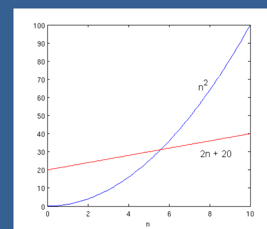
COMPSCI 130
Design and Analysis of Algorithms

2 – Asymptotic Analysis

Comparing Functions

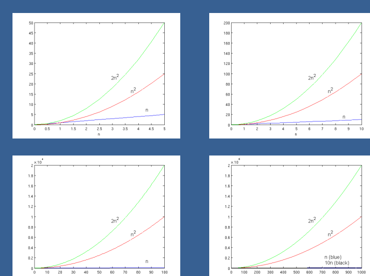


2n + 20 vs n^2

Big O

- Defined:
  f(n) = O(g(n)) means f(n) ≤ c g(n) for some c, all n
- The *asymptotic complexity* of f is upper bounded by g

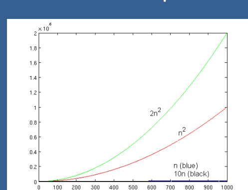
Asymptotic Complexity Illustrated



n, n^2, and $2n^2$
max(n) = 5, 10, 100, and 1000
n = O(n^2); also $2n^2$ = O(n^2)

Asymptotic Dominance

- At large scales, constant scaling of O(n^2) curves is unchanged
- O(n) curves vanish in comparison



Upper and Lower Bounds

- Formally, O(g) defines an *upper bound* on complexity
- Other notation exists for
  - lower boundedness (big Ω);
    f = Ω(g) ⇒ g = O(f)
    ⇒ ∃c: f ≥ cg
  - lower/upper boundedness (big Θ)
    f = Ω(g) ⇒ f = O(g), f = Ω(g)
    ⇒ ∃c,d: dg ≤ f ≤ cg

Common Usage

- Informally, we often use O to mean O, Ω, or Θ
  - Depends on context
  - Generically implies either O or Θ
- In literature, meaning is usually made explicit:
  - f is upper bounded by g
  - f has worst-case complexity of O(g)
  - f has best-case complexity of O(g) (i.e. f = Ω(g))
  - note: f possibly unknown!

Notation Propagation

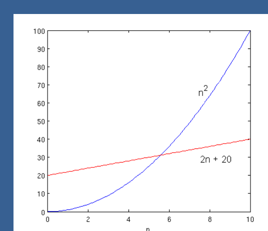
- Even more notation has been devised to designate whether or not a bound is tight
  - bound is tight ⇔ f = O(g) and g = O(f)
  - little o, ω versus big O, Ω
- Nobody uses this outside textbooks
- Instead, tightness is made explicit

Simplifying

- Use the simplest expression
- E.g., lower order polynomials can be ignored because they are completely dominated by higher order polynomials
  - $O(n)$ not $O(n + c)$
  - $O(n^2)$ not $O(n^2 + n + c)$

Example

$n^2 + 2n + 20 = O(n^2)$



Growth of Various Functions

n	1	10	100	1000	10^6
log(n)	0	1	2	3	6
√n	1	3.16	10	31.62	1000
n log(n)	1	10	200	3000	6×10^6
n^2	1	100	10^4	10^6	10^{12}
2^n	2	1024	$\sim10^{30}$	$\sim10^{300}$	Forget it!

Computation Time

Assuming 2×10^{10} operations/second

(approximately the FP performance of a modern Intel desktop chip)

n	10	50	100	10^6	10^9	10^{12}
n	< 1 ns	< 1 ns	< 1 ns	50 µs	50 ms	50 s
n log(n)	< 1 ns	< 1 ns	1 ns	300 ms	450 ms	10 min
n^2	<1 ns	125 ns	500 ns	50 s	1.6 years	1.6 million years
2^n	50 ns	16 hours	1.5 trillion years			

↑ Datasets of size 10^6 and above are now commonplace!

