# Recursion

CIS*2520 Summer 2006

## Overview

- An alternative to iteration
- General idea:
  - a subroutine, or collection of sub-routines, is defined in terms of itself
- Simple recursion
  - – problem defined in terms of itself
- Mutual recursion
  - – a cycle of problems involving one another



## Overview

- Contrast:
  - *iterative method of solving a problem*
    - –solution is a sequence of steps or sub-problems
    - –if one step needs to be performed multiple times, introduce a loop
  - *recursive method of solving a problem*
    - –solution is a sequence of sub-problems that are the ***same*** as (or defined in terms of) the original problem
    - –sub-problems are solved by recursive calls



## Recursion

- Recursion is a tool to allow the programmer to concentrate on the ***key step*** of an algorithm without having (initially) to worry about coupling that step with all the others
  - "how can this problem be divided into parts?"
  - "how will the key step in the middle be done?"
  - once you have a small simple step toward the solution
    - – ask whether the remainder of the problem can be done in the same or a similar way
    - – modify your method if necessary to be sufficiently general.

## Basis for Recursion

- **KEY STEP**
  - essential partial step
  - basis for, or applied to, the result of key step applied to a sub-problem
- **STOPPING RULE**
  - must have a sub-problem state at which point the problem (recursion) stops
  - ideally the "simplest" case you can handle directly
  - often when size is 0 or 1



## Thinking Recursively

- Any time you have a problem that can be defined in terms of solving a smaller/simpler version of itself, possibly with some other action applied at each step, you have a candidate for recursion
- Must be careful specifying order, i.e.
  - operation, recursion
  - recursion, operation
  - etc.
- Representing recursion
  - tree diagrams
  - execution flow (activation records)



## Implementing Recursion

- A subroutine calls itself
  - or calls a subroutine that eventually calls the original function again (simple vs. mutual recursion)
- Each instance of a function is unique
  - all local variables from previous call are saved, and new copies put in place
    - i.e. a new activation record/stack frame is created, *as is the case with any other function call*
  - not really anything special
    - in agreement with conventions associated with non-recursive calls
  - note: there is the potential for stack overflow



## Example: *Printing N Stars*

- Solution:
  - output N stars by printing a star, then printing N-1 stars
  - STOP: when N = 0

```
void writeStars(int n)
{
    if (n > 0)
    {
        printf("*");
        writeStars(n-1);
    }

} /* writeStars */
```

## Example: *Reversing a Line*

- Solution:
  - reverse a line of input by reading a character, reversing the remainder of the line, then printing the character
  - STOP: empty line
- Note: reversing a string is similar

```
void reverseLine(FILE *fp_in, FILE *fp_out)
{
    int ch;
    if ((ch = fgetc(fp_in)) >= 0)
    {
        reverse(fp_in, fp_out);
        fprintf(fp_out,"%c" (char)ch);
    }

} /* reverseLine */
```



## Example: *Factorials (N!)*

- Solution:
  - compute the factorial of *n* by multiplying *n* by the factorial of *n-1*
  - STOP: n = 0 (where 0! = 1)

```
int factorial(int n)
{
    if (n <= 0)
       return(1);
    else
        return(n * factorial(n-1));

} /* factorial */
```



## Examples (cont.)

- Other obvious candidates for recursion:
  - power
    - i.e. $x^n$
  - list traversal
  - fibonacci sequence
    - $f(1) = 1$
    - $f(2) = 1$
    - $f(i) = f(i-1) + f(i-2), i > 2$
- Consider how you would specify and implement these problems recursively

