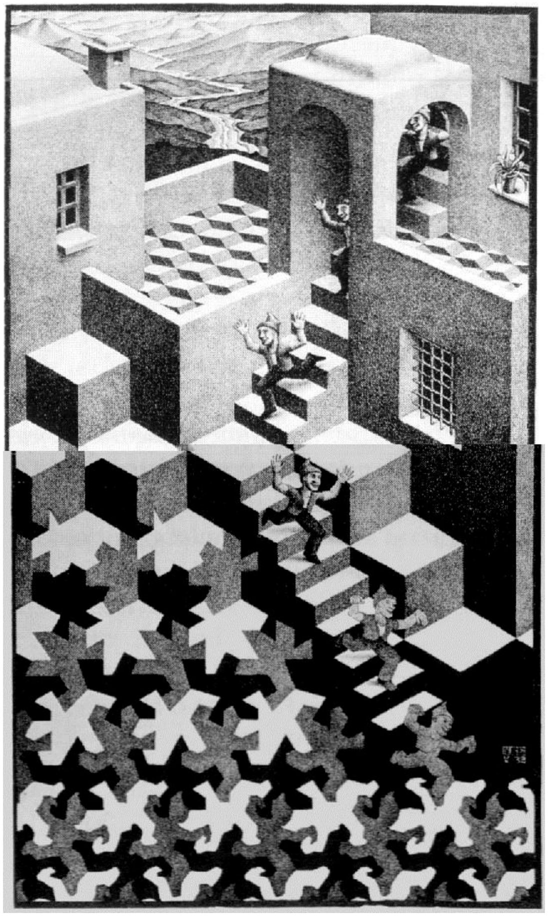
CS/ENGRD 2110
Object-Oriented Programming and Data Structures

Spring 2012

Thorsten Joachims



Lecture 23: Recurrences

Analysis of Merge-Sort

```
public static Comparable[] mergeSort(Comparable[] A, int low, int high) {
   if (low < high) { //at least 2 elements?          cost = c
      int mid = (low + high)/2;                      cost = d
      Comparable[] A1 = mergeSort(A, low, mid);      cost = T(n/2) + e
      Comparable[] A2 = mergeSort(A, mid+1, high);   cost = T(n/2) + f
      return merge(A1,A2);                           cost = g n + h
   }
   ....                                              cost = i
```

- Recurrence describing computation time:
  - $T(n) = c + d + e + f + 2\,T(n/2) + g\,n + h$ ← recurrence
  - $T(1) = i$ ← base case
- How do we solve this recurrence?

Analysis of Merge-Sort

- Recurrence:
  - $T(n) = c + d + e + f + 2\ T(n/2) + g\ n + h$
  - $T(1) = i$

- First, simplify by dropping lower-order terms and replacing constants by their max
  - $T(n) = 2\ T(n/2) + a\ n$
  - $T(1) = b$

- Simplify even more. Consider only the number of comparisons.
  - $T(n) = 2\ T(n/2) + n$
  - $T(1) = 0$

- How do we find the solution?

Solving Recurrences

- Unfortunately, solving recurrences is like solving differential equations
  - No general technique works for all recurrences

- Luckily, can get by with a few common patterns

- You learn some more techniques in CS2800

Analysis of Merge-Sort

- Recurrence for number of comparisons of MergeSort
  - $T(n) = 2T(n/2) + n$
  - $T(1) = 0$
  - $T(2) = 2$
- To show: $T(n)$ is $O(n \log(n))$ for $n \in \{2,4,8,16,32,\ldots\}$
  - Restrict to powers of two to keep algebra simpler

- Proof: use induction on $n \in \{2,4,8,16,32,\ldots\}$
  - Show $P(n) = \{T(n) \le c\, n \log(n)\}$ for some fixed constant c.
  - Base: $P(2)$
    - $T(2) = 2 \le c\, 2 \log(2)$ using $c=1$
  - Strong inductive hypothesis: $P(m) = \{T(m) \le c\, m \log(m)\}$ is true for all $m \in \{2,4,8,16,32,\ldots,k\}$.
  - Induction step: $P(2) \wedge P(4) \wedge \ldots \wedge P(k) \rightarrow P(2k)$
    - $T(2k) \le 2T(2k/2) + (2k) \le 2(c\, k \log(k)) + (2k) \le c\,(2k) \log(k) + c\,(2k)$
      $= c\,(2k)\,(\log(k) + 1) = c\,(2k) \log(2k)$ for $c \ge 1$

