Algorithm Analysis

What is Algorithm Analysis?

- How to estimate the time required for an algorithm
- Techniques that drastically reduce the running time of an algorithm
- A mathemactical framwork that more rigorously describes the running time of an algorithm

Asymptotic Performance

- In this course, we are interested in *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - o Running time
    - o Memory/storage requirements
    - o Bandwidth/power requirements/logic gates/etc.

Algorithm Analysis Overview

- RAM model of computation
- Concept of input size
- Measuring complexity
  - Best-case, average-case, worst-case
- Asymptotic analysis
  - Asymptotic notation

Assume the RAM Model

- RAM model represents a “generic” implementation of the algorithm
- Each “simple” operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are not simple operations, but depend upon the size of the data and the contents of a subroutine. We do not want “sort” to be a single step operation.
- Each memory access takes exactly 1 step.

Assume the RAM Model – contd.

- Has one processor (uniprocessor - RAM)
- Executes one instruction at a time (no concurrent operations)
- Each instruction takes "unit time“
- Has fixed-size operands, (constant word size)
- All memory equally expensive to access, and
- Has fixed size storage (RAM and disk).

Algorithm Analysis Overview

- RAM model of computation
- Concept of input size
- Measuring complexity
  - Best-case, average-case, worst-case
- Asymptotic analysis
  - Asymptotic notation

Input Size

- Time and space complexity
  - This is generally a function of the input size
    - o E.g., sorting, multiplication
  - How we characterize input size depends:
    - o Sorting: number of input items
    - o Multiplication: total number of bits
    - o Graph algorithms: number of nodes & edges
    - o Etc

Input Size – contd.

- In general, larger input instances require more resources to process correctly
- We standardize by defining a notion of size for an input instance
- Examples
  - What is the size of a sorting input instance?
  - What is the size of an “Odd number” input instance?

Algorithm Analysis Overview

- RAM model of computation
- Concept of input size
- Measuring complexity
  - Best-case, average-case, worst-case
- Asymptotic analysis
  - Asymptotic notation

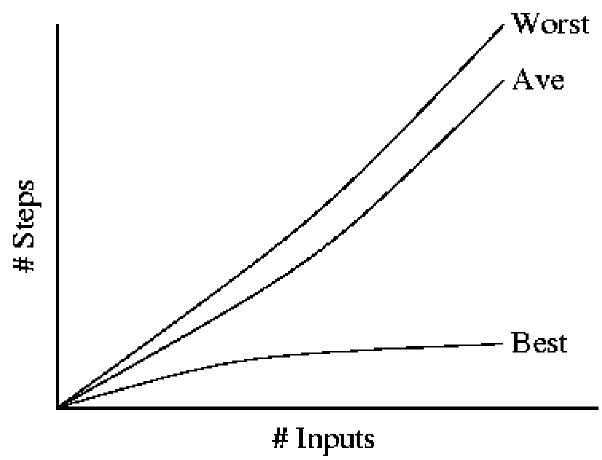
Measuring Complexity

- The running time of an algorithm is the function defined by the number of steps (or amount of memory) required to solve input instances of size n
  - F(1) = 3
  - F(2) = 5
  - F(3) = 7
  - …
  - F(n) = 2n+1
- Problem: Inputs of the same size may require different numbers of steps to solve

3 Different Analyses

- The *worst case running time* of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.
- The *best case running time* of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.
- The *average-case running time* of an algorithm is the function defined by an average number of steps taken on any instance of size n.
- Which of these is the best to use?

