CS221: Algorithms and Data Structures Lecture #1 Complexity Theory and Asymptotic Analysis

Steve Wolfman

2011W2

Today's Outline

- Programming Project #1 and Forming Teams
- Brief Proof Reminder
- Asymptotic Analysis, Briefly
- Silicon Downs and the SD Cheat Sheet
- Asymptotic Analysis, Proofs and Programs
- Examples and Exercises

Prog Proj #1 & Teams

Proof by...

- Counterexample
  - show an example which does not fit with the theorem
  - QED (the theorem is ***dis***proven)
- Contradiction
  - assume the opposite of the theorem
  - derive a contradiction
  - QED (the theorem is proven)
- Induction
  - prove for one or more base cases (e.g., n = 1)
  - assume for one or more anonymous values (e.g., k)
  - prove for the next value (e.g., k + 1)
  - QED

Example Proof by Induction

A number is divisible by 3 iff the sum of its digits is divisible by three

Example Proof by Induction (Worked)

"A number is divisible by 3 iff the sum of its digits is divisible by three."

First, some definitions:

Consider a positive integer x to be made up of its n digits: $x_1x_2x_3..x_n$.

For convenience, let's say SD(x) = $\sum_{i=1}^{n} x_i$

Example Proof by Induction (Worked)

"A number is divisible by 3 iff the sum of its digits is divisible by three."

There are ***many*** ways to solve this, here's one.

We'll prove a somewhat *stronger* property, that for a non-negative integer x with any positive integral number of digits *n*, SD(x) ≡ x (mod 3).

(That is, the remainder when we divide SD(x) by 3 is the same as the remainder when we divide x by 3.)

Example Proof by Induction (INSIGHT FIRST!)

How do we break a problem down?

We often break sums down by "pulling off" the first or last term:

$$SD(x) = \sum_{i=1}^{n} x_i = x_n + \sum_{i=1}^{n-1} x_i$$

We can "peel off" the rightmost or leftmost digit. Peeling off the rightmost is like dividing by 10 (dumping the remainder). Peeling off the leftmost is harder to describe.

Let's try peeling off the rightmost digit!

Example Proof by Induction (Worked)

(With our insight, we clearly only need one base case.)

Base case (where n = 1):

Consider any number x with one digit (0-9).

$SD(x) = \sum_{i=1}^{1} x_i = x_1 = x.$

So, it's trivially true that SD(x) ≡ x (mod 3).

Example Proof by Induction (Worked)

WLOG, let n be an arbitrary integer greater than 0.

Induction Hypothesis: Assume for any non-negative integer x with *n* digits: SD(x) ≡ x (mod 3).

Inductive step:

Consider an arbitrary number y with *n* + 1 digits.

We can think of y as being made up of its digits: $y_1y_2\ldots y_ny_{n+1}$. Clearly, $y_1y_2\ldots y_n$ (which we'll call z) is itself an *n* digit number; so, the induction hypothesis applies:

SD(z) ≡ z (mod 3).

Example Proof by Induction (Worked)

Inductive step continued:

Now, note that $y = z*10 + y_{n+1}$.

So: $y \equiv (z*10 + y_{n+1})$ (mod 3)

$\equiv (z*9 + z + y_{n+1})$ (mod 3)

z*9 is divisible by 3; so, it has no impact on the remainder of the quantity when divided by 3:

$\equiv (z + y_{n+1})$ (mod 3)

Example Proof by Induction (Worked)

Inductive step continued:

By the IH, we know SD(z) ≡ z (mod 3).

So:

$\equiv (z + y_{n+1})$ (mod 3)

$\equiv (SD(z) + y_{n+1})$ (mod 3)

$\equiv (y_1 + y_2 + \ldots + y_n + y_{n+1})$ (mod 3)

$\equiv SD(y)$ (mod 3)

QED!

Can I really sub SD(z) for z inside the mod, even though they're only equal *mod 3*? Yes… they only differ by a multiple of 3, which cannot affect the "mod 3" sum.

Proof by Induction Pattern Reminder

First, find a way to break down the theorem interpreted for some (large-ish, arbitrary) value **k** in terms of the theorem interpreted for smaller values.

Next, prove any base cases needed to ensure that the "breakdown" done over and over eventually hits a base case.

Then, assume the theorem works for all the "smaller values" you needed in the breakdown (as long as **k** is larger than your base cases).

Finally, build up from those assumptions to the **k** case.

