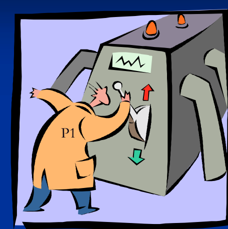
Big-O
Analyzing Algorithms Asymptotically

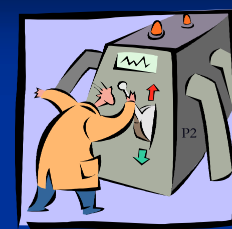
CS 226

5 February 2002

Noah Smith (`nasmith@cs`)

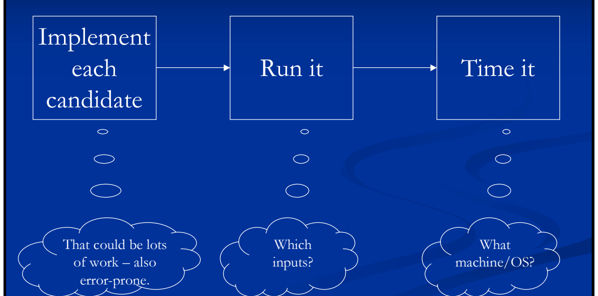
Comparing Algorithms





Should we use Program 1 or Program 2?
Is Program 1 "fast"? "Fast enough"?

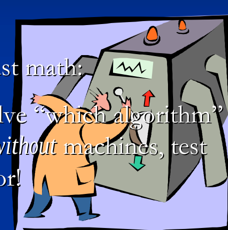
You and Igor: the empirical approach



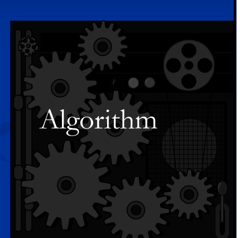
Toward an analytic approach …

Today is just math:

How to solve "which algorithm" problems *without* machines, test data, or Igor!



The Big Picture



How long does it take for the algorithm to finish?

Primitives

- Primitive operations
  - x = 4 — assignment
  - ... x + 5 ... — arithmetic
  - if (x < y) ... — comparison
  - x[4] — index an array
  - *x — dereference (C)
  - x.foo( ) — calling a method
- Others
  - new/malloc — memory usage

How many foos?

```
for (j = 1; j <= N; ++j) {
        foo( );
}
```

$$\sum_{j=1}^{N} 1 = N$$

How many foos?

```
for (j = 1; j <= N; ++j) {
        for (k = 1; k <= M; ++k) {
                foo( );
        }
}
```

$$\sum_{j=1}^{N} \sum_{k=1}^{M} 1 = NM$$

How many foos?

```
for (j = 1; j <= N; ++j) {
        for (k = 1; k <= j; ++k) {
                foo( );
        }
}
```

$$\sum_{j=1}^{N} \sum_{k=1}^{j} 1 = \sum_{j=1}^{N} j = \frac{N (N + 1)}{2}$$

How many foos?

```
for (j = 0; j < N; ++j) {
        for (k = 0; k < j; ++k) {
                foo( );
        }
}
```

N(N + 1)/2

```
for (j = 0; j < N; ++j) {
        for (k = 0; k < M; ++k) {
                foo( );
        }
}
```

NM

How many foos?

```
void foo(int N) {
        if(N <= 2)
                return;
        foo(N / 2);
}
```

$T(0) = T(1) = T(2) = 1$

$T(n) = 1 + T(n/2)$ if $n > 2$

$$\begin{aligned} T(n) &= 1 + (1 + T(n/4)) \\ &= 2 + T(n/4) \\ &= 2 + (1 + T(n/8)) \\ &= 3 + T(n/8) \\ &= 3 + (1 + T(n/16)) \\ &= 4 + T(n/16) \\ &\ldots \\ &\approx \log_2 n \end{aligned}$$

