CS 151
Recursion (Review!)
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. Prelabs 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. It is worth your while, and will make debugging your code so much easier.
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:

\[
\text{sum}( \, n \, ) = n + \text{sum}( \, n-1 \, )
\]

We can easily translate this definition into code:

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

What is wrong with this code?
It never terminates. We fix this by establishing a base case:

\[
\text{sum}( \, 0 \, ) = 0
\]

And add a corresponding if-statement to our method: if \((n==0)\) return 0;
Fundamental Rules of Recursion

Every recursive method needs
• a base case (or base cases, if necessary) –
  an instance that can be solved without recursion
• a recursive case –
  one or more recursive calls that make progress towards the base case

If you don’t make progress towards the base case, you will recurse forever.
If you make progress, but don’t have a base case, you will never “bottom out”.
Recursion Example

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

```java
// 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?
```

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

$$T(n) = T(n/2) + O(1), \text{ with initial condition } T(1)=1$$
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

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

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

For example, if we are trying to find a prize that is hidden in a maze, then one solution is to try going down one path as far as you can (“trying” all locations along the way). If you don’t find the prize, you “backtrack” as little as possible until you find an unexplored branch, at which point you search for the prize down that path.

\[
\text{findPrize}( \text{path} ) = \\
\quad \text{go down path as far as you can} \\
\quad \text{if you find the puzzle, return FOUND IT} \\
\quad \text{while( you’re not back at the start )} \\
\quad \quad \text{back up one spot} \\
\quad \quad \text{while there are unexplored branches at this location,} \\
\quad \quad \quad \text{if findPrize( this branch ), return FOUND IT} \\
\quad \text{return NO LUCK}
\]
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
Recursive Backtracking

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

\[
\begin{align*}
\text{fib}( n ) &= \text{fib}( n-1 ) + \text{fib}( n-2 ) \\
\text{fib}( 0 ) &= 1 \\
\text{fib}( 1 ) &= 1
\end{align*}
\]

Which translates into Java as follows:

```java
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 \( \text{Fib}( 6 ) \) 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. Its run-time is also horrendous.
To avoid duplication, “unwind” the recursion.

Start from the base cases and work up through the recursive cases until you compute the value you need. (This is often how you compute by hand!)

For example, to compute Fib(n), start with 1, 1, 2, 3, 5, 8,... until get to the nth value. In code:

```java
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];
}
```

What is the running time of this method? (Hint: much better!)
Dynamic Programming

Dynamic programming is an iterative (non-recursive) technique
• that is based on a recursive formula —
  so it is tempting to use recursion...
• but you use a table / array to compute the values from the bottom-up —
  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).