Induction Pattern to Prove P(n)

First, figure out how P(n) breaks down in terms of P(*something(s) smaller*).

P(n) is *theorem goes here*.

Theorem: P(n) is true for all n ≥ *smallest case*.

Proof: We proceed by induction on n.

Base Case(s) (P(·) is true for *whichever cases you need to "bottom out"*)**:**
Prove each base case via your other techniques.

Inductive Step (if P(·) is true for *the "something smaller" case(s)*, then P(n) is true, *for all n not covered by the base case (usually: greater than the largest base case)*):

WLOG, let n be an arbitrary integer *not covered by the base case (usually: greater than the largest base case)*.
Assume P(·) is true for *the "something smaller" case(s)*. (The Induction Hyothesis (IH).)
Break P(n) down in terms of the smaller case(s).
The smaller cases are true, by the IH.
Build back up to show that P(n) is true.

This completes our induction proof. QED

(almost always includes the smallest case)

Today's Outline

- Programming Project #1 and Forming Teams
- Brief Proof Reminder
- Asymptotic Analysis, Briefly
- Silicon Downs and the SD Cheat Sheet
- Asymptotic Analysis, Proofs and Programs
- Examples and Exercises

A Task to Solve and Analyze

Find a student's name in a class given her student ID

Analysis of Algorithms

- Analysis of an algorithm gives insight into how long the program runs and how much memory it uses
  - time complexity
  - space complexity
- Analysis can provide insight into alternative algorithms
- Input size is indicated by a number *n* (sometimes there are multiple inputs)
- Running time is a function of *n* ($\mathbf{Z}^0 \rightarrow \mathbf{R}^0$) such as
  $T(n) = 4n + 5$
  $T(n) = 0.5\ n \log n - 2n + 7$
  $T(n) = 2^n + n^3 + 3n$
- But...

Asymptotic Analysis Hacks

- Eliminate low order terms
  - $4n + 5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
  - $2^n + n^3 + 3n \Rightarrow 2^n$
- Eliminate coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

Rates of Growth

- Suppose a computer executes 10^{12} ops per second:

n =	10	100	1,000	10,000	10^{12}
n	10^{-11}s	10^{-10}s	10^{-9}s	10^{-8}s	1s
n lg n	10^{-11}s	10^{-9}s	10^{-8}s	10^{-7}s	40s
n^2	10^{-10}s	10^{-8}s	10^{-6}s	10^{-4}s	10^{12}s
n^3	10^{-9}s	10^{-6}s	10^{-3}s	1s	10^{24}s
2^n	10^{-9}s	10^{18}s	10^{289}s		

10^4s = 2.8 hrs 10^{18}s = 30 billion years

Order Notation

- $T(n) \in O(f(n))$ if there are constants c and n_0 such that $T(n) \le c\ f(n)$ for all $n \ge n_0$

Order Notation

- $T(n) \in O(f(n))$ if there are constants c and n_0 such that $T(n) \le c\ f(n)$ for all $n \ge n_0$
- $T(n) \in \Omega(f(n))$ if $f(n) \in O(T(n))$
- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

Order Notation

- $T(n) \in O(f(n))$ if there are constants c and n_0 such that $T(n) \le c\ f(n)$ for all $n \ge n_0$
- $T(n) \in \Omega(f(n))$ if $f(n) \in O(T(n))$
- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

How would you prove one of these?

Prove O(..) by finding a good c and n_0 (often helps in scratch work to "solve for" c, keeping notes of constraints on n).

Then, assume $n \ge n_0$ and show that $T(n) \le c\ f(n)$.

Prove Ω and Θ by breaking down in terms of O.

Examples

$10{,}000\ n^2 + 25\ n \in \Theta(n^2)$

$10^{-10}\ n^2 \in \Theta(n^2)$

$n \log n \in O(n^2)$

$n \log n \in \Omega(n)$

$n^3 + 4 \in O(n^4)$ but not $\Theta(n^4)$

$n^3 + 4 \in \Omega(n^2)$ but not $\Theta(n^2)$

Today's Outline

- Programming Project #1 and Forming Teams
- Brief Proof Reminder
- Asymptotic Analysis, Briefly
- Silicon Downs and the SD Cheat Sheet
- Asymptotic Analysis, Proofs and Programs
- Examples and Exercises

Silicon Downs

Post #1	Post #2
$n^3 + 2n^2$	$100n^2 + 1000$
$n^{0.1}$	$\log n$
$n + 100n^{0.1}$	$2n + 10 \log n$
$5n^5$	$n!$
$n^{-15}2^n/100$	$1000n^{15}$
$8^{2 \lg n}$	$3n^7 + 7n$
mn^3	$2^m n$