Solving Recurrences

- Recurrences are important when using divide & conquer to design an algorithm
- Solution techniques:
  - Can sometimes change variables to get a simpler recurrence
  - Make a guess, then prove the guess correct by induction
  - Build a recursion tree and use it to determine solution
  - Can use the Master Method
    - A “cookbook” scheme that handles many common recurrences

Master Method:

To solve $T(n) = a\,T(n/b) + f(n)$
compare $f(n)$ with $n^{\log_b a}$

- Solution is $T(n) = O(f(n))$
  if $f(n)$ grows more rapidly
- Solution is $T(n) = O(n^{\log_b a})$
  if $n\log_b a$ grows more rapidly
- Solution is $T(n) = O(f(n) \log n)$
  if both grow at same rate

Not an exact statement of the theorem – f(n) must be “well-behaved”

Recurrence Examples

Some common cases:

• $T(n) = T(n - 1) + 1$	$T(n)$ is $O(n)$	Linear Search
• $T(n) = T(n - 1) + n$	$T(n)$ is $O(n^2)$	QuickSort worst-case
• $T(n) = T(n/2) + 1$	$T(n)$ is $O(\log n)$	Binary Search
• $T(n) = T(n/2) + n$	$T(n)$ is $O(n)$	
• $T(n) = 2\ T(n/2) + n$	$T(n)$ is $O(n \log n)$	MergeSort
• $T(n) = 2\ T(n - 1)$	$T(n)$ is $O(2^n)$	

	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n\log n$	33	282	665	2469	9966
n^2	100	2500	10,000	90,000	1,000,000
n^3	1000	125,000	1,000,000	27 million	1 billion
2^n	1024	a 16-digit number	a 31-digit number	a 91-digit number	a 302-digit number
$n!$	3.6 million	a 65-digit number	a 161-digit number	a 623-digit number	unimaginably large
n^n	10 billion	an 85-digit number	a 201-digit number	a 744-digit number	unimaginably large

- protons in the known universe ~ 126 digits
- µsec since the big bang ~ 24 digits

- Source: D. Harel, *Algorithmics*

How long would it take @ 1 instruction / μsec ?

	10	20	50	100	300
n^2	1/10,000 sec	1/2500 sec	1/400 sec	1/100 sec	9/100 sec
n	1/10 sec	3.2 sec	5.2 min	2.8 hr	28.1 days
2^n	1/1000 sec	1 sec	35.7 yr	400 trillion centuries	a 75-digit number of centuries
n^n	2.8 hr	3.3 trillion years	a 70-digit number of centuries	a 185-digit number of centuries	a 728-digit number of centuries

- The big bang was 15 billion years ago ($5 \cdot 10^{17}$ secs)

- Source: D. Harel, *Algorithmics*

The Fibonacci Function

Fibonacci (Leonardo Pisano) 1170−1240?

Statue in Pisa, Italy

Giovanni Paganucci

1863

- Mathematical definition:
  - fib(0) = 0
  - fib(1) = 1
  - fib(n) = fib(n – 1) + fib(n – 2), n ≥ 2

