Announcements

- Prelim tomorrow!
  - Try to arrive a little early
  - Exam will begin at 10:00, end at 11:15
  - Topics include everything covered up to and including last Friday’s class

- Assignment 2 is graded
  - Rgrade requests due Wednesday, 11:59PM

- Assignment 3 is (still) posted
  - Due Thursday 11:59PM
  - Check newsgroup for clarifications, corrections, etc.

Analysis of Merge-Sort

Recurrence:
\[ T(n) = c + d + e + f + 2T(n/2) + gn + h \]
\[ T(1) = i \]

First, simplify by dropping lower-order terms
Simplified recurrence:
\[ T(n) \leq 2T(n/2) + cn \]
\[ T(1) = i \]

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 will learn some more techniques in CS 280

Proof: strong induction on \( n \)

- Show that
  - \( T(2) = 2c \)
  - \( T(n) = 2T(n/2) + cn \)
  - \( T(n) = cn \log n \)
- Basis
  - \( T(2) = 2c \leq 2 \log 2 \)
- Induction step
  - \( T(n) = 2T(n/2) + cn \)
  - \( = 2(cn \log n/2) + cn \) (IH)
  - \( = cn \log n - cn \) + \( cn \log n \)
Solving Recurrences

- Recurrences are important when designing divide & conquer algorithms.
- Solution techniques:
  - Can sometimes change variables to get a simpler recurrence
  - Make a guess, then prove the guess correct by induction
  - Can use the Master Method
  - A "cookbook" scheme that handles many common recurrences

Master Method

- To solve recurrences of the form
  \[ T(n) = aT\left(\frac{n}{b}\right) + f(n), \]
  with constants \(a \geq 0, b > 1\), compare \(f(n)\) with \(n^{\log_b a}\).

- If \(f(n)\) grows more rapidly,
  - Solution is \(T(n) = \Omega(f(n))\)

- If \(n^{\log_b a}\) grows more rapidly
  - Solution is \(T(n) = \Omega(n^{\log_b a})\)

- If both grow at same rate
  - Solution is \(T(n) = \Omega(f(n) \log n)\)

Recurrence Examples

- \(T(n) = T(n - 1) + 1\) → \(T(n) = O(n)\) Linear Search
- \(T(n) = T(n - 1) + n\) → \(T(n) = O(n^2)\) QuickSort worst-case
- \(T(n) = T(n/2) + 1\) → \(T(n) = O(\log n)\) Binary Search
- \(T(n) = 2T(n - 1)\) → \(T(n) = O(2^n)\) MergeSort
- \(T(n) = 4T(n/2) + n\) → \(T(n) = O(n \log n)\) MergeSort

How long would it take @ 1 instruction / \(\mu\)sec?

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

The Fibonacci Function

- Mathematical definition:
  \[ \text{fib}(0) = 0 \]
  \[ \text{fib}(1) = 1 \]
  \[ \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2), \ n \geq 2 \]

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

```c
static int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2); }
```
Recursive Execution

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

Execution of fib(4):
```
fib(4) → fib(3) → fib(2) → fib(1) → fib(0)
```

The Fibonacci Recurrence

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

T(0) = c
T(1) = c
T(n) = T(n – 1) + T(n – 2) + c

• Solution is exponential in n
• But not quite O(2^n)...

The Golden Ratio

\[ \phi = \frac{a+b}{b} = \frac{b}{a} \]
\[ \phi^2 = \phi + 1 \]
\[ \phi = \frac{1 + \sqrt{5}}{2} = 1.618... \]

Fibonacci Recurrence is O(\(\phi^n\))

• want to show T(n) \(\leq c\phi^n\)
• have \(\phi^2 = \phi + 1\)
• multiplying by \(c\phi^n\), \(c\phi^{n+2} = c\phi^{n+1} + c\phi^n\)
• Basis:
  • T(0) = c = c\(\phi^0\)
  • T(1) = c \(\leq c\phi^1\)
• Induction step:
  • T(n+2) = T(n+1) + T(n) \(\leq c\phi^{n+1} + c\phi^n = c\phi^{n+2}\)

Can We Do Better?

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

• Number of times loop is executed? Less than n
• Number of basic steps per loop? Constant
• Complexity of iterative algorithm = O(n)
• Much, much, much, much, much, better than O(\(\phi^n\))!

...But We Can Do Even Better!

• Let \(f_n\) denote the nth Fibonacci number
  • \(f_0 = 0\)
  • \(f_1 = 1\)
  • \(f_{n+2} = f_{n+1} + f_n\), \(n \geq 0\)
• Note that
  \[
  \begin{pmatrix}
  0 & 1 \\
  1 & 1
  \end{pmatrix}
  \begin{pmatrix}
  f_n \\
  f_{n+1}
  \end{pmatrix}
  =
  \begin{pmatrix}
  f_{n+1} \\
  f_{n+2}
  \end{pmatrix}
\]
• Can compute the nth power of a matrix by repeated squaring in O(log n) time
• Gives complexity O(log n)
• Just a little cleverness got us from exponential to logarithmic!
Matrix Multiplication in Less Than $O(n^3)$ (Strassen’s Algorithm)

- Idea: naive 2 x 2 matrix multiplication takes 8 scalar multiplications, but we can do it in 7:

\[
\begin{pmatrix}
  a & b \\
  c & d
\end{pmatrix}
\begin{pmatrix}
  u & v \\
  w & x
\end{pmatrix}
= \begin{pmatrix}
  s_1 + s_2 + s_4 + s_6 & s_3 + s_6 + s_7 \\
  s_2 + s_3 + s_7 & s_4 + s_5 + s_7
\end{pmatrix}
\]

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 $2^{n+1} \times 2^{n+1}$ matrices up into 4 $2^n \times 2^n$ submatrices
- Multiply them the same way

\[
\begin{pmatrix}
  A & B \\
  C & D
\end{pmatrix}
\begin{pmatrix}
  G & H \\
  I & J
\end{pmatrix}
= \begin{pmatrix}
  S_1 + S_2 + S_4 + S_6 & S_3 + S_6 + S_7 \\
  S_2 + S_3 + S_7 & S_4 + S_5 + S_7
\end{pmatrix}
\]

where

- $S_1 = (B - D)(G + H)$
- $S_2 = (A + D)(E + H)$
- $S_3 = (A - C)(E + F)$
- $S_4 = H(A + B)$
- $S_5 = A(F - H)$
- $S_6 = D(G - E)$
- $S_7 = E(C + D)$

Moral: Complexity Matters!

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