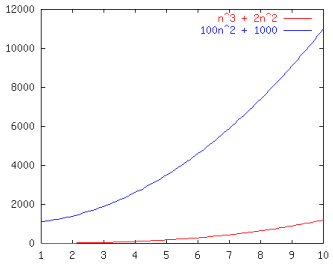
For each race, which "horse" is "faster". Note that faster means *smaller*, not larger!

a. Left
b. Right
c. Tied
d. It depends
e. I am opposed to algorithm racing.

Race I

a. Left
b. Right
c. Tied
d. It depends

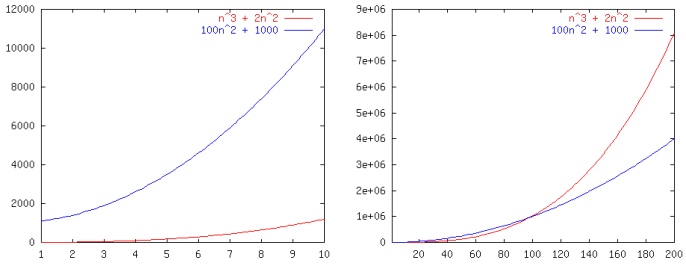
$n^3 + 2n^2$ vs. **$100n^2 + 1000$**



Race I

a. Left
b. Right
c. Tied
d. It depends

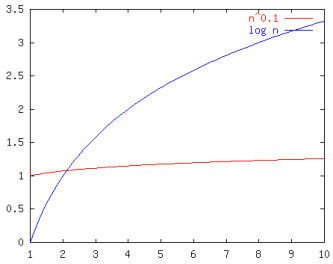
$n^3 + 2n^2$ vs. **$100n^2 + 1000$**



Race II

a. Left
b. Right
c. Tied
d. It depends

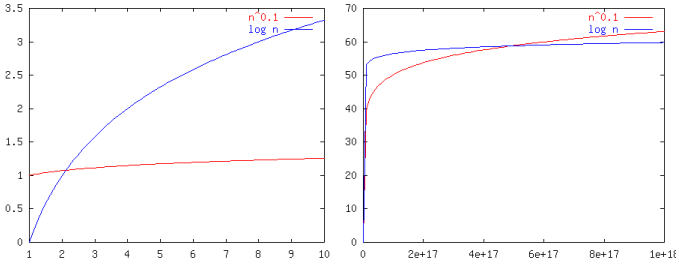
$n^{0.1}$ vs. **$\log n$**



Race II

a. Left
b. Right
c. Tied
d. It depends

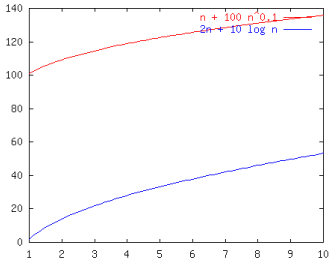
$n^{0.1}$ vs. **$\log n$**



Race III

a. Left
b. Right
c. Tied
d. It depends

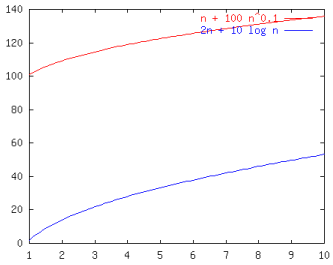
$n + 100n^{0.1}$ vs. **$2n + 10 \log n$**

