Solving Recurrences

Lecture 13

CS2110 – Summer 2009

In the past…

Complexity:

- Big-O definition
- Non-recursive programs

Today

- Complexity of recursive programs
  - Introduce *recurrences*
  - Methods to solve recurrences
    - Substitution
    - Recursion tree
    - Master

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 = gn + h
   }
   Comparable[] temp = new Comparable[1];            cost = i (base case)
   temp[0] = A[low];
   return temp;
}
```

Recurrence:

$T(n) = c + d + e + f + 2T(n/2) + gn + h$ ← recurrence

$T(1) = i$ ← base case

- Recurrences are important when designing divide & conquer algorithms
- How do we solve this recurrence?

Solving Recurrences

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

- Some techniques:
  - Substitution:
    - Make a guess, then prove the guess correct by induction
    - Can sometimes change variables to get a simpler recurrence
  - Recursion-tree:
    - Build a recursion tree and use it to determine solution
  - Can use the *Master Method*
    - A “cookbook” scheme that handles many common recurrences

Substitution Method

1. Guess the form of the solution
2. Use induction to show that the solution works.

Ex1: Analysis of Merge-Sort

Recurrence:

$T(n) = c + d + e + f + 2T(n/2) + gn + h$

$T(1) = i$

First, simplify the recurrence:

$T(n) \leq 2T(n/2) + kn$

$T(1) = i$

Analysis of Merge-Sort

- Recurrence for MergeSort
  - $T(n) \leq 2T(n/2) + kn$
  - $T(2) = 2i + 2k$
  - Solution is $T(n) = O(n \log n)$

- Proof: strong induction on n
- Show that

  $T(2) \leq 2i + 2k$

  $T(n) \leq 2T(n/2) + kn$

  imply

  $T(n) \leq pn \log n$ (where $p \geq i + k$)

- Basis

  $T(2) = 2(i + k) \leq 2p = p\, 2 \log 2$

- Induction step

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + kn \\
&\leq 2(pn/2 \log n/2) + kn \quad \text{(IH)} \\
&= pn (\log n - 1) + kn \\
&= pn \log n + (k - p)n \\
&\leq pn \log n
\end{aligned}
$$

Ex2: 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, …

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

Fibonacci (Leonardo Pisano) 1170–1240?

Statue in Pisa, Italy
Giovanni Paganucci
1863

Recursive Execution

```
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(2) fib(1) fib(1) fib(0)

fib(1) fib(0)

The Fibonacci Recurrence

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

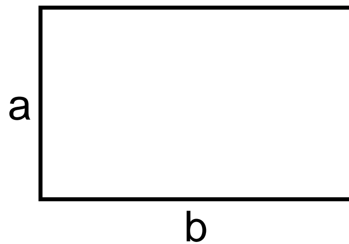
$T(0) = c$

$T(1) = c$

$T(n) = T(n-1) + T(n-2) + c$

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

The Golden Ratio



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

=

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

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

$\varphi^2 = \varphi + 1$

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

$= 1.618...$

Fibonacci Recurrence is $O(\varphi^n)$

- want to show $T(n) \leq c\varphi^n$
- have $\varphi^2 = \varphi + 1$
- multiplying by $c\varphi^n$, $c\varphi^{n+2} = c\varphi^{n+1} + c\varphi^n$

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

Detour:

