CSE332: Data Abstractions

Lecture 2: Math Review; Algorithm Analysis

Ruth Anderson

Winter 2011

Announcements

- Project 1 coming soon (phase A, due next week)
- Homework 1 coming soon (due next Friday)

- **Room changes** – for section (and possibly lecture)
  - If not registered for course yet, go to any section tomorrow:
  - AC: 1230-120 in MGH 241 (no change)
  - AA: 130-220 in MGH 241
  - AB: 230-320 in SAV 138

- Bring info sheet to lecture on Friday
- Fill out catalyst survey by Thursday evening

Today

- Review math essential to algorithm analysis
  - Proof by induction
  - Bit patterns
  - Powers of 2
  - Exponents and logarithms

- Begin analyzing algorithms
  - Using asymptotic analysis (continue next time)

Mathematical induction

Suppose *P(n)* is some predicate (involving integer *n*)

- Example: $n \geq n/2 + 1$ (for all n ≥ 2)

To prove *P(n)* for all integers *n* ≥ *c*, it suffices to prove

1. *P(c)* – called the "basis" or "base case"
2. If *P(k)* then *P(k+1)* – called the "induction step" or "inductive case"

Why we will care:

To show an algorithm is correct or has a certain running time *no matter how big a data structure or input value is*

(Our "*n*" will be the data structure or input size.)

Example

P(n) = " the sum of the first *n* powers of 2 (starting at 2^0) is 2^n-1 "

Theorem: *P(n)* holds for all *n* ≥ 1

1	= 2-1
1 + 2	= 4-1
1 + 2 + 4	= 8-1

So far so good…

Example

Theorem: *P(n)* holds for all *n* ≥ 1

Proof: By induction on *n*

- Base case, *n*=1: $2^0 = 1 = 2^1 - 1$
- Inductive case:
  - Inductive hypothesis: Assume the sum of the first *k* powers of 2 is 2^k-1
  - Show, given the hypothesis, that the sum of the first (*k*+1) powers of 2 is 2^{k+1}-1

  From our inductive hypothesis we know:

  $$1+2+4+...+2^{k-1} = 2^k - 1$$

  Add the next power of 2 to both sides…

  $$1+2+4+...+2^{k-1}+2^k = 2^k - 1 + 2^k$$

  We have what we want on the left; massage the right a bit

  $$1+2+4+...+2^{k-1}+2^k = 2(2^k) - 1 = 2^{k+1} - 1$$

Note for homework

Proofs by induction will come up a fair amount on the homework

When doing them, be sure to state each part clearly:

- What you're trying to prove
- The base case
- The inductive case
- The inductive hypothesis
  - In many inductive proofs, you'll prove the inductive case by just starting with your inductive hypothesis, and playing with it a bit, as shown above

N bits can represent how many things?

# Bits	Patterns	# of patterns
1		
2		

Powers of 2

- A bit is 0 or 1
- A sequence of *n* bits can represent 2^n distinct things
  - For example, the numbers 0 through 2^n-1
- 2^{10} is 1024 ("about a thousand", kilo in CSE speak)
- 2^{20} is "about a million", mega in CSE speak
- 2^{30} is "about a billion", giga in CSE speak

Java: an **`int`** is 32 bits and signed, so "max int" is "about 2 billion"
a **`long`** is 64 bits and signed, so "max long" is 2^{63}-1

Therefore…

Could give a unique id to…

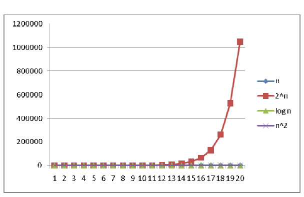
- Every person in the U.S. with 29 bits
- Every person in the world with 33 bits
- Every person to have ever lived with 38 bits (estimate)
- Every atom in the universe with 250-300 bits

So if a password is 128 bits long and randomly generated, do you think you could guess it?

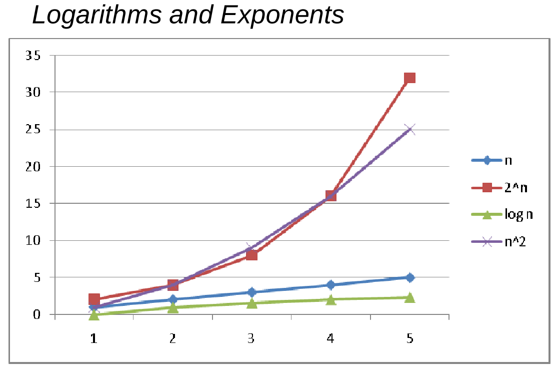
Logarithms and Exponents

- Since so much is binary in CS, **`log`** almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

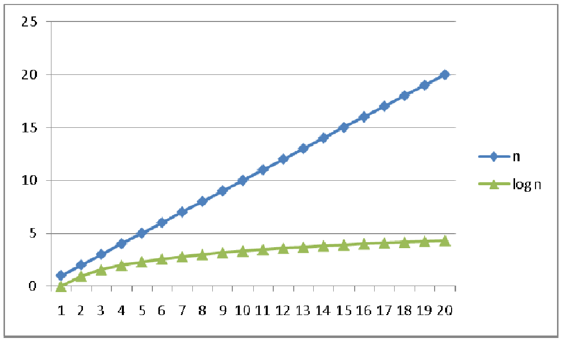
See Excel file for plot data – play with it!



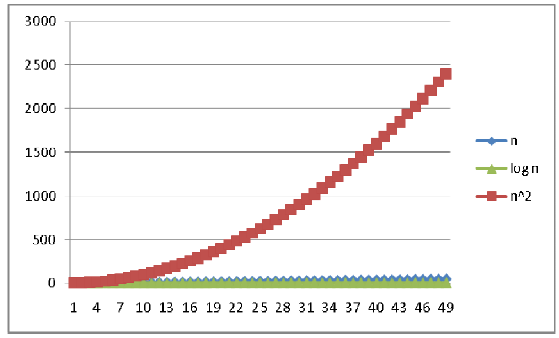
Logarithms and Exponents



Logarithms and Exponents



Logarithms and Exponents



Properties of logarithms

- **`log(A*B) = log A + log B`**
  - So **`log(N^k)= k log N`**
- **`log(A/B) = log A – log B`**
- **`x =`** $\log_2 2^x$
- **`log(log x)`** is written **`log log x`**
  - Grows as slowly as 2^{2^y} grows fast
  - Ex: $$\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$$
- **`(log x)(log x)`** is written **`log²x`**
  - It is greater than **`log x`** for all **`x > 2`**

Log base doesn't matter (much)

"Any base *B* log is equivalent to base 2 log within a constant factor"
- And we are about to stop worrying about constant factors!
- In particular, **$\log_2 x = 3.22 \log_{10} x$**
- In general, we can convert log bases via a constant multiplier
- Say, to convert from base B to base A:

  $\log_B x = (\log_A x) / (\log_A B)$

Algorithm Analysis

As the "size" of an algorithm's input grows (integer, length of array, size of queue, etc.):
- How much longer does the algorithm take (time)
- How much more memory does the algorithm need (space)

Because the curves we saw are so different, we often only care about "which curve we are like"

Separate issue: Algorithm *correctness* – does it produce the right answer for all inputs
- Usually more important, naturally

Example

- What does this pseudocode return?

```
x := 0;
for i=1 to N do
  for j=1 to i do
     x := x + 3;
return x;
```

- Correctness: For any N ≥ 0, it returns…

Example

- What does this pseudocode return?

```
x := 0;
for i=1 to N do
  for j=1 to i do
     x := x + 3;
return x;
```

- Correctness: For any N ≥ 0, it returns 3N(N+1)/2
- Proof: By induction on *n*
  - *P(n)* = after outer for-loop executes *n* times, **x** holds 3n(n+1)/2
  - Base: n=0, returns 0
  - Inductive: From *P(k)*, **x** holds 3k(k+1)/2 after *k* iterations. Next iteration adds 3(k+1), for total of 3k(k+1)/2 + 3(k+1) = (3k(k+1) + 6(k+1))/2 = (k+1)(3k+6)/2 = 3(k+1)(k+2)/2

Example

- How long does this pseudocode run?

```
x := 0;
for i=1 to N do
  for j=1 to i do
     x := x + 3;
return x;
```

- Running time: For any N ≥ 0,
  - Assignments, additions, returns take "1 unit time"
  - Loops take the sum of the time for their iterations
- So: 2 + 2*(number of times inner loop runs)
  - And how many times is that…

Example

- How long does this pseudocode run?

```
x := 0;
for i=1 to N do
  for j=1 to i do
     x := x + 3;
return x;
```


Example

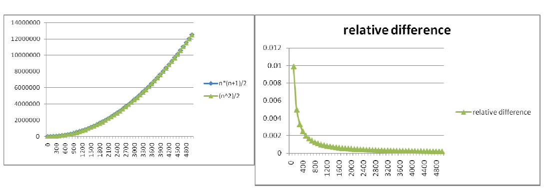
- How long does this pseudocode run?

```
x := 0;
for i=1 to N do
  for j=1 to i do
     x := x + 3;
return x;
```

- The total number of loop iterations is N*(N+1)/2
  - This is a very common loop structure, worth memorizing
  - This is *proportional to* N^2 , and we say *O*(N^2), "big-Oh of"
    - For large enough N, the N and constant terms are irrelevant, as are the first assignment and return
    - See plot… N*(N+1)/2 vs. just N^2/2

Lower-order terms don't matter

N*(N+1)/2 vs. just N^2/2



Big-O: Common Names

O(1)	constant (same as *O*(*k*) for constant *k*)
O(**`log`** *n*)	logarithmic
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)

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