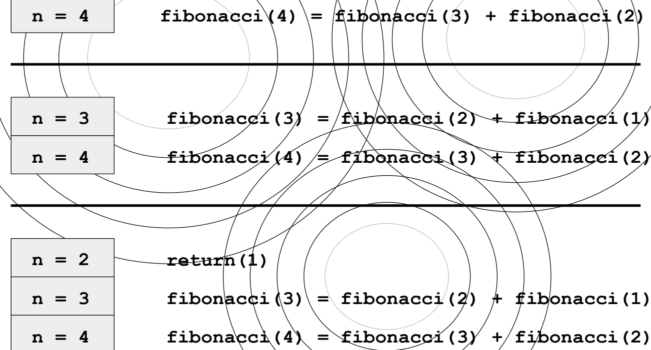


## More Sophisticated Recursion

- Everything we have seen to this point is relatively straightforward (read: trivial)
- Consider the fibonacci numbers:

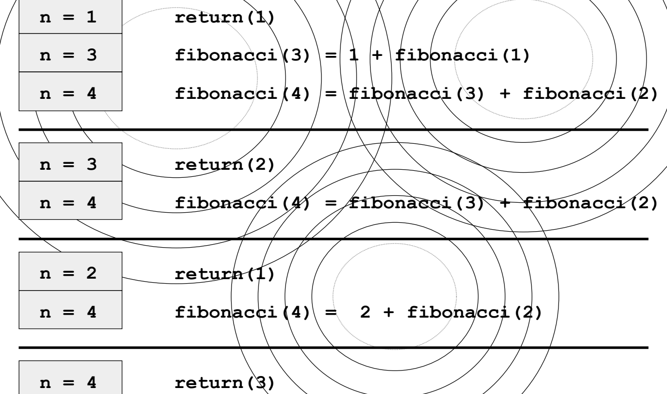
```
int fibonacci(int n)
{
    if (n <= 2)
       return(1);
    else
        return(fibonacci(n-1) + fibonacci(n-2));

} /* fibonacci */
```

## Example: *Fibonacci Numbers*



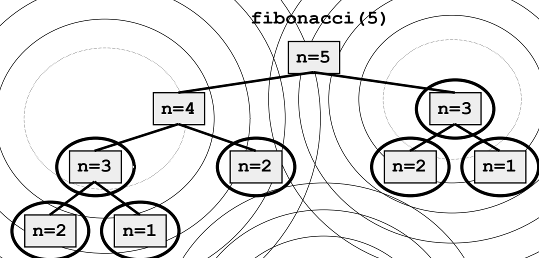



## Example: *Fibonacci Numbers* (cont.)






## Fibonacci Implementation




- Consider the execution tree for recursive calculation of fibonacci numbers
  - is this actually a reasonable method for computing them?
- Answer: a resounding NO
  - we are recomputing the same value over and over needlessly
  - a simple iterative method is much, much more efficient



## Example: *List Traversal*

- e.g. Sequential Search
  - visit head then traverse remainder of list (stop on empty list)

```
DataType *sequentialSearch(Element *head, KeyType *target)
{
    if (head != null)
    {
        if (isEqual(target, getKey(head)))
            return(getData(head));
        else
            return(sequentialSearch(getNext(head), target));
    }

    return(null);

} /* sequentialSearch */
```

## Comments on Recursive Searching

- Contrast the recursive version with the iterative one
- Three important concepts illustrated here:
  1. importance of correctly implementing stopping criterion
  2. references/pointers and sequences lend themselves well to recursion
  3. “instantiation” of variables (i.e. DataType in the example)
- Often it is simplicity of implementation that recursion is buying us
- What about binary search?



## Backtracking Search

- Application of recursion useful in solving a large class of problems
- Many problems can be solved by searching for a solution among a "space" of possible answers
- problem is defined in terms of paths that may eventually lead to a solution



## Backtracking Search (cont.)

- Terminology
  - <u>Dead End</u>
    - a path that is known ***not*** to lead to a solution
  - <u>Search Space</u>
    - a set of possible right answers to be explored
  - <u>Backtracking</u>
    - a technique whereby the search space is explored by systematically trying each possible path, backing up to try another whenever a dead end is encountered