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);
```

- 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 $O(\varphi^n)$!

Detour:

...But We Can Do Even Better!

- Let f_n denote the nth Fibonacci number
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_{n+2} = f_{n+1} + f_n, \; n \ge 0$

- Note that $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} f_{n+1} \\ f_{n+2} \end{pmatrix}$, 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 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!

Detour:

Recall Problem-Size Examples

- Suppose we have a computing device that can execute 1000 operations per second; how large a problem can we solve?

	1 second	1 minute	1 hour	1 century
log n	2^{1000}	$2^{60,000}$	$2^{3,600,000}$	$2^{3.2 \times 10^{12}}$
n	1000	60,000	3,600,000	3.2×10^{12}
n log n	140	4893	200,000	8.7×10^{10}
n^2	31	244	1897	1,776,446
$3n^2$	18	144	1096	1,025,631
n^3	10	39	153	1,318
2^n	9	15	21	41

Hints on Guessing

- Use recursion trees (next)
- Use similar, known recurrences
  - E.g., $T(n) = 2T(n/2 + 17) + n \Rightarrow O(n \log n)$
- Loose bounds for first guess, then adjust gradually
  - E.g., $T(n) = 2T(n/2) + n$
  - First lower bound: $O(n)$
  - First upper bound: $O(n^2)$

Hints on Proving

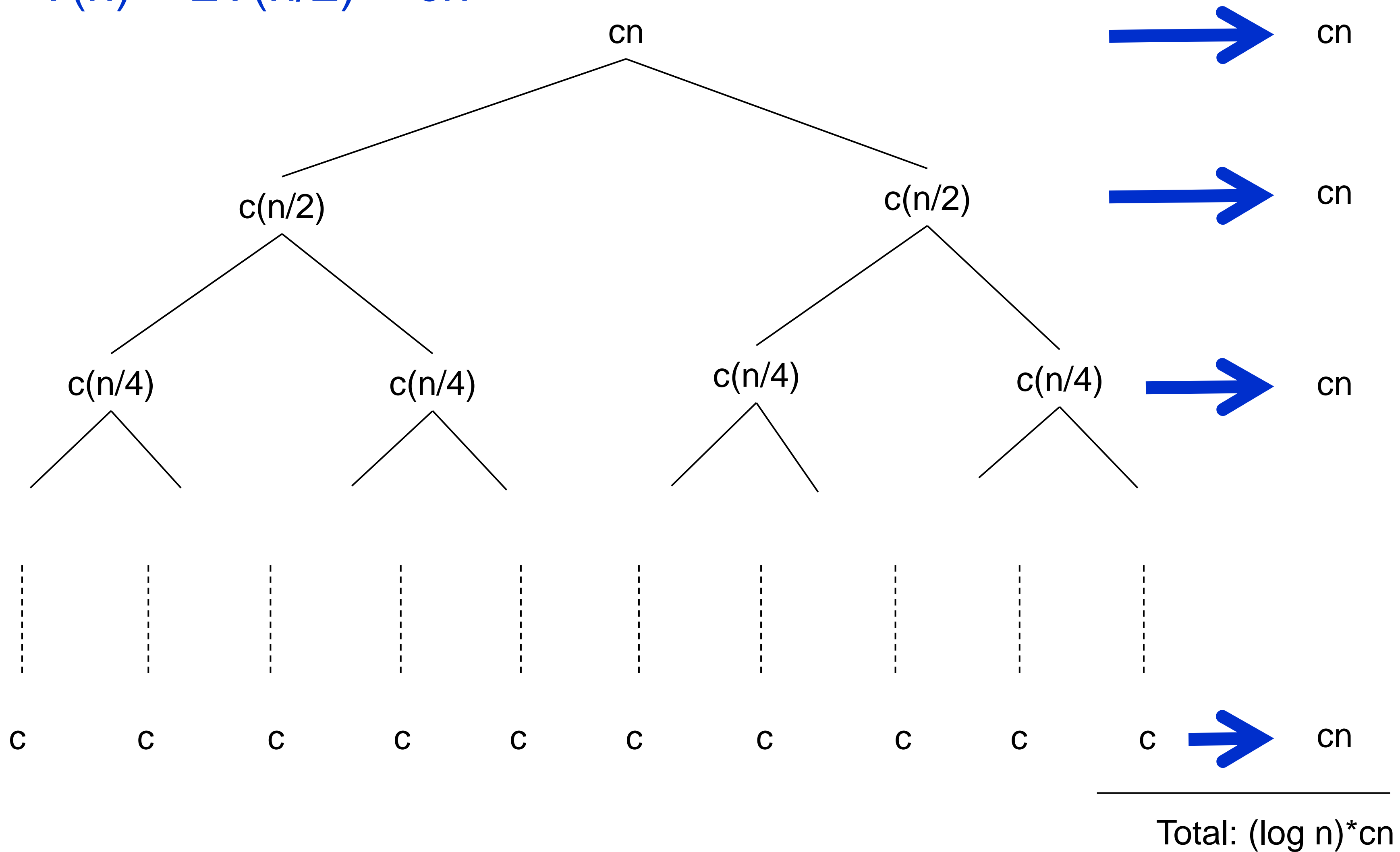
- Prove the *exact form*
  - E.g., $T(n) = 2T(n/2) + n$, guess $T(n) = O(n)$
    - $T(n) \leq 2(cn/2) + n$
      $\leq cn + n$
      $= O(n)$ ← WRONG
- Change variables
  - E.g., $T(n) = 2T(n^{.5}) + \log n$
  - Let $m = \log n$
  - $\Rightarrow T(2^m) = 2T(2^{m/2}) + m$
  - Let $S(m) = T(2^m)$
  - $\Rightarrow S(m) = 2S(m/2) + m$
  - $\Rightarrow S(m) = O(m \log m)$
  - $\Rightarrow T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n * \log \log n)$

Recursion Tree Method

- Node = cost of a *single subproblem*
- Sum nodes in same level to get per-level costs
- Sum all per-level costs to get total costs

Ex: Merge-Sort

- $T(n) = 2T(n/2) + cn$



Master Method

- To solve recurrences of the form

$$T(n) = aT(n/b) + f(n),$$

with constants $a \geq 1$, $b > 1$, compare $f(n)$ with $n^{\log_b a}$

  - if $f(n)$ grows more rapidly, and if $a*f(n/b) \leq c*f(n)$ for some $c \leq 1$, n sufficiently large
    - 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 $T(n) = O(f(n) \log n)$

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
- $T(n) = T(n/2) + n$ → $T(n) = O(n)$
- $T(n) = 2\,T(n/2) + n$ → $T(n) = O(n \log n)$ MergeSort
- $T(n) = 2\,T(n-1)$ → $T(n) = O(2^n)$
- $T(n) = 4\,T(n/2) + n$ → $T(n) = O(n^2)$

Moral: Complexity Matters!

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