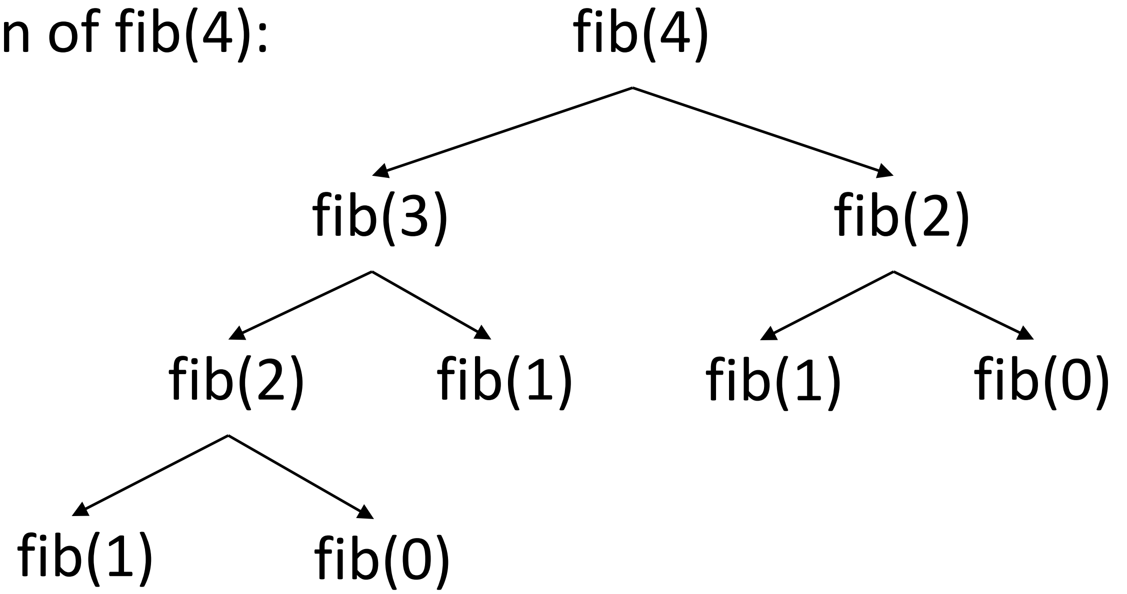
```
int fib(int n) {
   if (n == 0 || n == 1) return n;
   else return fib(n-1) + fib(n-2);
}
```

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, …

Recursive Execution

```
int fib(int n) {
   if (n == 0 || n == 1) return n;
   else return fib(n-1) + fib(n-2);
}
```

Execution of fib(4):



fib(4)

fib(3)

fib(2)

fib(2)

fib(1)

fib(1)

fib(0)

fib(1)

fib(0)

The Fibonacci Recurrence

```
int fib(int n) {
   if (n == 0 || n == 1) return n;
   else return fib(n-1) + fib(n-2);
}
```

- Recurrence for computation time:
  - $T(0) = a$
  - $T(1) = a$
  - $T(n) = T(n - 1) + T(n - 2) + a$

- What is computation time?

Analysis of Recursive Fib

- Recurrence for computation time of fib
  - $T(0) = a$
  - $T(1) = a$
  - $T(n) = T(n - 1) + T(n - 2) + a$
- To show: $T(n)$ is $O(2^n)$

- Proof: use induction on n
  - Show $P(n) = \{T(n) \leq c\, 2^n\}$ for some fixed constant c.
  - Basis: $P(0)$
    - $T(0) = a \leq c\, 2^0$ using $c=a$
  - Basis: $P(1)$
    - $T(1) = a \leq c\, 2^1$ using $c=a$
  - Strong inductive hypothesis: $P(m) = \{T(m) \leq c\, 2^m\}$ is true for all $m \leq k$.
  - Induction step: $P(0) \wedge \ldots \wedge P(k) \rightarrow P(k+1)$
    - $T(k+1) \leq T(k) + T(k-1) + a \leq c\, 2^n + c\, 2^{n-1} + a = c\, \tfrac{3}{4}\, 2^{n+1} + a \leq c\, 2^{n+1}$ for any $c \geq \tfrac{1}{4}\, a$ and any $n \geq 2$.

The Golden Ratio

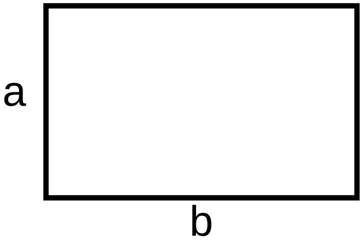
Actually, can prove a tighter bound than $O(2^n)$.

