Recursion (Review!)

Wednesday, September 19

Announcements

- No class on Friday, and no office hours either (Alexa out of town)
- Unfortunately, Alexa will also just be incommunicado, even via email, from Wednesday night until Monday morning. Apologies.
- Prelab 3 is due by the beginning of class on Monday. You must hand in a paper copy by the beginning of class in class. Problems put in my box or handed in in lab are not accepted. Just reminding you.
- Please do the Eclipse tutorial on its debugger (as suggested in lab 0).
- Please read section 3.2 of the class notes. It is worth your while, and will make debugging your code so much easier.

Recursive Methods

A recursive method is a method that either directly or indirectly calls itself. Use recursion when solving a smaller version of your problem will help you find a solution to the original (larger) version of your problem.

For example, we can (recursively) define the sum of the first n positive integers:

$$\text{sum}(\ n) = \text{sum}(n-1) + n$$

Recursive methods need:

- A recursive case
- An instance that can be solved without recursion
- A base case (or base cases, if necessary)

Every recursive method needs

And add a corresponding elseif statement to our method:

```java
public long sum( int n ) {
    if (n==0) return 0;
    return n + sum(n-1);
}
```

We can easily translate this definition into code:

```java
public long sum( int n ) {
    if (n==0) return 0;
    return n + sum(n-1);
}
```
Recursion Example

// input: a list L of n items, in sorted order, an item x
// goal: determine whether x is in the list L

binarySearch( list L, item x )

what is the base case? do we need more than 1?
what is our recursive case?

Last week we saw an iterative (i.e. non-recursive) implementation of binary search. We can also implement it recursively:

What is the running time of this method? Hopefully same as iterative version...

If T(n) represents the running time of this method on a list of length n, then

T(n) = T(n/2) + O(1), with initial condition T(1)=1

Wednesday, September 19, 12

"Divide and Conquer" Example

For example, consider the following problem, called sorting:

input: a list L of n integers
goal: return L sorted in increasing (or, non-decreasing) order

Divide and conquer is a 2-step recursive technique
• divide ---
  divide your problem into (> 1) smaller problems that are solved recursively
• conquer ---
  combine the recursive solutions into an answer for the original problem

mergeSort( list L ) // returns L sorted in non-decreasing order

// base case:
If the list L has 0 or 1 element, return it as sorted
// divide:
Otherwise, let split L down the middle into two halves Q & R
recursively sort Q by calling mergeSort( Q )
recursively sort R by calling mergeSort( R )
// conquer: how do we produce sorted L out of sorted Q, R?
reset L to be empty
while( both Q and R have elements left )
  compare the front (smallest) elements of Q and R
  remove the smallest of the 2 from its list and add it to L
(why is this element the next-smallest element of L?)
  add the rest of the remaining list to the end of L
return L, the sorted list

If T(n) represents the running time of this method on a list of length n, then

T(n) = 2T(n/2) + O(n), with initial condition T(1)=1

Wednesday, September 19, 12

Recursive Backtracking

Recursive backtracking is a recursive technique that uses recursion to try all possible solutions of something (i.e. brute force)

Possible solutions of something (e.g. finite number of places)

Recursive Backtracking

Example

If there are no elements, return the sorted list

mergeSort( list L ) // returns L sorted in non-decreasing order

Wednesday, September 19, 12

Recursive Example

What is our recursive case?

If T(n) represents the running time of this method on a list of length n, then

T(n) = T(n/2) + O(n), with initial condition T(1)=1

Wednesday, September 19, 12

"Divide and Conquer" Example

"Divide and Conquer" Example

Recursive Example

Recursive Example
Recursive Backtracking

Another example of recursive backtracking is the 8 queens problem: given an 8x8 chessboard, place 8 queens such that no queen threatens another.

Our recursive backtracking solution will basically exhaustively try all possible placements of all queens until it finds one that works. In particular,

```
placeQueen( col ) --- place the queen in this column, assuming 0..col-1 done
if( col == 8 ) then we're DONE! queens 0 through 7 already placed.
while( there are rows left to try that jibe with columns 0..col-1 )
try placing this column's queen in the current row, and recursively try
  to place the remaining queens (placeQueen( col+1 ))
if this works, return DONE.
return NO LUCK
```

Dynamic Programming

We can enumerate all length-n binary numbers using recursive backtracking:

```
printBinary( n ) --- print out all length-n binary numbers
if( n == 0 ) print nothing, return
print 0 + printBinary( n-1 )
print 1 + printBinary( n-1 )
return
```

Beware!

Recursion is not always your friend!

Consider our old pal, the Fibonacci numbers. The recursive definition is

\[
fib( n ) = fib( n-1 ) + fib( n-2 )
\]

\[
fib( 0 ) = 1
\]

\[
fib( 1 ) = 1
\]

Which translates into Java as follows:

```
public long Fib( int n ) {
  if( n <= 1 )         return 1;
  return Fib( n-1 ) + Fib( n-2 );
}
```

This certainly correctly computes the Fibonacci numbers. So what's the prob?

Consider Fib( 6 ) and the tree of recursive calls. It ain't pretty; there is a lot of
duplicate work here: we should really never duplicate work by solving the
same problem multiple times in recursive calls. It's much better to work your way up
from the base cases and avoid the recursion until you need it. (This is often how you compute by hand.)

To avoid duplication, "unwind" the recursion:

```
public long Fib( int n ) {
  if( n <= 1 )         return 1;
  int[] fibs = new int[n];
fibs[0] = 1;
fibs[1] = 1;
for( int i=2; i < n; i++ ) {
  fibs[i] = fibs[i-1] + fibs[i-2];
}
return fibs[n-1];
}
```

For example, to compute fib(6), start with 0, 1, 2, 3, 5, 8, ... until get to the 6th value. For example, to compute fib(8)
we should really never duplicate work by solving the same problem multiple times in recursive calls. Instead, we can compute the value we need (this is often how you compute by hand). Start from the base cases and work your way up through the recursive cases until you reach the case you need. This is often how you compute by hand.

```
printBinary( n ) --- print out all length-n binary numbers using recursive backtracking
```

We can enumerate all length-n binary numbers using recursive backtracking:

```
print 0 + printBinary( n-1 )
print 1 + printBinary( n-1 )
```

Recursion backtracking
Dynamic Programming

Dynamic programming is an iterative (non-recursive) technique
• that is based on a recursive formula
so it is improving the use of recursion...
• that is based on a recursive formula

This way you avoid duplicating any work with duplicate recursive calls

We'll talk more about dynamic programming later in the course when we talk
about "shortest paths." Then we talk a lot about it in CS 280 (Algorithms).

We'll talk more about dynamic programming later in the course (when we talk
about shortest paths.) Then we talk a lot about it in CS 280 (algorithms).