## Backtracking Search (cont.)

- Each path is followed until either a solution is found, or it is discovered to be a dead end
- Each recursive call represents one step along a path
- One of two things can happen during procedure execution:
  1. *find a solution*
     - every calling procedure should be told about it
  2. *fails, finds a dead end*
     - try a different path
- e.g. Searching a maze

## Searching a Maze

- Solution:
  - find the path to the exit by finding the path to the exit from the cell ***above*** or ***right*** or ***below*** or ***left*** of the current cell (which is then also on the path)
  - STOP: cell is outside maze, a wall, already on path or the exit
- Consider this modification:
  - find all paths



## Maze Search Implementation

```
int findPath(int **maze, int curi, int curj)
{
    if (...)
        /* check stopping conditions (off edges, current loc
           is wall or already on path, found exit) */
    else
    {
        maze[curi][curj] = PATH;
        if (findPath(curi-1,curj))      return(1);
        else if (findpath(curi,curj+1)) return(1);
        else if (findpath(curi+1,curj)) return(1);
        else if (findpath(curi,curj-1)) return(1);
        else
        {
            maze[curi][curj] = EMPTY;
            return(0);
        }
    }
} /* findPath */
```



## 8 Queens Problem

- Problem:
  - place 8 queens on a chessboard so that no queen attacks another
- Recursive backtracking solution
  - what "paths" will we follow?
    - some will lead to dead ends (failure)
    - some will lead to a goal
- Idea: a move consists of placing a queen in a row
  - there will be exactly one queen in each row
  - in each row, you place a queen in one of the 8 columns, then take a step forward by making a recursive call



## 8 Queens Problem Example

## 8 Queens Problem Implementation

- What is the base case?
- When do we stop recursing?
  - solution is found
  - no more possible column positions to try for a given row (dead-end)



## 8 Queens Problem Solution Sketch

```
int 8Queens(…. int row)
{
    int column;

    if (row >= 8)
        return(1);
    else
    {
        for (each column in row)
            if (it is safe to place the queen)
            {
                place the queen at (row,column)
                if (!8Queens(row+1))
                    remove queen from (row,column)
            }
    }

} /* 8Queens */
```



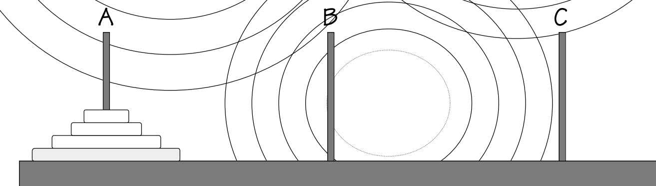
## Comments on the 8 Queens Problem

- Issues:
  - the solution sketch provided is missing detail
    - not indicating failure (dead ends: `if (!8Queens()) ...` implies call can return false)
    - also will not terminate the for loop when a solution is found
      - as is it would just continue---i.e. find all solutions
      - better choice might be a while loop
  - how do we know when it is safe to place the queen?
  - efficiently tracking this information?
  - additional data structures?
  - output solution at the end?
  - finding one vs. finding all solutions?
  - *there are easier and harder ways to do this*...requires some thought



## Towers of Hanoi

- 3 pegs and a set of concentric rings
- Problem:
  - move the disks from peg A to peg C according to the following rules:
    1. when a disk is moved it must be placed on one of the 3 pegs
    2. only one disk may be moved at a time, and it must be the top disc on one of the pegs
    3. a larger disk cannot be placed on top of a smaller one



## Towers of Hanoi (cont.)

- Easy to solve by trial & error for small number of disks
- Seems to become progressively more complicated as number of disks increases
- This is because the solution is inherently recursive
  - solving for N requires solving for N-1, etc. which involves many repeating recursive steps which human beings tend to be quite poor at



## Towers of Hanoi (cont.)

- Note:
  - 1 disk
    - solution is trivial (move it from A->C)
  - 2 disks
    - move smallest from A->B, largest from A->C, then smallest from B->C
    - i.e. use of B as an intermediate peg
  - 3 disks
    - as for 2 but move top 2 from A->B using C as intermediate, move largest from A->C, then move top to from B->C using A as intermediate
  - 4 disks
    - as for 3 but moving top 3 from A->B using C, move largest from A->C, them move top 3 from B->C using A