Best, Worst, and Average Case



3 Different Analyses – contd.

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee

- Worst case running time is often comparable to average case running time (see next graph)
  - Counterexamples to above point:
    - o Quicksort
    - o simplex method for linear programming

3 Different Analyses – contd.

Worst case Analysis:

- Provides guarantee that is independent of any assumptions about the input
- Typically much simpler to compute as we do not need to “average” performance on many inputs
  - Instead, we need to find and understand an input that causes worst case performance
- Often reasonably close to average case running time
- The standard analysis performed

3 Different Analyses – contd.

- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs
- Average Case Analysis - Drawbacks
  - Based on a probability distribution of input instances
    - The distribution may not be appropriate
    - Provides little consolation if we have a worst-case input
  - More complicated to compute than worst case running time

Algorithm Analysis Overview

- RAM model of computation
- Concept of input size
- Measuring complexity
  - Best-case, average-case, worst-case
- Asymptotic analysis
  - Asymptotic notation

Motivation for Asymptotic Analysis

- An *exact computation* of worst-case running time can be difficult
  - Function may have many terms:
    - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75$
- An *exact computation* of worst-case running time is unnecessary
  - Remember that we are already approximating running time by using RAM model

Asymptotic Analysis

- We focus on the infinite set of large n ignoring small values of n
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymptotic Notation

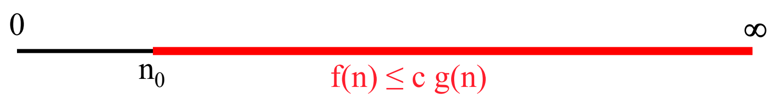
- Our first task is to define asymptotic notation more formally and completely

“Big Oh” Notation

- $O(g(n)) =$

  $\{f(n)$: there exist positive constants c and n_0 such that $\forall\ n \geq n_0,\ 0 \leq f(n) \leq c\ g(n)\ \}$

  - What are the roles of the two constants?
    - n_0:
    - c:



Set Notation Comment

- $O(g(n))$ is a set of functions.
- However, we will use one-way equalities like

  $n = O(n^2)$
- This really means that function n belongs to the set of functions $O(n^2)$
- Incorrect notation: $O(n^2) = n$
- Analogy
  - “A dog is an animal” but not “an animal is a dog”

Three Common Sets

f(n) = O(g(n)) means c × g(n) is an *Upper Bound* on f(n)

f(n) = Ω(g(n)) means c × g(n) is a *Lower Bound* on f(n)

f(n) = Θ(g(n)) means c_1 × g(n) is an *Upper Bound* on f(n)
and c_2 × g(n) is a *Lower Bound* on f(n)

These bounds hold for all inputs beyond some threshold n_0.

Asymptotic Notation – contd.

- Upper Bound Notation:
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
  - Formally, $O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$
- Big O fact:
  - A polynomial of degree k is $O(n^k)$

Asymptotic Notation – contd.

- Asymptotic lower bound:

  f(n) is *$\Omega(g(n))$* if $\exists$ positive constants *c* and n_0 such that $0 \le c \cdot g(n) \le f(n) \quad \forall\, n \ge n_0$
- Asymptotic tight bound:

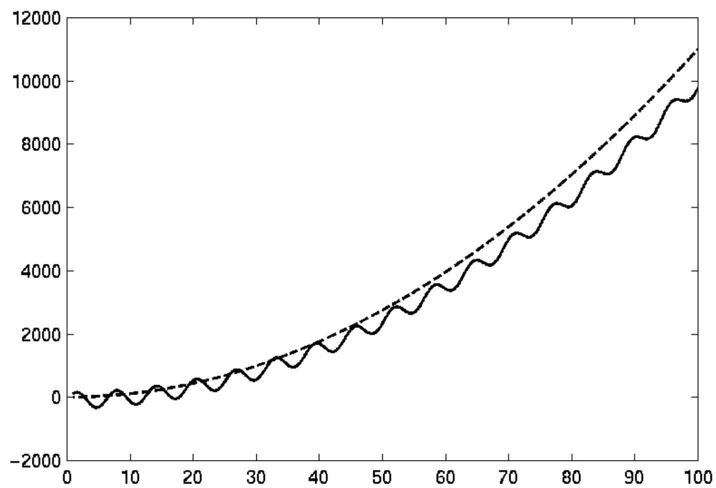
  A function f(n) is $\Theta(g(n))$ if $\exists$ positive constants *c1*, *c2*, and *n0* such that

  $$c1\ g(n) \le f(n) \le c2\ g(n)\ \forall\ n \ge n0$$

Asymptotic Notation – contd.

