CSE332: Data Abstractions

Lecture 2: Math Review; Algorithm Analysis

Tyler Robison (covering for James Forgarty)

Winter 2012

Lecture Overview

Largely reviewing math relevant to this course

- Proof by induction
- Logarithms
- Start on Algorithm Analysis
  - How much time & space an algorithm takes to run

Proof via mathematical induction

Suppose *P(n)* is some rule involving n

- Example: $n \geq n/2 + 1$, for all integers $n \geq 2$

To prove *P(n)* for all integers $n \geq c$, 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:

Use to show that 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 the next power of 2 minus 1"

Theorem: *P(n)* holds for all integers $n \geq 1$

1=2-1

1+2=4-1

1+2+4=8-1

So far so good…

Example

Theorem: *P(n)* holds for all $n \geq 1$

Proof: By induction on *n*

- Base case, *n*=1: $2^0 = 1 = 2^1 - 1$
- Inductive case: If it holds for k, then it holds for k+1
  - 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:

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

  Add the next power of 2 to both sides…

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^k - 1 + 2^k$$

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

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

Another Example

For all $n \geq 1$

$1+2+3+\ldots+(n-1)+n = n(n+1)/2$

Ex: $1+2+3+4+5+6 = 6*7/2 = 21$

Proof: By induction on *n*

- Base case, $n=1$: $1=1*(1+1)/2$
- Inductive case:
  - Inductive hypothesis: Assume the sum of the first *k* integers (from 1 up) equals k(k+1)/2
  - Show, given the hypothesis, that it holds true for the next integer (k+1)

  From our inductive hypothesis we know:

  $1+2+3+\ldots+k = k(k+1)/2$

  Add k+1 to both sides…

  $1+2+3+\ldots+k +(k+1)= k(k+1)/2 + (k+1)$

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

  $1+2+3+\ldots+k +(k+1)= (k(k+1) + 2(k+1))/2 = (k^2+k+2k+2)/2 = (k+1)(k+2)/2$

Note for homework

Proofs by induction may come up in 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

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…

We could give a unique id to…

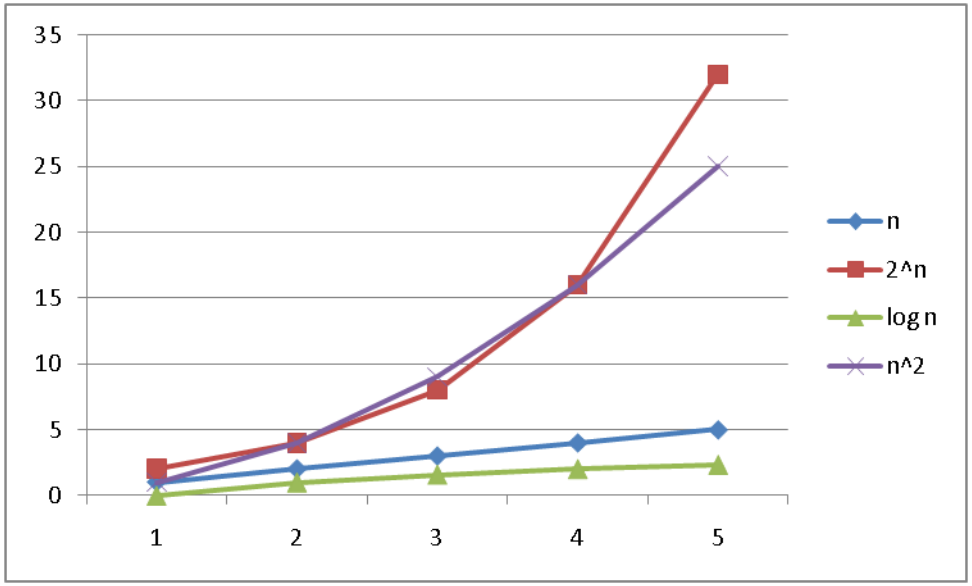
- Every person in this room with 7 bits
- Every person in the U.S. with 29 bits
- Every person in the world with 33 bits
- Every person to have ever lived with 38 bits (estimate)
- Every atom in the universe with 250-300 bits

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

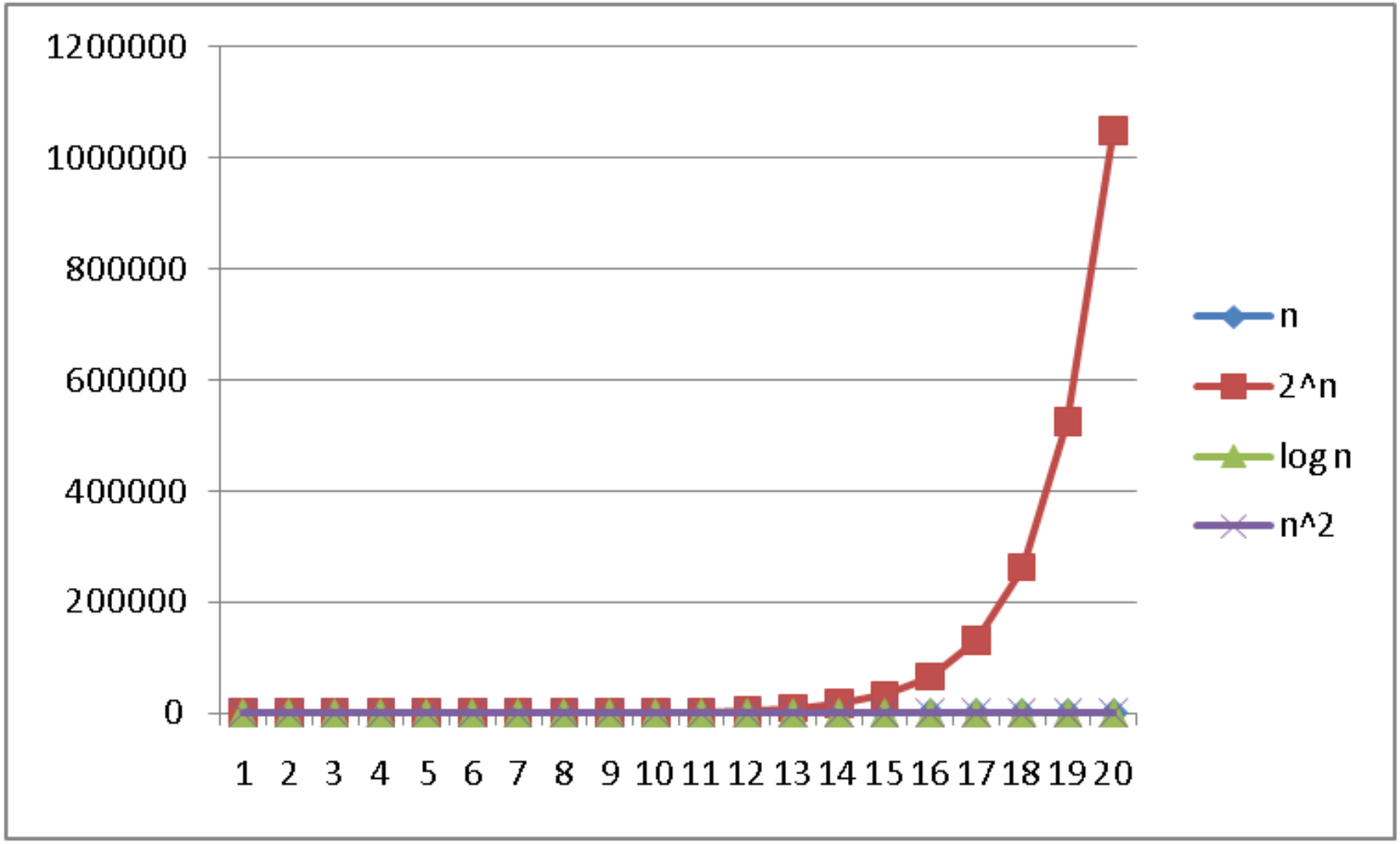
Logarithms and Exponents

- Since so much is binary in CS, **log** almost always means $\log_2$
- Definition: $\log_2 x = y$ iff $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"

Just as exponents grow *very* quickly, logarithms grow *very* slowly



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**
- $\log_2 2^x$**=x**
- **log(log x)** is written **log log x**
  - Grows as slowly as $2^{2^{\wedge}x}$ 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 **$\log^2$x**
  - 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:

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

$$\log_{10} x = (\log_2 x) / (\log_2 10)$$

Algorithm Analysis

As the "size" of an algorithm's input grows
(length of array to sort, size of queue to search, etc.):

- How much longer does the algorithm take (time)
- How much more memory does the algorithm need (space)

We are generally concerned about approximate runtimes

- Whether T(n)=3n+2 or T(n)=n/4+8, we say it runs in linear time
- Common categories:
  - Constant: T(n)=1
  - Linear: T(n)=n
  - Logarithmic: T(n)=logn
  - Exponential: $T(n)=2^n$

Example

- First, what does this pseudocode return?

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

- For any n ≥ 0, it returns 3n(n+1)/2
- Why?
  - Consider, how many times does the inner loop run?
  - For i=1, it runs once
  - For i=2, it runs twice
  - Etc.
  - 1+2+3+…+n = n(n+1)/2
  - x gets raised by 3 each time

Example

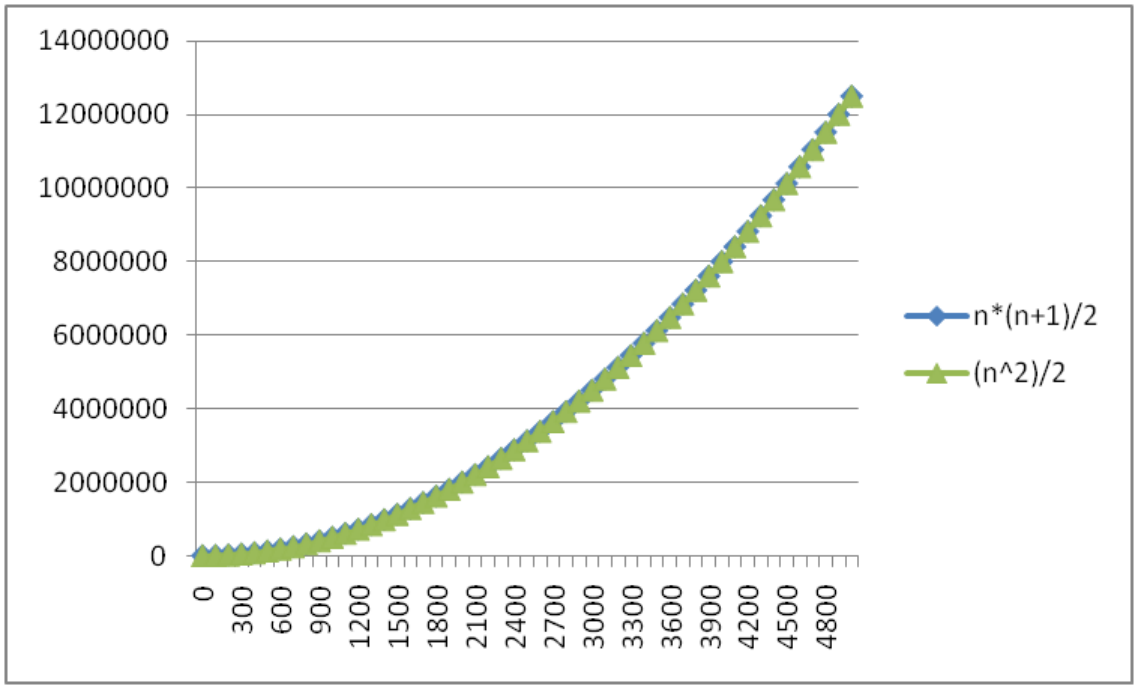
- How long does this pseudocode run?

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

- Find running time in terms of n, for any $n \geq 0$
  - Assignments, additions, simple comparisons, etc. take "1 unit time"
    - Constant time
  - Loops take the sum of the time for their iterations
- Say, (roughly) 2+5*(number of times inner loop runs)
  - Inner loop runs n(n+1)/2 times
  - So $O(n^2)$ time

Lower-order terms don't matter for our purposes

n*(n+1)/2 vs. just $n^2/2$



We'll discuss why on Monday

In essence, we're mostly concerned with behavior as n approaches infinity

Big Oh (also written Big-O)

- Big Oh is used for comparing asymptotic behavior of functions
- We'll get into the definition later, but for now:
  - 'f(n) is O(g(n))' roughly means
    - The function f(n) is at least as small as g(n) as they go toward infinity
    - Think of it as a ≤ for functions
  - BUT: Big Oh ignores constant factors
    - n+10 is O(n); we drop out the '+10'
    - 5n is O(n); we drop out the 'x5'
    - The following is NOT true though: n^2 is O(n)
  - Also note that 'f(n) is O(g(n))' gives an upper bound for f(n)
    - n is O(n^2)
    - 5 is O(n)

Big Oh: Common Categories

From fastest to slowest

$O(1)$	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)

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

- A savings account accrues interest exponentially (k=1.01?)