Complexity Analysis II

CSE 373
Data Structures & Algorithms
Linda Shapiro
Spring 2013

Today's Outline

- **Announcements**
  - Assignment #1, due Friday, April 12 at 11pm
  - Assignment #2, posted later this week, due Friday April 19 at BEGINNING of lecture
- **Algorithm Analysis**
  - Big-Oh
  - Analyzing code

Ignoring constant factors

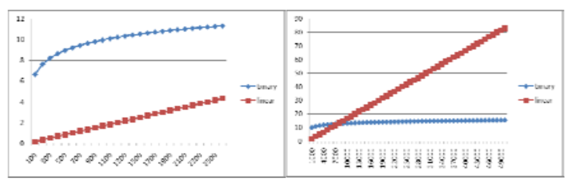
- So binary search is $O(\log n)$ and linear search is $O(n)$
  - But which is faster?
- Could depend on constant factors:
  - How *many* assignments, additions, etc. for each *n*
    - *E.g.* $T(n) = 5{,}000{,}000n$ vs. $T(n) = 5n^2$
  - And could depend on size of *n (if n is small then constant additive factors could be more important)*
    - *E.g.* $T(n) = 5{,}000{,}000 + \log n$ vs. $T(n) = 10 + n$
- **But** there exists ***some*** n_0 such that for all $n > n_0$ binary search wins
- Some plots will give us intuition…

Linear Search vs. Binary Search

Let's try to "help" linear search:

- Run it on a computer 100x as fast (say 2010 model vs. 1990)
- Use a new compiler/language that is 3x as fast
- Be a clever programmer to eliminate half the work
- So doing each iteration is 600x as fast as in binary search

For small n, linear search is faster! But eventually binary search wins.



Asymptotic notation

The formal definition of Big-O amounts to saying:

1. Eliminate low-order terms
2. Eliminate coefficients

Examples:

- $4n + 5$
- $0.5n \log n + 2n + 7$
- $n^3 + 2^n + 3n$
- $n \log (10n^2)$

Examples

True or false?

1. 4+3n is O(n)
2. n+2logn is O(logn)
3. logn+2 is O(1)
4. n^{50} is $O(1.1^n)$

Examples

True or false?

1. $4+3n$ is $O(n)$ — True
2. $n+2\log n$ is $O(\log n)$ — False
3. $\log n+2$ is $O(1)$ — False
4. n^{50} is $O(1.1^n)$ — True

n	n^{50}	1.1^n
1	1	1.1
2	1.1E15	1.2
3	7.2E23	1.3
4	1.3E30	1.5
5	8.9E34	1.6
6	8.1E38	1.8
7	1.8E42	1.9
8	1.4E45	2.1
9	5.6E47	2.4
10	1.0E50	2.6
....		
BUT		
1M	1.0E450	1.0E500 (over)

Big-Oh relates functions

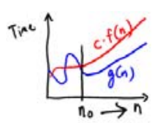
We use O on a function $f(n)$ (for example n^2) to mean *the set of functions* with *asymptotic behavior less than or equal to* $f(n)$

So $(3n^2+17)$ is in $O(n^2)$
$3n^2+17$ and n^2 have the same asymptotic behavior

Confusingly, we also say/write:
- **$(3n^2+17)$ is $O(n^2)$** ★
- **$(3n^2+17) \in O(n^2)$**
- **$(3n^2+17) = O(n^2)$**
- **$(3n^2+17)$ is order n^2** ★

But we would never say $O(n^2) = (3n^2+17)$

Formal Big-Oh (again)



> Definition: **$g(n)$** is O(**$f(n)$**) iff there exist positive constants c and n_0 such that
>
> **$g(n) \le c\ f(n)$** for all $n \ge n_0$

To show **$g(n)$** is O(**$f(n)$**), pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

- Example: Let **$g(n)$** $= 3n^2+17$ and **$f(n)$** $= n^2$
  - $c = 5$ and $n_0 = 10$ is more than good enough
  - since $3n^2+17 \le 5n^2$ for $n \ge 10$

This is "less than or equal to"
- So $3n^2+17$ is also $O(n^5)$ and $O(2^n)$ etc.
- BUT NOBODY SAYS THAT WHEN DOING COMPLEXITY ANAL.