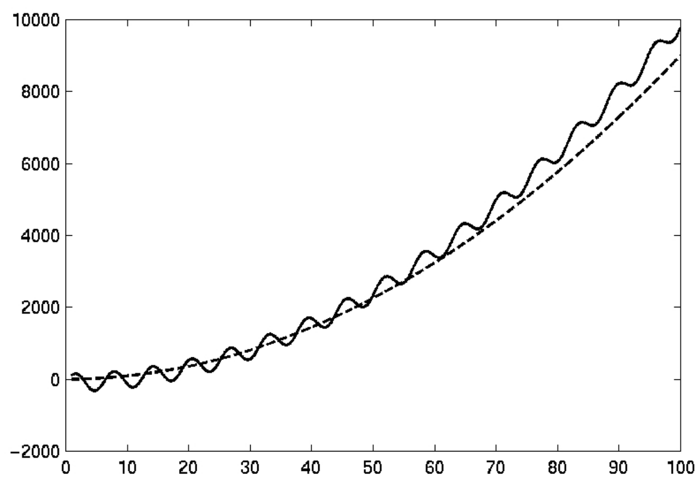
- Theorem
  - f(n) is $\Theta(g(n))$ iff f(n) is both $O(g(n))$ and $\Omega(g(n))$

O(g(n))



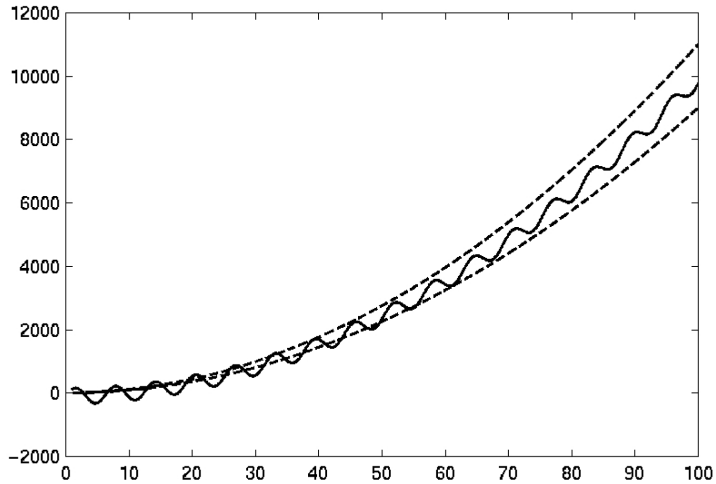
12000
10000
8000
6000
4000
2000
0
−2000
0 10 20 30 40 50 60 70 80 90 100

Ω(g(n))

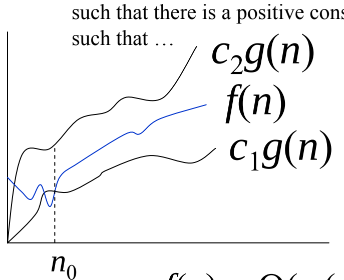


10000
8000
6000
4000
2000
0
−2000
0 10 20 30 40 50 60 70 80 90 100

$\Theta(g(n))$

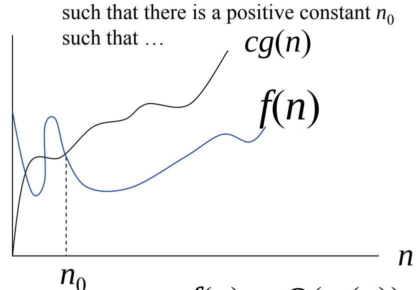


There exist positive constants c_1 and c_2 such that there is a positive constant n_0 such that …