↑ # of unique URLs seen by Google indexer

Rules of Thumb

- Drop constant factors
- Drop lower order polynomials
- Any polynomials dominate any logarithms
- Exponential functions (x^n) dominate polynomials
- x^n dominates y^n for $x>y$

A Word About Logarithms

- Conventionally, log means $\log_{10}$
  - For this course, log usually means $\log_2$
  - The distinction is unimportant in big O because $\log_2(x) = \log_k(x)/\log_k(2)$
- log(n) is
  - the power to which you raise 2 to get n
  - the number of times you divide n by 2 to get to 1
  - max height of a binary tree with n nodes
  - number of bits required to represent n

Some Useful Identities

$\log 2^n = n$	$2^{\log n} = n$
$\log x^y = y \log x$	$(2^x)^y = 2^{(xy)}$
$\log xy = \log x + \log y$	$2^x 2^y = 2^{(x+y)}$

Example

$f = n^2 + 2n + 20$

$g = n^2$

Divide f / g

$= 1 + 2/n + 20/n^2$

< 23 for all n

$f < 23\,g \Rightarrow f = O(g)$

By inspection, we also have $g = O(f)$

Example

2^n vs 3^n

Claim: $2^n = O(3^n)$, but not the reverse.

Proof:

By inspection, $2^n < c\,3^n$ is true for all n when $c = 1$ so $2^n = O(3^n)$.

Now, suppose $2^n = O(3^n)$. This implies $3^n < c\,2^n$ for some c and all n.

Then

$3^n/2^n < c$ for all n

but it is easy to see that

$\lim_{n\to\infty} 3^n/2^n = \infty$,

so we can always choose an n that contradicts our assumption. □

Example

Claim: log n is $O(\sqrt[k]{n})$ for all k, but not the reverse

Proof:

To start, raise both sides to the power of k.

We want c such that:

$\log^k n \le c^k n$

$\log^k n/n \le c^k$

$\ln^k n/n \le (c \ln 2)^k$

Let's find the maxima of the left hand side. Taking the derivative and setting to zero:

$k \ln^{k-1} n / n^2 - \ln^k n / n^2 = 0$

$k \ln^{k-1} n = \ln^k n$

$k = \ln n$

$e^k = n$

A second derivative test will verify that this is a maximum and not a minimum.

Example (con't)

Claim: log n is $O(\sqrt[k]{n})$ for all k, but not the reverse

Recall we want c such that:

$$\ln^k n/n \quad \le (c \ln 2)^k$$

and we found that the LHS is maximized at $n = e^k$.

[Rather than doing all of this again in reverse, note that we can easily show that the LHS has no minimum; this implies that we cannot find a constant c such that the inequality is reversed.]

The existence of a maximum completes the proof; plugging back in, we can also find the correct constant, c:

$$\begin{aligned} \ln^k e^k/e^k &= (c \ln 2)^k \\ k^k/e^k &= (c \ln 2)^k \\ k/(e \ln 2) &= c \end{aligned}$$

When Constants Matter

- If crossover point is at a point with high enough costs, it may pay to mix algorithms
  - e.g., use a simple n^2 sort inside recursive n log n sort when n gets very small
- If two algorithms have the same asymptotic costs, the one with the lower constants wins

Other Useful Formulas

$n! \approx \sqrt{2\pi n}\,(n/e)^n$ (Stirling's approximation)

$1 + 2 + \ldots + n = n(n+1)/2$ (Gauss' formula)

$1 + x + x^2 + x^3 + \ldots + x^n = (x^{n+1} - 1)/(x-1)$

$1 + ½ + ¼ + \ldots = 2$

$1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n = \ln n + O(1)$

Big O in Recurrence

- Be careful to not simplify O's in recurrence – it does not reduce (constants matter!)
- Example

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ T(n) &= O(1) + O(2) + \ldots + \\ &= O(1) + O(1) + \ldots \\ &= O(1) \quad \text{wrong!} \end{aligned}$$

A better notation might be $T(n) = O(1 + 2 + \ldots + n)$