Using the definition of Big-Oh (Example 1)

> Def'n:
> $g(n)$ is **O($f(n)$)** iff there exist positive constants c and n_0 s.t.
> $g(n) \le c\ f(n)$ for all $n \ge n_0$

Given: **g(n)** = 1000n

1. Let f(n) = n

Prove: **g(n)** is O(**f(n)**)

$1000n \le 1000n$

$c = 1000$, $n_0 = 1$ (or anything)

2. Let f(n) = n^2

Prove: **g(n)** is in O(**f(n)**)

$1000n \le 1000n^2$ for $n \ge 1$

$c = 1000$, $n_0 = 1$

Also try $c = 1$, $n_0 = 1000$

But anyone doing a complexity analysis would do 1 and not 2, ie.
Choose the smallest common function that works.

Using the definition of Big-Oh (Example 2)

> Def'n:
> $g(n)$ is in **O($f(n)$)** iff there exist positive constants c and n_0 s.t.
> $g(n) \le c\ f(n)$ for all $n \ge n_0$

Given: **g(n)** = $3n^2 + 4n$ & **f(n)** = n^2

Prove: **g(n)** is in O(**f(n)**)

- A valid proof is to find valid **c** & **n_0**
- $3n^2 + 4n$
- $\le 3n^2 + 4n^2 = 7n^2$ for $n \ge 1$
- Are there other combinations of constants that work?

Using the definition of Big-Oh (Example 3)

> Def'n:
> $g(n)$ is in **O($f(n)$)** iff there exist positive constants c and n_0 s.t.
> $g(n) \le c\ f(n)$ for all $n \ge n_0$

Given: **g(n)** = n^4 & **f(n)** = 2^n,

Prove: **g(n)** is O(**f(n)**)

- A valid proof is to find valid **c** & **n_0**
- One possible answer: **n_0** = 20, and **c** = 1

- But this f(n) is an upper bound on g(n).
- The function **f(n)** = 2^n has *exponential growth*.
- The function **g(n)** = n^4 has *polynomial growth* (of degree 4).
- The exponential function is for large n going to be *much* larger.

- Try some comparisons. 2^{50} = 1.1E15 50^4 = 6,250,000

What's with the c?

- To capture this notion of similar asymptotic behavior, we allow a constant multiplier (called c)
- Consider:
  - g(n) = 7n+5
  - f(n) = n
- These have the same asymptotic behavior (linear), so g(n) is O(f(n)) even though g(n) is always larger
- There is no positive n_0 such that g(n) ≤ f(n) for all n ≥ n_0
- The 'c' in the definition allows for that: $g(n) \leq c\ f(n)$ for all $n \geq n_0$
- To prove g(n) is O(f(n)), have c = 12, n_0 = 1

Big Oh: Common Categories

From fastest to slowest:

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\log n)$	logarithmic ($\log_k n$, $\log n^2$ is $O(\log n)$)
$O(n)$	linear
$O(n \log n)$	"$n \log n$"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is an constant)
$O(k^n)$	exponential (where k is any constant > 1)

Usage note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"

We tend to use the smallest common function that satisfies the def.

More Definitions

- **Upper bound**: $O($ f(n) $)$ is the set of all functions asymptotically less than or equal to f(n)
  - g(n) is $O($ f(n) $)$ if there exist positive constants c and n_0 such that
    $g(n) \leq c\ f(n)$ for all $n \geq n_0$
- **Lower bound**: $\Omega($ f(n) $)$ is the set of all functions asymptotically greater than or equal to f(n)
  - g(n) is $\Omega($ f(n) $)$ if there exist positive constants c and n_0 such that
    $g(n) \geq c\ f(n)$ for all $n \geq n_0$
- **Tight bound**: $\theta($ f(n) $)$ is the set of all functions asymptotically equal to f(n)
  - g(n) is $\theta($ f(n) $)$ if <u>both</u>: g(n) is $O($ f(n) $)$ AND g(n) is $\Omega($ f(n) $)$

Even More Definitions…

o(f(n)) is the set of all functions asymptotically less than f(n)