$\varphi = (a+b)/b = b/a$

$\varphi^2 = \varphi + 1$

$$\varphi = \frac{1 + \mathrm{sqrt}(5)}{2}$$

$= 1.618...$



ratio of sum of sides (a+b) to longer side (b)

=

ratio of longer side (b) to shorter side (a)

Fibonacci Recurrence is O(φ^n)

- Simplification: Ignore constant effort in recursive case.
  - $T(0) = a$
  - $T(1) = a$
  - $T(n) = T(n-1) + T(n-2)$

- Want to show $T(n) \leq c\varphi^n$ for all $n \geq 0$.
  - have $\varphi^2 = \varphi + 1$
  - multiplying by $c\varphi^n$ → $c\varphi^{n+2} = c\varphi^{n+1} + c\varphi^n$

- Base:
  - $T(0) = c = c\varphi^0$ for $c = a$
  - $T(1) = c \leq c\varphi^1$ for $c = a$
- Induction step:
  - $T(n+2) = T(n+1) + T(n) \leq c\varphi^{n+1} + c\varphi^n = c\varphi^{n+2}$

Can We Do Better?

```
if (n <= 1) return n;
int parent = 0;
int current = 1;
for (int i = 2; i ≤ n; i++) {
   int next = current + parent;
   parent = current;
   current = next;
}
return (current);
```

Time Complexity:

- Number of times loop is executed? n – 1
- Number of basic steps per loop? Constant

→ Complexity of iterative algorithm = O(n)

Much, much, much, much, better than O(φ^n)!

...But We Can Do Even Better!

- Denote with f_n the n-th Fibonacci number
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_{n+2} = f_{n+1} + f_n$
- Note that $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} f_{n+1} \\ f_{n+2} \end{pmatrix}$, thus $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$
- Can compute nth power of matrix by repeated squaring in O(log n) time.
  - Gives complexity O(log n)
  - A little cleverness got us from exponential to logarithmic.

But We Are Not Done Yet...

- Would you believe constant time?

$$f_n = \frac{\varphi^n - \varphi'^n}{\sqrt{5}}$$

where $\varphi = \dfrac{1 + \sqrt{5}}{2}$ $\qquad \varphi' = \dfrac{1 - \sqrt{5}}{2}$

Matrix Mult in Less Than $O(n^3)$

- Idea (Strassen's Algorithm): naive 2 x 2 matrix multiplication takes 8 scalar multiplications, but we can do it in 7:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{pmatrix}$$

- where
  - $s_1 = (b - d)(g + h)$
  - $s_2 = (a + d)(e + h)$
  - $s_3 = (a - c)(e + f)$
  - $s_4 = h(a + b)$

$s_5 = a(f - h)$
$s_6 = d(g - e)$
$s_7 = e(c + d)$

Now Apply This Recursively – Divide and Conquer!

- Break 2^{n+1} x 2^{n+1} matrices up into 4 2^n x 2^n submatrices
- Multiply them the same way

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

- where

$S_1 = (B - D)(G + H)$

$S_2 = (A + D)(E + H)$

$S_3 = (A - C)(E + F)$

$S_4 = H(A + B)$

$S_5 = A(F - H)$

$S_6 = D(G - E)$

$S_7 = E(C + D)$

Now Apply This Recursively – Divide and Conquer!

- Recurrence for the runtime of Strassen’s Alg
  - $M(n) = 7\ M(n/2) + cn^2$
  - Solution is $M(n) = O(n^{\log 7}) = O(n^{2.81})$
- Number of additions
  - Separate proof
  - Number of additions is $O(n^2)$

Is That the Best You Can Do?

- How about 3 x 3 for a base case?
  - best known is 23 multiplications
  - not good enough to beat Strassen
- In 1978, Victor Pan discovered how to multiply 70 x 70 matrices with 143640 multiplications, giving $O(n^{2.795\ldots})$
- Best bound to date (obtained by entirely different methods) is $O(n^{2.376\ldots})$ (Coppersmith & Winograd 1987)
- Best know lower bound is still $\Omega(n^2)$

Moral: Complexity Matters!

- But you are acquiring the best tools to deal with it!