Solving Recurrences

Lecture 13
CS2110 – Summer 2008

Announcements

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

- Assignment 2 is graded
- Re-grade requests due Friday (??)
- Solutions available on CMS

- Assignment 3 solutions?

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:

\[ T(n) = 2T(n/2) + cn \]

\[ T(1) = i \]

How do we find the solution?

Proof: strong induction on n

Show that

\[ T(2) = 2c \]

\[ T(n) \leq 2T(n/2) + cn \]

imply

\[ T(n) \leq cn \log n \]

Basis

\[ T(2) = 2c = c \log 2 \]

Solution is \( T(n) = \Theta(n \log n) \)

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

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:

\[ T(n) = 2T(n/2) + gn \]

\[ T(1) = i \]

How do we find the solution?

Recurrence:

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

\[ T(1) = i \]

How do we solve this recurrence?

How do we solve this recurrence?

Solution is \( T(n) = \Theta(n \log n) \)
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
  - Build a recursion tree and use it to determine solution
  - Use the Master Method
    - A "cookbook" scheme that handles many common recurrences

Master Method

- To solve recurrences of the form $T(n) = aT(n/b) + f(n)$, with constants $a \geq 0$, $b > 1$, compare $f(n)$ with $n^{\log_b a}$
  - If $f(n)$ grows more rapidly, solution is $T(n) = O(f(n))$
  - If $n^{\log_b a}$ grows more rapidly, solution is $T(n) = O(n^{\log_b a})$
  - If both grow at same rate, solution is $T(n) = O(f(n) \log n)$
- Not an exact statement of the theorem – $f(n)$ must be "well-behaved"

Recurrence Examples

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

How long would it take @ 1 instruction / μsec?

- The big bang was 15 billion years ago (5.10^{17} sec)
- Source: D. Harel, Algorithmics

The Fibonacci Function

- Mathematical definition:
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$, $n \geq 2$
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, …

```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);
}
```

- Source: D. Harel, Algorithmics
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 $\phi^n$, \(c\phi^{n+2} = c\phi^{n+1} + c\phi^n\)
- 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?

- 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 $n^{th}$ Fibonacci number
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_{n+2} = f_{n+1} + f_n$, $n \geq 0$
- Note that \[
\begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
f_n \\
f_{n+1}
\end{bmatrix}
= \begin{bmatrix}
f_{n+1} \\
f_{n+2}
\end{bmatrix}
\]
- Can compute the $n^{th}$ 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:

\[
\begin{pmatrix}
a & b \\
c & d \\
\end{pmatrix}
\begin{pmatrix}
a' & b' \\
c' & d' \\
\end{pmatrix} =
\begin{pmatrix}
a_1 + b_2 + s_5 + s_6 & s_1 + s_2 - s_4 - s_6 \\
\end{pmatrix}
\]


where

\[
\begin{align*}
s_1 & = (b - d)(g + h) \\
s_2 & = (a + d)(e + h) \\
s_3 & = (a - c)(e + f) \\
s_4 & = h(a + b) \\
s_5 & = a(f - h) \\
s_6 & = d(g - e) \\
s_7 & = e(c + d)
\end{align*}
\]

Now Apply This Recursively – Divide and Conquer!

- Break 2^{2^k} x 2^{2^k} matrices up into 4 2^k x 2^k submatrices

\[
\begin{pmatrix}
a & b \\
c & d \\
\end{pmatrix}
\begin{pmatrix}
a' & b' \\
c' & d' \\
\end{pmatrix} =
\begin{pmatrix}
s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\
\end{pmatrix}
\]

Now Apply This Recursively – Divide and Conquer!

- Gives recurrence $M(n) = 7 M(n/2) + cn^2$ for the number of multiplications

\[
\begin{align*}
S_1 & = (B - D)(G + H) \\
S_2 & = (A + D)(E + H) \\
S_3 & = (A - C)(E + F) \\
S_4 & = H(A + B) \\
S_5 & = A(F - H) \\
S_6 & = D(G - E) \\
S_7 & = E(C + D)
\end{align*}
\]

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...)

- Best bound to date (obtained by entirely different methods) is O(n^2.376...) (Coppersmith & Winograd 1987)

- Best known lower bound is still Ω(n^2)

Moral: Complexity Matters!

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