The trick

$$\underbrace{an^k + bn^{k-1} + \ldots + yn + z}$$

$$\underbrace{an^k}$$

$$n^k$$

Big O

Definition: Let f and g be functions mapping $\mathbf{N}$ to $\mathbf{R}$. We say that $f(n)$ is $O(g(n))$ if there exist

$$c \in \mathbf{R},\ c > 0$$

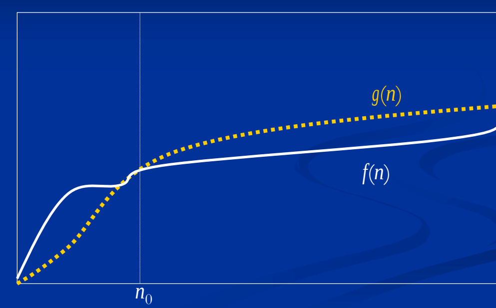
and

$$n_0 \in \mathbf{N},\ n_0 \geq 1$$

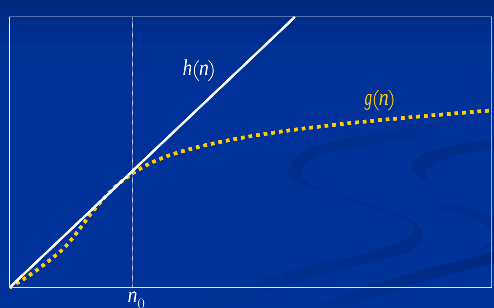
such that

$$f(n) \leq cg(n) \text{ for all } n \in \mathbf{N},\ n \geq n_0$$

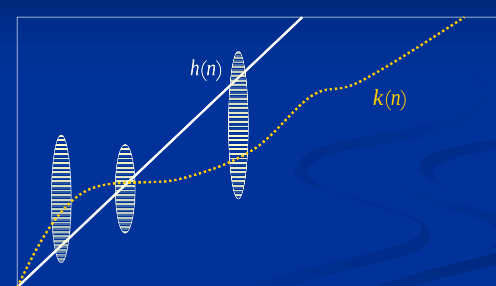
Example 1



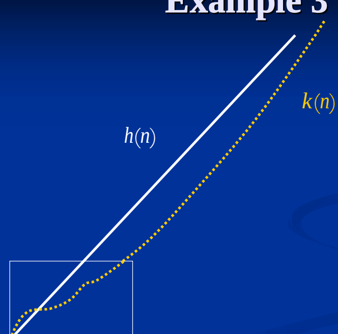
Example 2



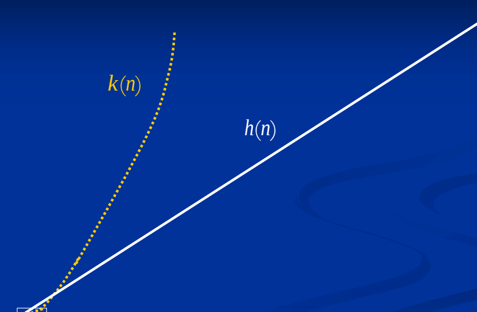
Example 3



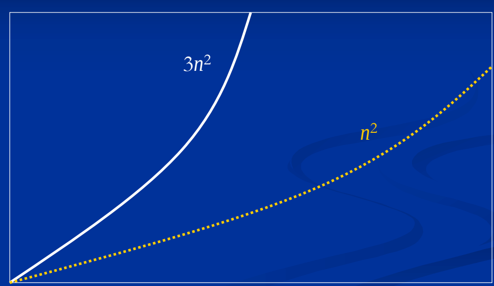
Example 3



Example 3



Example 4



Some complexity classes …

Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Polynomial	$O(n^p)$
Exponential	$O(a^n)$

Don't be confused …

- We typically say,

  $f(n)$ is $O(g(n))$ or $f(n) = O(g(n))$
- But $O(g(n))$ is really a set of functions.
- It might be more clear to say,

  $f(n) \in O(g(n))$
- But I don't make the rules.
- Crystal clear: "$f(n)$ is **order** $(g(n))$"

Intuitively …

- To say $f(n)$ is $O(g(n))$ is to say that

  $f(n)$ is "less than or equal to" $g(n)$
- We also have (G&T pp. 118-120):

