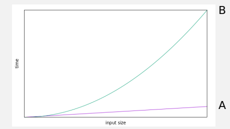


how can an algorithm A with a slower growth rate take *more* time on an input than algorithm B with a faster growth rate?

- n is small – constant factors and lower-order terms have a greater impact on running time for small *n*
- there could be different environments – language, programmer cleverness, compiler optimizations, computer speed, …
- "growth rate of algorithm" typically refers to the growth rate of the worst-case running time
  - input instance used may not be worst case for B



Understanding Limitations



how can an algorithm A with a slower growth rate could take *more* time on an input than algorithm B with a faster growth rate?

Or it might not been a fair test –

- different inputs used e.g. A's input was bigger
- (really) inefficient implementation of A
  - e.g. looping through whole array instead of only accessing one slot

---

- A takes more space, making it slower
  - each memory access is assumed to take one time step so the running time puts a limit on how much space A can use
  - A's computer could be pushed into swapping while B's is not
    - constant factors could mean that A's memory usage exceeds B's
    - A's computer could have less memory

Definitions

- O gives an *upper bound* on a function's growth rate
- Ω gives a *lower bound* on a function's growth rate
- Θ gives a *tight bound* on a function's growth rate

notation	meaning	definition
f(n) = O(g(n))	c g(n) is an upper bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≤ c g(n) for all n ≥ n_0
f(n) = Ω(g(n))	c g(n) is an lower bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≥ c g(n) for all n ≥ n_0
f(n) = Θ(g(n))	c_1 g(n) is an upper bound on f(n) c_2 g(n) is an lower bound on f(n)	there exists c_1 > 0, c_2 > 0, and n_0 > 0 such f(n) ≤ c_1 g(n) and f(n) ≥ c_2 g(n) for all n ≥ n_0

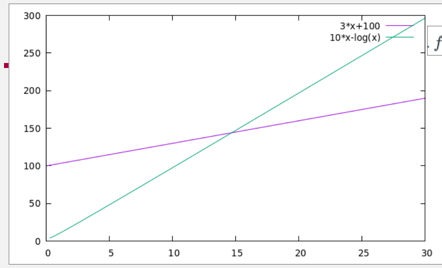
Understanding Definitions

For each of the following pairs of functions, indicate whether f = O(g), f = Ω(g), or f = Θ(g).

a. $f(n) = 3n + 100, g(n) = 10n - \log n$ [pairA]
b. $f(n) = (\log\ n)^2 + 5n\ \log\ n, g(n) = 2n$ [pairB]
c. $f(n) = 3n^2 + n^3, g(n) = 3^n - 5n^3$ [pairC]

O gives an *upper bound* on a function's growth rate
Ω gives a *lower bound* on a function's growth rate
Θ gives a *tight bound* on a function's growth rate

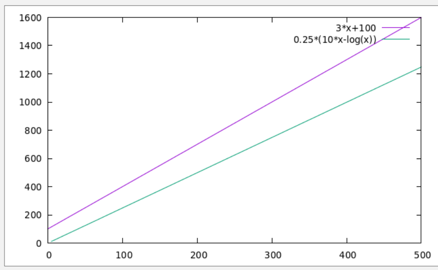
notation	meaning	definition
f(n) = O(g(n))	c g(n) is an upper bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≤ c g(n) for all n ≥ n_0
f(n) = Ω(g(n))	c g(n) is an lower bound on f(n)	there exists c > 0 and n_0 > 0 such that f(n) ≥ c g(n) for all n ≥ n_0
f(n) = Θ(g(n))	c_1 g(n) is an upper bound on f(n) c_2 g(n) is an lower bound on f(n)	there exists c_1 > 0, c_2 > 0, and n_0 > 0 such that f(n) ≤ c_1 g(n) and f(n) ≥ c_2 g(n) for all n ≥ n_0