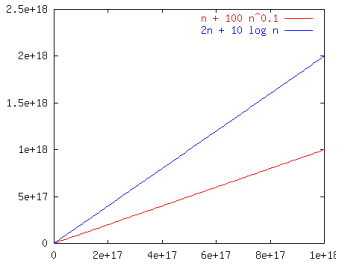


Race III

a. Left
b. Right
c. Tied
d. It depends

$n + 100n^{0.1}$ vs. $2n + 10 \log n$

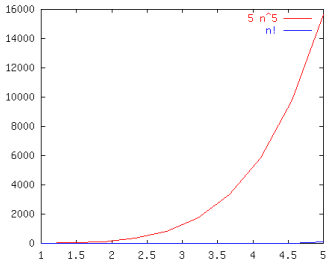




Race IV

a. Left
b. Right
c. Tied
d. It depends

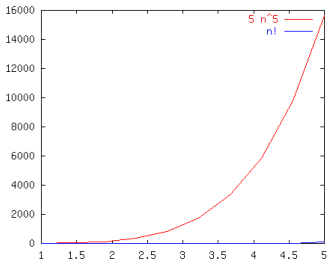
$5n^5$ vs. $n!$

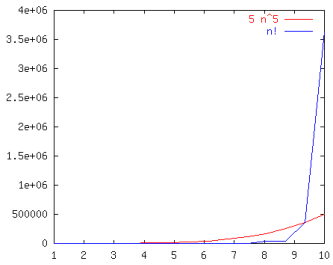


Race IV

a. Left
b. Right
c. Tied
d. It depends

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

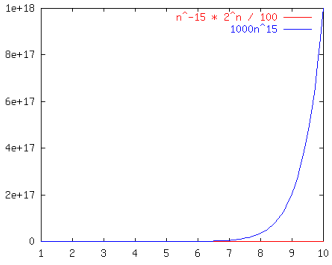




Race V

a. Left
b. Right
c. Tied
d. It depends

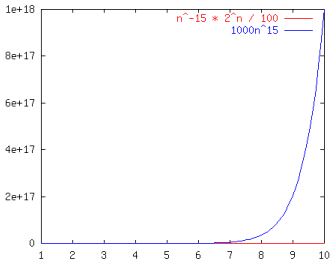
$n^{-15}2^n/100$ vs. $1000n^{15}$

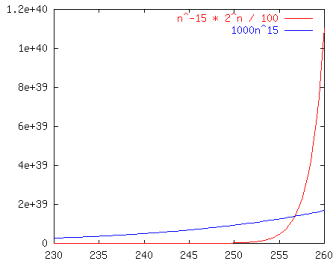


Race V

a. Left
b. Right
c. Tied
d. It depends

$n^{-15}2^n/100$ vs. $1000n^{15}$

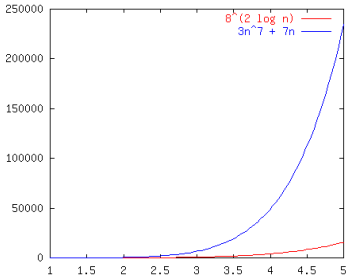




Race VI

a. Left
b. Right
c. Tied
d. It depends

$8^{2\lg(n)}$ vs. $3n^7 + 7n$



Race VII

mn^3 vs. $2^m n$

a. Left
b. Right
c. Tied
d. It depends

Silicon Downs

Post #1	Post #2	Winner
$n^3 + 2n^2$	$100n^2 + 1000$	$O(n^2)$
$n^{0.1}$	$\log n$	$O(\log n)$
$n + 100n^{0.1}$	$2n + 10 \log n$	**TIE** $O(n)$
$5n^5$	$n!$	$O(n^5)$
$n^{-15}2^n/100$	$1000n^{15}$	$O(n^{15})$
$8^{2\lg n}$	$3n^7 + 7n$	$O(n^6)$
mn^3	$2^m n$	**IT DEPENDS**

Mounties Find Silicon Downs Fixed

- The fix sheet (typical growth rates in order)
  - constant: $O(1)$
  - logarithmic: $O(\log n)$ ($\log_k n$, $\log (n^2) \in O(\log n)$)
  - poly-log: $O((\log n)^k)$
  - linear: $O(n)$
  - log-linear: $O(n \log n)$ note: even a tiny power "beats" a log
  - superlinear: $O(n^{1+c})$ (c is a constant > 0)
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - polynomial: $O(n^k)$ (k is a constant) "tractable"
  - exponential: $O(c^n)$ (c is a constant > 1) "intractable"

The VERY Fixed Parts

- There's also a notion of asymptotic "dominance", which means one function as a fraction of another (asymptotically dominant) function goes to zero.
- Each line below dominates the one above it:
  - $O(1)$
  - $O(\log^k n)$, where $k > 0$
  - $O(n^c)$, where $0 < c < 1$
  - $O(n)$
  - $O(n (\log n)^k)$, where $k > 0$
  - $O(n^{1+c})$, where $0 < c < 1$
  - $O(n^k)$, where $k \geq 2$ (the rest of the polynomials)
  - $O(c^n)$, where $c > 1$

USE those cheat sheets!

- Which is faster, n^3 or $n^3 \log n$?

(Hint: try dividing one by the other.)

- Which is faster, n^3 or $n^{3.01}/\log n$?

(Ditto the hint above!)