## Towers of Hanoi Solution

- Solution:
  - assuming a solution exists for n-1 disks, a solution for n disks can be obtained recursively by making use of an intermediate peg:
    1. move topmost n-1 disk from peg A to peg B using peg C as an intermediate peg
    2. move the large disk remaining on peg A to peg C
    3. move the n-1 disks from peg B to peg C using peg A as an intermediate peg
  - STOP: when there is one disk (N == 1) - move it from peg A to peg C



## Towers of Hanoi Implementation

```
void move(int n, char start, char intr, char end)
{
    if (n == 1)
        printf("move %c to %c\n", start,end);
    else
    {
        move(n-1, start, end, intr);
        move(1, start, ' ', end);
        move(n-1, intr, start, end);
    }

} /* move */
```

- Note: this is not all all trivial to do iteratively
  - consider how you would do this

## Mutual Recursion

- Consider a subroutine that calls another, which then calls the first
  - this new activation of the first may again call the second, which may call the first...
- This is an example of mutual recursion
  - subprograms that call themselves indirectly (i.e. through some intermediate subprogram(s)
- Contrast with data indirection



## Mutual Recursion Implementation Issues

- Declaration order (of subroutines) may be important
  - if subroutine *foo* calls subroutine *bar*, and *foo* is declared first, then bar can call *foo*, but how can *foo* call *bar* if it hasn't been defined at that point?
    - this is ultimately a compiler issue, and most languages have to deal with this issue of forward references on some level
  - ***C***: single-pass compile using prototypes
  - ***Java***: multi-pass compile (order of declarations irrelevant)
  - ***Pascal***: single-pass compile using FORWARD directives
  - etc.

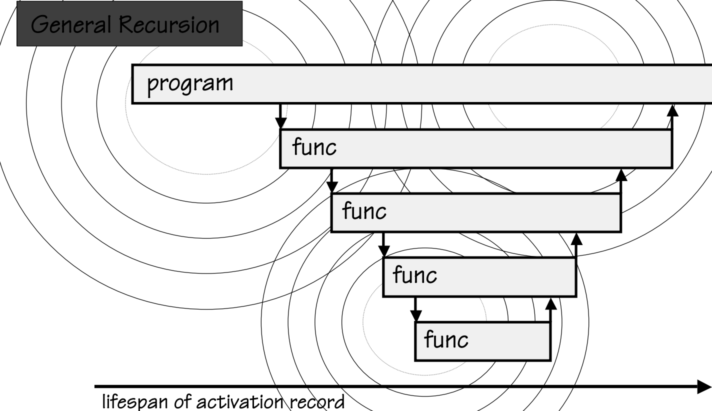


## Removal of Tail Recursion

- ***Tail Recursion*** is not necessary
  - recall: local variables allocated on the stack as recursive calls are initiated
  - when recursive call finishes, the values from the previous activation are restored
  - this is pointless however as the next thing to happen is the subroutine exists, and these local variables are ignored anyway
- If the last executed statement of a subroutine is a recursive call to itself, then this call can be eliminated
  - changing the values of the calling parameters to those specified in the recursive call (i.e. assignment)
  - repeat the entire function (i.e. introduce a loop)

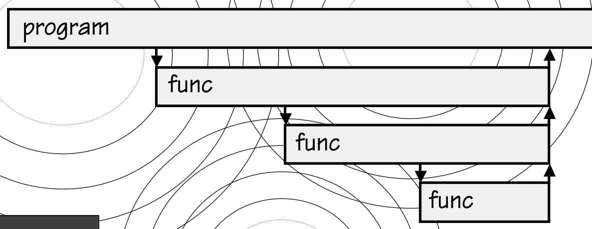


## Tail Recursion Example



## Tail Recursion Example

Tail Recursion




Equivalent To: