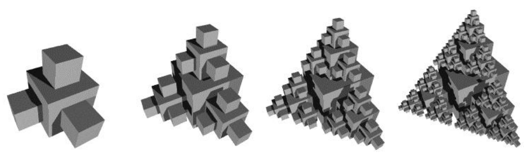
CSE 326: Data Structures
Class #4
Analysis of Algorithms III
Analysis of Recursive Algorithms



Exercise

- Form groups of 5 people (split rows in half)
- Person sitting in middle is note-taker
- Share the lists of steps for analyzing a recursive procedure. Come up with a revised list combining best ideas. (5 minutes)
- Note-taker: copy list on a transparency.
- Then: use your method to analyze the following procedure. (10 minutes)
- Note-taker: copy solution on a transparency

Recursive Selection Sort

```
Sort(int A[], int n)
{
  if (n<=1) return;
  int m = A[0];
  for (int i=1; i<n; i++){
      if (m > A[i]) {
            int tmp = A[i];
            A[i] = m;
            m = tmp;
      }
  }
  Sort( &A[1], n-1 );
}
```

How I Analyze a Recursive Program

1. Write recursive equation, using constants a, b, etc.
2. Expand the equation repeatedly, until I can see the pattern
3. Write the equation that captures the pattern – make an inductive leap! – in terms of a new variable *k*
4. Select a particular value for the variable *k* in terms of *n* – *pick a value that will make the recursive function a constant*
5. Simplify

Along the way, can throw out terms to simplify, if this is an upper-bound O( ) calculation.

Example: Sum of Integer Queue

```
sum_queue(Q){
  if (Q.length == 0 ) return 0;
  else return Q.dequeue() +
              sum_queue(Q);  }
```

- One subproblem
- Linear reduction in size (decrease by 1)
- Combining: constant c (+), 1×subproblem

Equation: $T(0) \leq b$

$T(n) \leq c + T(n-1)$ for n>0

Sum, Continued

Equation: $T(0) \leq b$

$T(n) \leq c + T(n-1)$ for n>0

Solution:

T(n)	$\leq c + c + T(n-2)$	expand recursion
	$\leq c + c + c + T(n-3)$	
	$\leq ck + T(n-k)$ for all k	inductive leap
	$\leq cn + T(0)$ for k=n	select value for k
	$\leq cn + b = O(n)$	simplify

Example: Binary Search

7	12	30	35	75	83	87	90	97	99

One subproblem, half as large

Equation: $T(1) \le b$

$T(n) \le T(n/2) + c$ for $n>1$

Solution:

$T(n) \le T(n/2) + c$	write equation
$\le T(n/4) + c + c$	expand
$\le T(n/8) + c + c + c$	
$\le T(n/2^k) + kc$	inductive leap
$\le T(1) + c \log n$ where $k = \log n$	select value for k
$\le b + c \log n = O(\log n)$	simplify

Example: MergeSort

Split array in half, sort each half, merge together

- 2 subproblems, each half as large
- linear amount of work to combine

$T(1) \le b$

$T(n) \le 2T(n/2) + cn$ for $n>1$

$T(n) \le 2T(n/2)+cn \le 2(2(T(n/4)+cn/2)+cn$	
$= 4T(n/4) +cn +cn \le 4(2(T(n/8)+c(n/4))+cn+cn$	
$= 8T(n/8)+cn+cn+cn$	expand
$\le 2^kT(n/2^k)+kcn$	inductive leap
$\le nT(1) + cn \log n$ where $k = \log n$	select value for k
$= O(n \log n)$	simplify

Lower Bound Analysis: Recursive 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 $\Omega(n)$
- Just like before, but be careful that equations are all $\ge$

Analysis

$T(0) = T(1) = a$	base case
$T(n) = b + T(n-1) + T(n-2)$	recursive case
$T(n) \ge b + 2T(n-2)$	simplify, because T is increasing
$T(n) \ge b + 2(b + 2T(n-2-2))$	expand
$T(n) \ge 3b + 4T(n-4))$	simplify
$T(n) \ge 3b + 4(b + 2T(n-4-2))$	expand
$T(n) \ge 7b + 8T(n-6))$	simplify
$T(n) \ge 7b + 8(b + 2T(n-6-2))$	expand
$T(n) \ge 15b + 16T(n-8)$	simplify
$T(n) \ge (2^k - 1)b + 2^k T(n-2k)$ for $k \le (n/2)$	inductive leap
$T(n) \ge (2^{n/2} - 1)b + 2^{n/2} T(n - 2(n/2))$	choose k=(n/2)
$T(n) \ge 2^{n/2}(b+a) - b$	simplify
$T(n) = \Omega(2^{n/2})$	Note: this is not the same as $\Omega(2^n)$!!!

Important: you introduce a new variable k! It is not necessarily the case that k=n!

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
- Memoization
  - a form of *dynamic programming*
- How much time does memoized version take?

Logs and exponents

- We will be dealing mostly with binary numbers (base 2)
- Definition: $\log_X B = A$ means $X^A = B$
- Any base is equivalent to base 2 within a constant factor:

$$\log_X B = \frac{\log_2 B}{\log_2 X}$$

- Why?