Today's Outline

- Programming Project #1 and Forming Teams
- Brief Proof Reminder
- Asymptotic Analysis, Briefly
- Silicon Downs and the SD Cheat Sheet
- Asymptotic Analysis, Proofs and Programs
- Examples and Exercises

Terminology (Reminder)

Given an algorithm whose running time is T(n)

- $T(n) \in O(f(n))$ if there are constants c and n_0 such that $T(n) \leq c\ f(n)$ for all $n \geq n_0$
- $T(n) \in \Omega(f(n))$ if $f(n) \in O(T(n))$
- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

Types of analysis

Orthogonal axes

- bound flavor
  - upper bound (O)
  - lower bound (Ω)
  - asymptotically tight (Θ)
- analysis case
  - worst case (adversary)
  - average case
  - best case
  - "common" case
- analysis quality
  - loose bound (any true analysis)
  - tight bound (no better "meaningful" bound that is asymptotically different)

Analyzing Code

- C++ operations - constant time
- consecutive stmts - sum of times
- conditionals - sum of branches, condition
- loops - sum of iterations
- function calls - cost of function body

Above all, use your head!

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 1: What's the input size **n**?

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 2: What kind of analysis should we perform?
Worst-case? Best-case? Average-case?

Expected-case, amortized, …

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 3: How much does each line cost? (Are lines the right unit?)

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 4: What's **T(n)** in its raw form?

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 5: Simplify **T(n)** and convert to order notation. (Also, which order notation: O, Θ, Ω?)

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 6: Casually name-drop the appropriate terms in order to sound bracingly cool to colleagues: "Oh, linear search? That's tractable, polynomial time. What polynomial? Linear, duh. See the name?! I hear it's sub-linear on quantum computers, though. Wild, eh?"

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 7: **Prove** the asymptotic bound by finding constants **c** and n_0 such that for all $n \geq n_0$, $T(n) \leq cn$.

You usually won't do this in practice.

Today's Outline

- Programming Project #1 and Forming Teams
- Brief Proof Reminder
- Asymptotic Analysis, Briefly
- Silicon Downs and the SD Cheat Sheet
- Asymptotic Analysis, Proofs and Programs
- Examples and Exercises

More Examples Than You Can Shake a Stick At (#0)

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Here's a whack-load of examples for us to:

1. find a function **T(n)** describing its runtime
2. find **T(n)**'s asymptotic complexity
3. find **c** and n_0 to prove the complexity

METYCSSA (#1)

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

Time complexity:
a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

METYCSSA (#2)

```
i = 1
while i < n do
  for j = i to n do
    sum = sum + 1
  i++
```

Time complexity:
a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

METYCSSA (#3)

```
i = 1
while i < n do
  for j = 1 to i do
    sum = sum + 1
  i += i
```

Time complexity:
a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

METYCSSA (#4)

- Conditional

  `if C then` S_1 `else` S_2

- Loops

  `while C do S`

METYCSSA (#5)

- Recursion almost always yields a *recurrence*
- Recursive max:

```
if length == 1: return arr[0]
else: return larger of arr[0] and max(arr[1..length-1])
T(1) <= b
T(n) <= c + T(n - 1)                    if n > 1
```

- Analysis

```
T(n) <= c + c + T(n - 2)                (by substitution)
T(n) <= c + c + c + T(n - 3)            (by substitution, again)
T(n) <= kc + T(n - k)                   (extrapolating 0 < k ≤ n)
T(n) <= (n - 1)c + T(1) = (n - 1)c + b  (for k = n - 1)
```

- **T(n)** ∈

METYCSSA (#6): Mergesort

- Mergesort algorithm
  - split list in half, sort first half, sort second half, merge together
- `T(1) <= b`
  `T(n) <= 2T(n/2) + cn` *if* $n > 1$
- Analysis

```
T(n) <= 2T(n/2) + cn
     <= 2(2T(n/4) + c(n/2)) + cn
      = 4T(n/4) + cn + cn
     <= 4(2T(n/8) + c(n/4)) + cn + cn
      = 8T(n/8) + cn + cn + cn
     <= 2^k T(n/2^k) + kcn              (extrapolating 1 < k ≤ n)
     <= nT(1) + cn lg n                 (for 2^k = n or k = lg n)
```

- **T(n)** ∈

METYCSSA (#7): Fibonacci

- Recursive Fibonacci:

```
int Fib(n)
  if (n == 0 or n == 1) return 1
  else return Fib(n - 1) + Fib(n - 2)
```