- g(n) is o(f(n)) if for any positive constant c, there exists a positive constant n_0 such that
  $g(n) < c\ f(n)$ for all $n \geq n_0$

$\omega($ f(n) $)$ is the set of all functions asymptotically greater than f(n)

- g(n) is ω(f(n)) if for any positive constant c, there exists a positive constant n_0 such that
  $g(n) > c\ f(n)$ for all $n \geq n_0$

Intuitively

Asymptotic Notation	Mathematics Relation
O	≤
Ω	≥
Θ	=
o	<
ω	>

Types of Analysis

Two <u>orthogonal</u> axes:

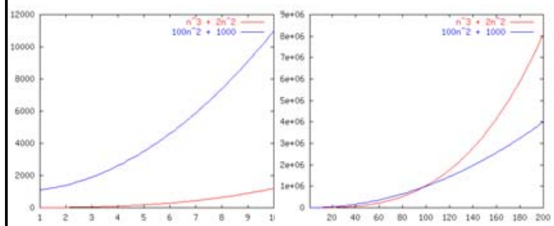
- bound flavor (usually we talk about upper or tight)
  - upper bound (O, o)
  - lower bound (Ω, ω)
  - asymptotically tight (Θ)
- analysis case (usually we talk about worst)
  - worst case (adversary)
  - average case
  - best case
  - "amortized" (not in this class) uses the idea that certain costly operations cannot occur frequently enough to cause trouble.

Which Function Grows Faster?

$n^3 + 2n^2$ vs. $100n^2 + 1000$

Which Function Grows Faster?

$n^3 + 2n^2$ vs. $100n^2 + 1000$

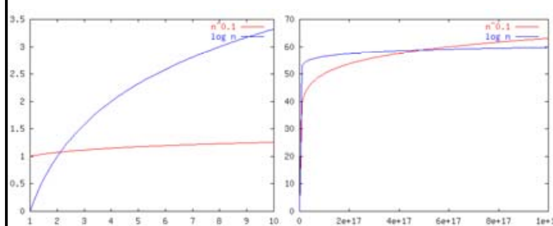


Which Function Grows Faster?

$n^{0.1}$ vs. $\log n$

Which Function Grows Faster?

$n^{0.1}$ vs. $\log n$

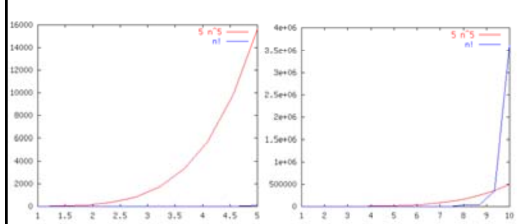


Which Function Grows Faster?

$5n^5$ vs. $n!$

Which Function Grows Faster?

$5n^5$ vs. $n!$



Nested Loops

```
one nested loop
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

$O(n^2)$

```
two consecutive loops
for i = 1 to n do
 sum = sum + 1
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

$n + n^2 <= 2n^2$

$O(n^2)$

More Nested Loops

```
for i = 1 to n do
  for j = 1 to n do
    if (cond) {
            do_stuff(sum)
      } else {
            for k  = 1 to n*n
                  sum += 1
```

For if-else statement, we assume for the worst case that max complexity branch will be taken.

What happens here?

Big-Oh Caveats

- Asymptotic complexity (Big-Oh) focuses on behavior for **_large n_** and is independent of any computer / coding trick
  - But you can "abuse" it to be misled about trade-offs
  - Example: $n^{1/10}$ vs. `log` n
    - Asymptotically $n^{1/10}$ grows more quickly
    - But the "cross-over" point is around $5 * 10^{17}$
    - So if you have input size less than 2^{58}, you prefer $n^{1/10}$
- Comparing O() for **_small n_** values can be misleading
  - Quicksort: O(nlogn) (expected)
  - Insertion Sort: $O(n^2)$ (expected)
  - Yet in reality Insertion Sort is faster for small n's

Addendum: Timing vs. Big-Oh?

- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of n
  - Be able to mathematically prove things about performance
- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards
  - We will do some timing in HW 2