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
    - subroutine call is just the call and return, not the execution of the subroutine body
- loops and subroutines are not simple operations
  - composed of (many) simple operations
  - time required is the sum of the time required for each simple operation
- each memory access takes exactly one time step

Key Points

All three of these 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 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 do when the input is small – it's what happens for big inputs that matters
- simplifies analysis
  - 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 instance with specific properties

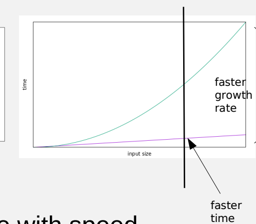
Key Points

We are 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)

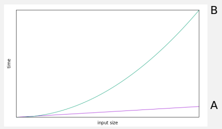
Understanding Limitations

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?



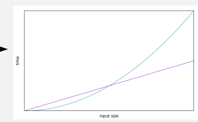
- be careful not to confuse growth rate with speed
  - the *speed* refers to the running time for a particular input
    - faster speed = less time
  - the *growth rate* refers to how quickly the running time increases
    - slower growth rate means the running time doesn't increase as quickly – the 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

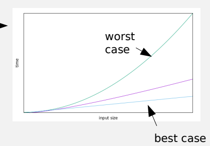
Understanding Limitations



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

- n is small – due constant factors or lower-order terms
- 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 may not be worst case for B
- different inputs e.g. different size





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