- *Lower* bound analysis
- $T(0), T(1) \geq b$

  $T(n) \geq T(n-1) + T(n-2) + c$ *if* $n > 1$
- Analysis

  let ϕ be $(1 + \sqrt{5})/2$ which satisfies $\phi^2 = \phi + 1$

  show by induction on n that $T(n) \geq b\phi^{n-1}$

Example #7 continued

- Basis: $T(0) \geq b > b\phi^{-1}$ and $T(1) \geq b = b\phi^0$
- Inductive step: Assume $T(m) \geq b\phi^{m-1}$ for all $m < n$

$$\begin{aligned} T(n) &\geq T(n-1) + T(n-2) + c \\ &\geq b\phi^{n-2} + b\phi^{n-3} + c \\ &\geq b\phi^{n-3}(\phi + 1) + c \\ &= b\phi^{n-3}\phi^2 + c \\ &\geq b\phi^{n-1} \end{aligned}$$

- $T(n) \in$
- Why? Same recursive call is made numerous times.

Example #7: Learning from Analysis

- To avoid recursive calls
  - store all basis values in a table
  - each time you calculate an answer, store it in the table
  - before performing any calculation for a value n
    - check if a valid answer for n is in the table
    - if so, return it
- This strategy is called "*memoization*" and is closely related to "*dynamic programming*"
- How much time does this version take?

Final Concrete Example (#8): Longest Common Subsequence

- Problem: given two strings (*m* and *n*), find the longest sequence of characters which appears in order in both strings
  - lots of applications, DNA sequencing, blah, blah, blah
- Example:
  - "**search me**" and "**insane method**" = "**same**"

Abstract Example (#9): It's Log!

Problem: find a tight bound on $T(n) = \lg(n!)$

Time complexity:

a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

Log Aside

$\log_a b$ means "the exponent that turns **a** into **b**"

$\lg x$ means "$\log_2 x$" (our usual log in CS)

$\log x$ means "$\log_{10} x$" (the common log)

$\ln x$ means "$\log_e x$" (the natural log)

But… $O(\lg n) = O(\log n) = O(\ln n)$ because:

$\log_a b = \log_c b / \log_c a$ (for $c > 1$)

so, there's just a constant factor between log bases

Asymptotic Analysis Summary

- Determine what characterizes a problem's size
- Express how much resources (time, memory, etc.) an algorithm requires as a function of input size using O(•), Ω(•), Θ(•)
  - worst case
  - best case
  - average case
  - common case
  - overall???

Some Well-Known Horses from the Downs

For general problems (not particular algorithms):
We can prove lower bounds on any solution.
We can give example algorithms to establish "upper bounds" for the best possible solution.

Searching an unsorted list using comparisons:
provably Ω(n), linear search is O(n).

Sorting a list using comparisons:
provably Ω(n lg n), mergesort is O(n lg n).

Aside: Who Cares About Ω(lg (n!))? Can You Beat O(n lg n) Search?

Chew these over:

1. How many values can you represent with n bits?
2. Comparing two values (x < y) gives you one bit of information.
3. There are n! possible ways to reorder a list. We could number them: 1, 2, …, n!
4. Sorting basically means choosing which of those reorderings/numbers you'll apply to your input.
5. How many comparisons does it take to pick among n! different values?

Some Well-Known Horses from the Downs

- Searching and Sorting: polynomial time, tractable
- Traveling Salesman Problem: non-deterministic polynomial… can check a guess in polynomial time, *maybe* exponential time to solve.

Are problems in NP really in P? **$1M** prize to prove yea or nay.

Some Well-Known Horses from the Downs

- Searching and Sorting numbers: P, tractable
- Traveling Salesman Problem: NP, intractable
- Halting Problem: uncomputable

Halting Problem: Does a given program halt on a given input. Clearly solvable in many (interesting) cases, but provably unsolvable in general.

(We can substitute "halt on" for almost anything else interesting: "print the value 7 on", "call a function named Buhler on", "access memory location 0xDEADBEEF on", …)

To Do

- Find a teammate for labs and assignments (not necessarily the same!)
- Start first written (theory) homework
- Start first programming project
- Read Epp 9.2-9.3 (for 4th ed sections, see website) and Koffman 2.6
- Prepare for upcoming labs

Coming Up

- Recursion and Induction
- Loop Invariants and Proving Program Correctness
- Call Stacks and Tail Recursion
- First Written Homework due
- First Programming Project due