Logs and exponents

- We will be dealing mostly with binary numbers (base 2)
- Definition: $\log_X B = A$ means $X^A = B$
- Any base is equivalent to base 2 within a constant factor:

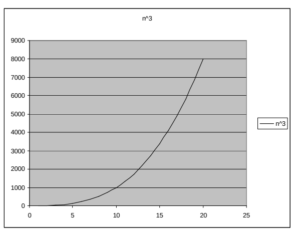
$$\log_X B = \frac{\log_2 B}{\log_2 X}$$

- Why?
- Because: if $R = \log_2 B$, $S = \log_2 X$, and $T = \log_X B$,
  - $2^R = B$, $2^S = X$, and $X^T = B$
  - $2^R = X^T = 2^{ST}$ i.e. $R = ST$ and therefore, $T = R/S$.

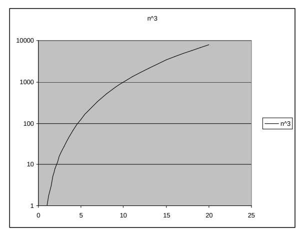
Properties of logs

- We will assume logs to base 2 unless specified otherwise
- $\log AB = \log A + \log B$ (note: $\log AB \neq \log A \cdot \log B$)
- $\log A/B = \log A - \log B$ (note: $\log A/B \neq \log A / \log B$)
- $\log A^B = B \log A$ (note: $\log A^B \neq (\log A)^B = \log^B A$)
- $\log \log X < \log X < X$ for all $X > 0$
  - $\log \log X = Y$ means $2^{2^Y} = X$
  - log X grows slower than X; called a “sub-linear” function
- $\log 1 = 0$, $\log 2 = 1$, $\log 1024 = 10$

Normal scale plot



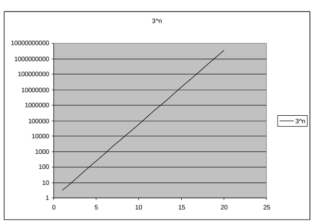
Log-Normal Plot



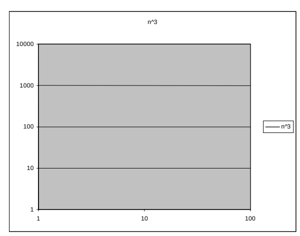
Why?

What would give a straight line?

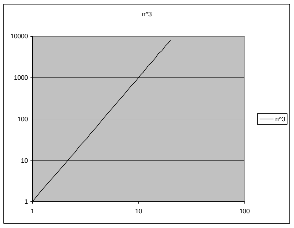
Log-Normal Plot



Log-log plot



Log-log plot



Kinds of Analysis

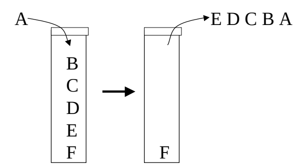
- So far we have considered worst case analysis
- We may want to know how an algorithm performs "on average"
- Several distinct senses of "on average"
  - amortized
    - average time per operation over a sequence of operations
  - average case
    - average time over a random distribution of inputs
  - expected case
    - average time for a randomized algorithm over different random seeds for any input

Amortized Analysis

- Consider any sequence of operations applied to a data structure
  - *your worst enemy could choose the sequence!*
- Some operations may be fast, others slow
- Goal: show that the average time per operation is still good

$$\frac{\text{total time for n operations}}{\text{n}}$$

Stack ADT



- Stack operations
  - push
  - pop
  - is_empty
- Stack property: if x is on the stack before y is pushed, then x will be popped after y is popped

What is biggest problem with an array implementation?

Stretchy Stack Implementation

```
int * data;
int maxsize;
int top;

Push(e){
  if (top == maxsize){
     temp = new int[2*maxsize];
     for (i=0;i<maxsize;i++) temp[i]=data[i]; ;
     delete data;
     data = temp;
     maxsize = 2*maxsize; }
  else { data[++top] = e; }
```

Best case Push = O( )

Worst case Push = O( )

Stretchy Stack Amortized Analysis

- Consider sequence of n operations
  push(3); push(19); push(2); …
- What is the max number of stretches?
- What is the total time?
  - let's say a regular push takes time *a*, and stretching an array contain *k* elements takes time *bk*.
- Amortized time =

Stretchy Stack Amortized Analysis

- Consider sequence of n operations
  push(3); push(19); push(2); …
- What is the max number of stretches?   log n
- What is the total time?
  - let's say a regular push takes time *a*, and stretching an array contain *k* elements takes time *bk*.

$$an + b(1+2+4+8+...+n) = an + b\sum_{i=o}^{\log n} 2^i$$

- Amortized time =

Series

- Arithmetic series: $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$
- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$

$$\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1}-1$$

$$\sum_{i=0}^{\log n} 2^i = \frac{2^{\log n+1}-1}{2-1} = (2^{\log n})2^1 - 1 = 2n-1$$

Stretchy Stack Amortized Analysis

- Consider sequence of n operations
  push(3); push(19); push(2); …
- What is the max number of stretches?   log n
- What is the total time?
  - let's say a regular push takes time *a*, and stretching an array contain *k* elements takes time *bk*.

$$an + b(1+2+4+8+...+n) = an + b\sum_{i=o}^{\log n} 2^i$$

$$= an + b(2n-1)$$

- Amortized time = *(an+b(2n-1))/n = O(   )*

Moral of the Story

To Do

- Assignment #1 due:
  - Electronic turnin: midnight, Monday Jan 21
  - Hardcopy writeup due in class Wednesday, Jan 23
- Finish reading Chapter 3.
  - Be prepared to discuss these questions (bring written notes to refer to):
    1. What is a call stack?
    2. Could you write a compiler that did **not** use one?
    3. What data structure does a printer queue use?