$\Theta(g(n))$	Big Theta	"equal to"
$\Omega(g(n))$	Big Omega	"greater than or equal to"
$o(g(n))$	Little o	"strictly less than"
$\omega(g(n))$	Little omega	"strictly greater than"

Big-Omega and Big-Theta

Ω is just like O except that $f(n) \geq cg(n)$;

$f(n)$ is $O(g(n)) \Leftrightarrow g(n)$ is $\Omega(f(n))$

Θ is both O and Ω (and the constants need not match);

$f(n)$ is $O(g(n)) \wedge f(n)$ is $\Omega(g(n)) \Leftrightarrow f(n)$ is $\Theta(g(n))$

little o

Definition: Let f and g be functions mapping $\mathbf{N}$ to $\mathbf{R}$. We say that $f(n)$ is $o(g(n))$ if

for any $c \in \mathbf{R}$, $c > 0$

there exists

$n_0 \in \mathbf{N}$, $n_0 > 0$

such that

$f(n) \leq cg(n)$ for all $n \in \mathbf{N}$, $n \geq n_0$

(little omega, ω, is the same but with $\geq$)

Multiple variables

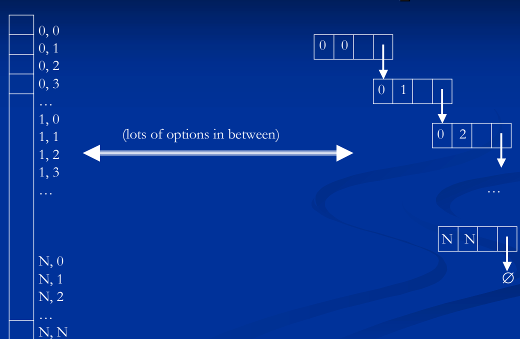
```
for(j = 1; j <= N; ++j)
   for(k = 1; k <= N; ++k)
        for(l = 1; l <= M; ++l)
                foo();
for(j = 1; j <= N; ++j)
   for(k = 1; k <= M; ++k)
        for(l = 1; l <= M; ++l)
                foo();
```

$O(N^2M + NM^2)$

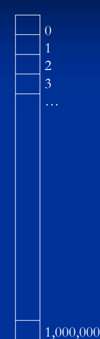
Multiple primitives

```
for(j = 1; j <= N; ++j){
   sum += A[j];
   for(k = 1; k <= M; ++k) {
        sum2 += B[j][k];
        C[j][k] = B[j][k] * A[j] + 1;
        for(l = 1; l <= k; ++l)
                B[j][k] -= B[j][l];
   }
}
```

Tradeoffs: an example



Another example



- I have a set of integers between 0 and 1,000,000.
- I need to store them, and I want $O(1)$ lookup, insertion, and deletion.
- Constant time and constant space, right?

Big-O and Deceit

- Beware huge coefficients
- Beware key lower order terms
- Beware when n is "small"

Does it matter?

Let n = 1,000, and 1 ms / operation.

	n = 1000, 1 ms/op	max n in one day
n	1 second	86,400,000
$n \log_2 n$	10 seconds	3,943,234
n^2	17 minutes	9,295
n^3	12 days	442
n^4	32 years	96
n^{10}	3.17×10^{19} years	6
2^n	1.07×10^{301} years	26

Worst, best, and average

Gideon is a fast runner

- … up hills.
- … down hills.
- … on flat ground.

Gideon is the fastest swimmer

- … on the JHU team.
- … in molasses.
- … in our research lab.
- … in 5-yard race.
- … on Tuesdays.
- … in an average race.

What's average?

- Strictly speaking, average (mean) is relative to some probability distribution.

$$\text{mean}(X) = \sum_x \Pr(x) \times x$$

- Unless you have some notion of a probability distribution over test cases, it's hard to talk about average requirements.

Now you know …

- How to analyze the run-time (or space requirements) of a piece of pseudo-code.
- Some new uses for Greek letters.
- Why the order of an algorithm matters.
- How to avoid some pitfalls.