

CS 432, Algorithms: Recurrences, D & C

- First design strategy: Divide and Conquer
 - Examples...
- Recursive algorithms
- Counting basic operations in recursive algorithms: Solving recurrence relations
 - By iteration method
 - Recursion trees (quick view)
 - The "Main" and "Master" Theorems
- Mergesort
- Trominos

Recursion: Basic Concepts and Review

- Recursive definitions in mathematics
 - Factorial: $n! = n(n-1)!$ and $0! = 1! = 1$
 - Fibonacci numbers:
 $F(0) = F(1) = 1$
 $F(n) = F(n-1) + F(n-2)$ for $n > 1$
 - Note base case
- In programming, recursive functions can be implemented
 - First, check for simple solutions and solve directly
 - Then, solve simpler subproblem(s) by calling same function
 - Must make progress towards base cases
- Design strategy: *method99* "mental trick"

Designing Recursive Procedures

- Think Inductively!
- converging to a base case (stopping the recursion)
 - identify some unit of measure (running variable)
 - identify base cases
- How to solve p for all inputs from size 0 through 100
 - Assume *method99* solves sub-problem all sizes 0 through 99
 - if p detect a case that is not base case it calls *method99*
- *method99* works and is called when:
 1. The sub-problem size is less than p 's problem size
 2. The sub-problem size is not below the base case
 3. The sub-problem satisfies all other preconditions of *method99* (which are the same as the preconditions of p)

Recursion: Good or Evil?

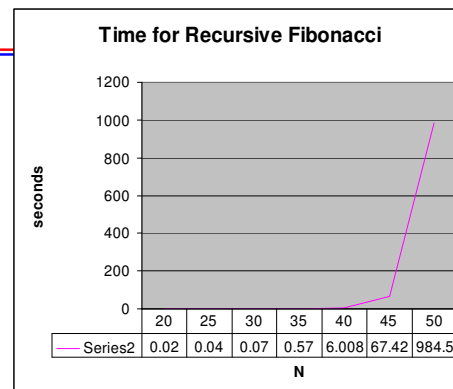
- It depends...
- Sometimes recursion is an efficient design strategy, sometimes not
 - Important! we can define recursively and implement non-recursively
- Note that many recursive algorithms can be re-written non-recursively
 - Use an explicit stack
 - Remove tail-recursion (compilers often do this for you)
- Consider: factorial, binary search, Fibonacci
 - Let's consider Fibonacci carefully...

Implement Fibonacci numbers

- It's beautiful code, no?

```
long fib(int n) {  
    assert(n >= 0);  
    if ( n == 0 ) return 1;  
    if ( n == 1 ) return 1;  
    return fib(n-1) + fib(n-2);  
}
```

- Let's run and time it.
- Let's trace it.



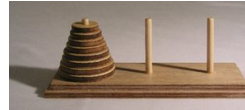
Towers of Hanoi

- Ah, the legend:
 - 64 golden disks
 - Those diligent priests
 - The world ends!



Towers of Hanoi

- Back in the commercial Western world...
- Game invented by the French mathematician, Edouard Lucas, in 1883.
- Now, for only \$19.95, call now!



Wake Up and Design!

- Write a recursive function for the Towers of Hanoi.
 - Number each peg: 1, 2, 3
 - Function signature:
hanoi (n, source, dest, aux)
where:
n is number of disks (from the top), and
other parameters are peg values
In function body print:
Move a disk from <peg> to <peg>
- Do this in pairs. Then pairs group and compare. Find bugs, issues, etc. Explain to each other. Turn in one sheet with all four names.

Divide and Conquer: A Strategy

- Our first design strategy: Divide and Conquer
- Often recursive, at least in definition
- Strategy:
 - Break a problem into 1 or more smaller subproblems that are identical in nature to the original problem
 - Solve these subproblems (recursively)
 - Combine the results for the subproblems (somehow) to produce a solution to original problem
- Note the assumption:
 - We can solve original problem given subproblems' solutions

Design Strategy: Divide and Conquer

- It is often easier to solve several small instances of a problem than one large one.
 - divide** the problem into smaller instances of the same problem
 - solve (**conquer**) the smaller instances recursively
 - combine** the solutions to obtain the solution for original input
 - Must be able to solve one or more small inputs **directly**
- Solve(I)
n = size(I)
if (n <= smallsize)
 solution = directlySolve(I);
else
 divide I into I1, ..., Ik.
 for each i in {1, ..., k}
 Si = solve(Ii);
 solution = combine(S1, ..., Sk);
return solution;

Why Divide and Conquer?