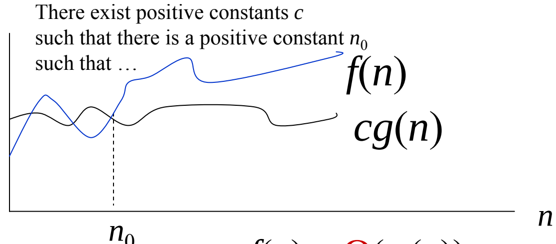
$$f(n) = \Theta(\,g(n))$$

There exist positive constants c such that there is a positive constant n_0 such that …



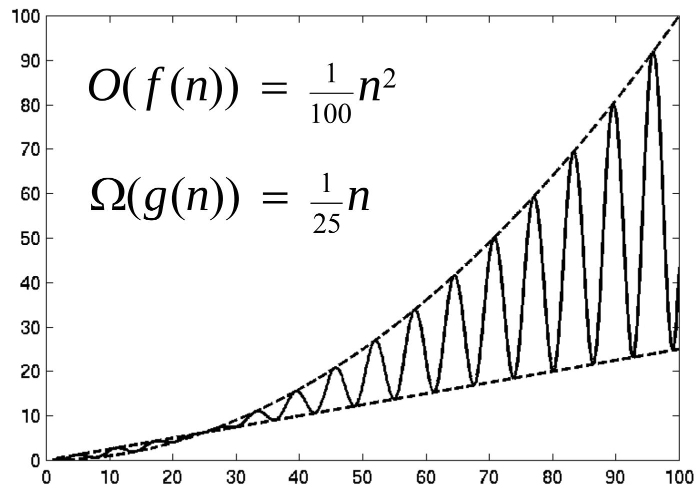
$$f(n) = O(\,g(n))$$

There exist positive constants c such that there is a positive constant n_0 such that …



$$f(n) = \Omega(\,g(n))$$

O(f(n)) and Ω(g(n))



Other Asymptotic Notations

- A function f(n) is o(g(n)) if ∃ positive constants c and n_0 such that
  $f(n) < c\, g(n)\ \forall\ n \geq n_0$
- A function f(n) is ω(g(n)) if ∃ positive constants c and n_0 such that
  $c\, g(n) < f(n)\ \forall\ n \geq n_0$
- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

Review: Asymptotic Performance

- *Asymptotic performance*: How does algorithm behave as the problem size gets very large?
  - o Running time
  - o Memory/storage requirements
  - Remember that we use the RAM model:
    - o All memory equally expensive to access
    - o No concurrent operations
    - o All reasonable instructions take unit time
      - Except, of course, function calls
    - o Constant word size
      - Unless we are explicitly manipulating bits

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

A concrete example

The following table shows how long it would take to perform T(n) steps on a computer that does 1 billion steps/second. Note that a microsecond is a millionth of a second and a millisecond is a thousandth of a second.

N	$T(n) = n$	$T(n) = n\lg n$	$T(n) = n^2$	$T(n) = n^3$	$Tn = 2^n$
5	0.005 microsec	0.01 microsec	0.03 microsec	0.13 microsec	0.03 microsec
10	0.01 microsec	0.03 microsec	0.1 microsec	1 microsec	1 microsec
20	0.02 microsec	0.09 microsec	0.4 microsec	8 microsec	1 millisec
50	0.05 microsec	0.28 microsec	2.5 microsec	125 microsec	13 days
100	0.1 microsec	0.66 microsec	10 microsec	1 millisec	4×10^{13} years

Notice that when n >= 50, the computation time for $T(n) = 2^n$ has started to become too large to be practical. This is most certainly true when n >= 100. Even if we were to increase the speed of the machine a million-fold, 2^n for n = 100 would be 40,000,000 years, a bit longer than you might want to wait for an answer.