$f(n) = 3n + 100, g(n) = 10n - \log n$

$3n+100 = O(10n - \log n)$

because
$3n+100 \leq c(10n - \log n)$
for $c = 1$ and $n > 15$

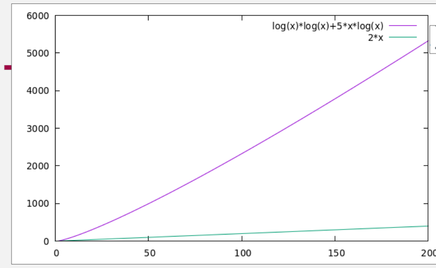


$3n+100 = \Omega(10n - \log n)$

because
$3n+100 \geq c(10n - \log n)$
for $c = 0.25$ and $n > 0$

thus
$3n+100 = \Theta(10n - \log n)$

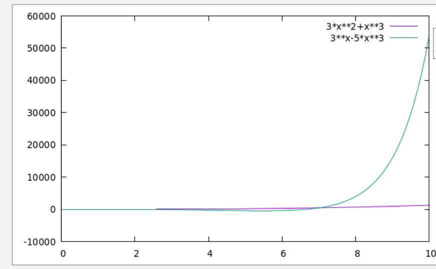
because
$3n+100 \leq c_1(10n - \log n)$ and
$3n+100 \geq c_2(10n - \log n)$ for
$c_1 = 1$, $c_2 = 0.25$, and $n > 15$



$f(n) = (\log n)^2 + 5n \log n, g(n) = 2n$

$(\log n)^2 + 5n \log n = \Omega(2n)$

because
$(\log n)^2 + 5n \log n \geq 2n$
for $c = 1$ and $n > 5$



$f(n) = 3n^2 + n^3, g(n) = 3^n - 5n^3$

$3n^2+n^3 = O(3^n-5n^3)$

because
$3n^2+n^3 \leq c(3^n-5n^3)$
for $c = 1$ and $n > 8$

O, Ω, Θ vs Best and Worst Cases

The big-Oh notation compares growth rates of functions – comparing shapes of curves.

- $f(n) = O(g(n))$ says that $f(n)$ grows no faster than $g(n)$
  - $g(n)$ is an upper bound on the growth rate
- $f(n) = \Omega(g(n))$ says that $f(n)$ grows no slower than $g(n)$
  - $g(n)$ is a lower bound on the growth rate
- $f(n) = \Theta(g(n))$ says that $f(n)$ grows at the same rate as $g(n)$
  - $g(n)$ is a tight bound on the growth rate

The best (or worst) case is the specific input instance that yields the fastest (or slowest) running time over all possible input instances of a given size – comparing the actual number of steps required.

- no input instance will take longer than the worst case for that size, or take less time than the best case for that size

Understanding Terminology and Concepts

If Alice proves that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ time on some inputs?

- **yes** – worst-case means no case is slower, but faster is possible

If Alice proves that an algorithm takes $O(n^2)$ worst-case time, it is possible that it takes $O(n)$ time on all inputs?

- **yes** – O is an upper bound, so $f(n) = O(n^2)$ says that $f(n)$ doesn't grow any faster than n^2, but it doesn't preclude it growing slower i.e. $n = O(n^2)$ though typically we want to give the tightest bound we can

If Alice proves that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ time on some inputs?

- **yes** – Θ means that the worst case won't actually turn out to be better than n^2, but the worst case is the slowest input of a given size and others (e.g. best case) may be better

O, Ω, Θ vs Best and Worst Cases

Saying that the worst-case behavior is **$O(n^2)$** means –

- some inputs could be O(n) because the worst case is the slowest instance for a given size
- all inputs could be O(n) because n grows no faster than n^2, though one generally tries to give the tightest O possible

Saying that the worst-case behavior is **$\Theta(n^2)$** means –

- some inputs could be O(n) because the worst case is the slowest instance for a given size
- not all inputs could be O(n) because then the worst case instances would also be O(n) and n does not grow at the same rate as n^2

O, Ω, or Θ?

- give as tight as bound as possible
- use Θ if you can
  - e.g. mergesort is Θ(n log n)
  - e.g. insertion sort is best case Θ(n) and worst case $\Theta(n^2)$
- can use O if best case running time grows more slowly than the worst case but you don't want to distinguish – only the worst case is important
  - e.g. insertion sort is $O(n^2)$
- can use Ω if worst case running time grows faster than the best case but you don't want to distinguish – only the best case is important
  - e.g. insertion sort is Ω(n)
- can use O (or Ω) if you can't establish a tight bound
  - you don’t know if the best case is better or if the worst case is worse