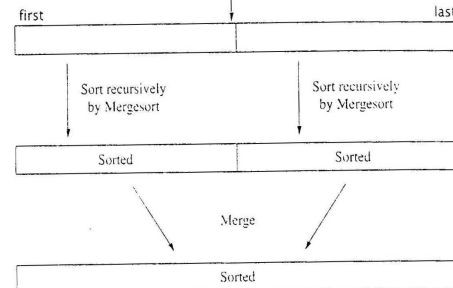
- Sometimes it's the simplest approach
- Divide and Conquer is often more efficient than "obvious" approaches
 - E.g. Mergesort, Quicksort
- But, not necessarily efficient
 - Might be the same or worse than another approach
- Must analyze cost
- Note: divide and conquer may or may not be implemented recursively

Cost for a Divide and Conquer Algorithm

- Perhaps there is...
 - A cost for dividing into sub problems
 - A cost for solving each of several subproblems
 - A cost to combine results
- So (for $n > \text{smallSize}$)
 $T(n) = D(n) + \text{Sum}[T(\text{size}(I_i))] + C(n)$
 - often rewritten as
 $T(n) = a T(n/b) + f(n)$
- These formulas are *recurrence relations*

Mergesort is Classic Divide & Conquer

- Mergesort Strategy $\lfloor (first + last)/2 \rfloor$



Algorithm: Mergesort

- Specification:
 - Input: Array E and indexes first, and Last, such that the elements $E[i]$ are defined for $first \leq i \leq last$.
 - Output: $E[first], \dots, E[last]$ is sorted rearrangement of the same elements
- Algorithm:

```
void mergeSort(Element[] E, int first, int last)
if (first < last)
    int mid = (first+last)/2;
    mergeSort(E, first, mid);
    mergeSort(E, mid+1, last);
    merge(E, first, mid, last); // merge two sorted halves
return;
```

Exercise: Find Max and Min

- Given a list of elements, find both the maximum element and the minimum element
- Obvious solution:
 - Consider first element to be max
 - Consider first element to be min
 - Scan linearly from 2nd to last, and update if something larger than max or if something smaller than min
- Class exercise:
 - Write a recursive function that solves this using divide and conquer.
 - Prototype: `void maxmin(list, first, last, max, min);`
 - Base case(s)? Subproblems? How to combine results?

Solving Recurrence Relations

- Several methods:
 - Substitution method, AKA iteration method, AKA method of backwards substitutions
 - We'll do this in class
 - Recurrence trees
 - Not in our text. (In the Baase text from 2003.)
 - Sometimes a picture is worth 2^{10} words!
 - "Main" Theorem and the "Master" theorem
 - Easy to find Order-Class for a number of common cases
 - Textbook: Main Theorem
 - Other texts: slightly different Master Theorem

Iteration or Substitution Method

- Strategy
 - Write out recurrence, e.g. $W(n) = W(n/2) + 1$
 - BTW, this is a recurrence for binary search
 - Substitute for the recursive definition on the right-hand side by re-applying the general formula with the smaller value
 - In other words, plug the smaller value back into the main recurrence
 - So now: $W(n) = (W(n/4) + 1) + 1$
 - Repeat this several times and write it in a general form (perhaps using some index i to show how often it's repeated)
 - So now: $W(n) = W(n/2^i) + i$

Substitution Method (cont'd)

- So far we have: $W(n) = W(n/2^i) + i$
- This is the form after we repeat i times. How many times can we repeat?
 - Use base case to solve for i
 - Here, $W(1) = 1$, so we reach this when $n/2^i$ is 1.
 - Solve for i : so $i = \lg n$
 - Plug this value of i back into the general recurrence:

$$W(n) = W(n/2^i) + i = W(n/n) + \lg n = \lg n + 1$$
 - Note: We assume n is some power of 2, right?
 - That's OK. There is a theorem called the smoothness rule that states that we'll have the correct order-class
 - See Example 2.4.6, page 58

Examples Using the Substitution Method

Practice with the following:

1. Finding max and min
 - $W(1) = 0, W(n) = 2W(n/2) + 2$
 - Is this better or worse than the "scanning" approach?
2. Mergesort
 - $W(1) = 0, W(n) = 2W(n/2) + n - 1$
3. Towers of Hanoi
 - Write the recurrence. (Now, in class.)
 - Solve it. (At home!)

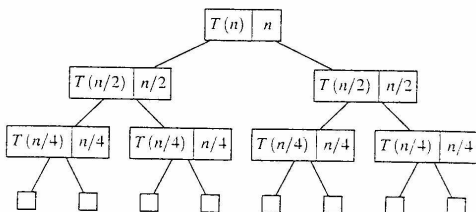
Return to Fibonacci...

- Can we use the substitution method to find out the $W(n)$ for our recursive implementation of $\text{fib}(n)$?
 - Nope. There's another way to solve recurrence, which we won't do in this class (you saw this in CS 202?)
 - homogenous second-order linear recurrence with constant coefficients
 - This method allows us to calculate $F(n)$ "directly":

$$F(n) = (1 / \sqrt{5}) \Phi^n$$
 rounded to nearest int, where Φ is the Golden Ratio, about 1.618
 - Isn't this $\Theta(1)$ while a loop is $\Theta(n)$? (Just punch buttons on my calculator!)
 - Without a table or a calculator, finding Φ^n is linear (just like finding $F(n)$ with a loop)

Evaluate recursive equation using Recursion Tree

- Evaluate: $T(n) = T(n/2) + T(n/2) + n$
 - Work copy: $T(k) = T(k/2) + T(k/2) + k$
 - For $k=n/2, T(n/2) = T(n/4) + T(n/4) + (n/2)$
- [size] non-recursive cost]



Recursion Tree: Total Cost

- To evaluate the total cost of the recursion tree
 - sum all the non-recursive costs of all nodes
 - = Sum (rowSum(cost of all nodes at the same depth))
- Determine the maximum depth of the recursion tree:
 - For our example, at tree depth d the size parameter is $n/(2^d)$
 - the size parameter converging to base case, i.e. case 1
 - such that, $n/(2^d) = 1$,
 - $d = \lg(n)$
 - The rowSum for each row is n
- Therefore, the total cost, $T(n) = n \lg(n)$

The Master Theorem

- Given: a *divide and conquer* algorithm
 - An algorithm that divides the problem of size n into a subproblems, each of size n/b
 - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
- Then, the Master Theorem gives us a cookbook for the algorithm's running time
 - Our textbook has a simpler version they call the "Main Recurrence Theorem"

The Master Theorem (from Cormen's)

- If $T(n) = aT(n/b) + f(n)$ then let $k = \lg a / \lg b = \log_b a$ (critical exponent)
- Then three common cases based on how quickly $f(n)$ grows
 - If $f(n) \in O(n^{k-\epsilon})$ for some positive ϵ , then $T(n) \in \Theta(n^k)$
 - If $f(n) \in \Theta(n^k)$ then $T(n) \in \Theta(f(n) \log(n)) = \Theta(n^k \log(n))$
 - If $f(n) \in \Omega(n^{k+\epsilon})$ for some positive ϵ , and $f(n) \in O(n^{k+\delta})$ for some positive $\delta > \epsilon$, then $T(n) \in \Theta(f(n))$
- Note: none of these cases may apply

The Main Recurrence Theorem (from our text)

- If $T(n) = aT(n/b) + f(n)$ and $f(n) = \Theta(n^k)$
- Cases for exact bound:
 - $T(n) \in \Theta(n^k)$ if $a < b^k$
 - $T(n) \in \Theta(n^k \log(n))$ if $a = b^k$
 - $T(n) \in \Theta(n^E)$ where $E = \log_b(a)$ if $a > b^k$
- Note book's similar cases for upper and lower bound
- Note $f(n)$ is polynomial
 - This is less general than earlier Master Theorem

Using The Master Method

- $T(n) = 9T(n/3) + n$
 - $a=9, b=3, f(n) = n$
- Main Recurrence Theorem
 - $a > b^k$ $9 > 3$, so $\Theta(n^E)$ where $E = \log_3(9) = 2$, $\Theta(n^2)$
- Master Theorem
 - $k = \lg 9 / \lg 3 = \log_3 9 = 2$
 - Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, case 1 applies:
 $T(n) \in \Theta(n^E)$
 - Thus the solution is $T(n) = \Theta(n^2)$ since $E=2$

Problems to Try

- Can you use a theorem on these?
Can you successfully use the iteration method?
- $T(n) = T(n/2) + \lg n$
- $T(n) = T(n/2) + n$
- $T(n) = 2T(n/2) + n$ (like Mergesort)
- $T(n) = 2T(n/2) + n \lg n$

Common Forms of Recurrence Equations

- Recognize these:
 - Divide and conquer
 $T(n) = bT(n/c) + f(n)$
 - Solve directly or apply master theorem
 - Chip and conquer:
 $T(n) = T(n-c) + f(n)$
 - Note: One subproblem of lesser cost!
 - Chip and Be Conquered:
 $T(n) = bT(n-c) + f(n)$ where $b > 1$
 - Like Towers of Hanoi

Back to Towers of Hanoi

- Recurrence:
 $W(1) = 1; \quad W(n) = 2 W(n-1) + 1$
- Closed form solution:
 $W(n) = 2^n - 1$
- Original "legend" says the monks moves 64 golden disks
 - And then the world ends! (Uh oh.)
 - That's 18,446,744,073,709,551,615 moves!
 - If one move per second, day and night, then 580 billion years
 - Whew, that's a relief!