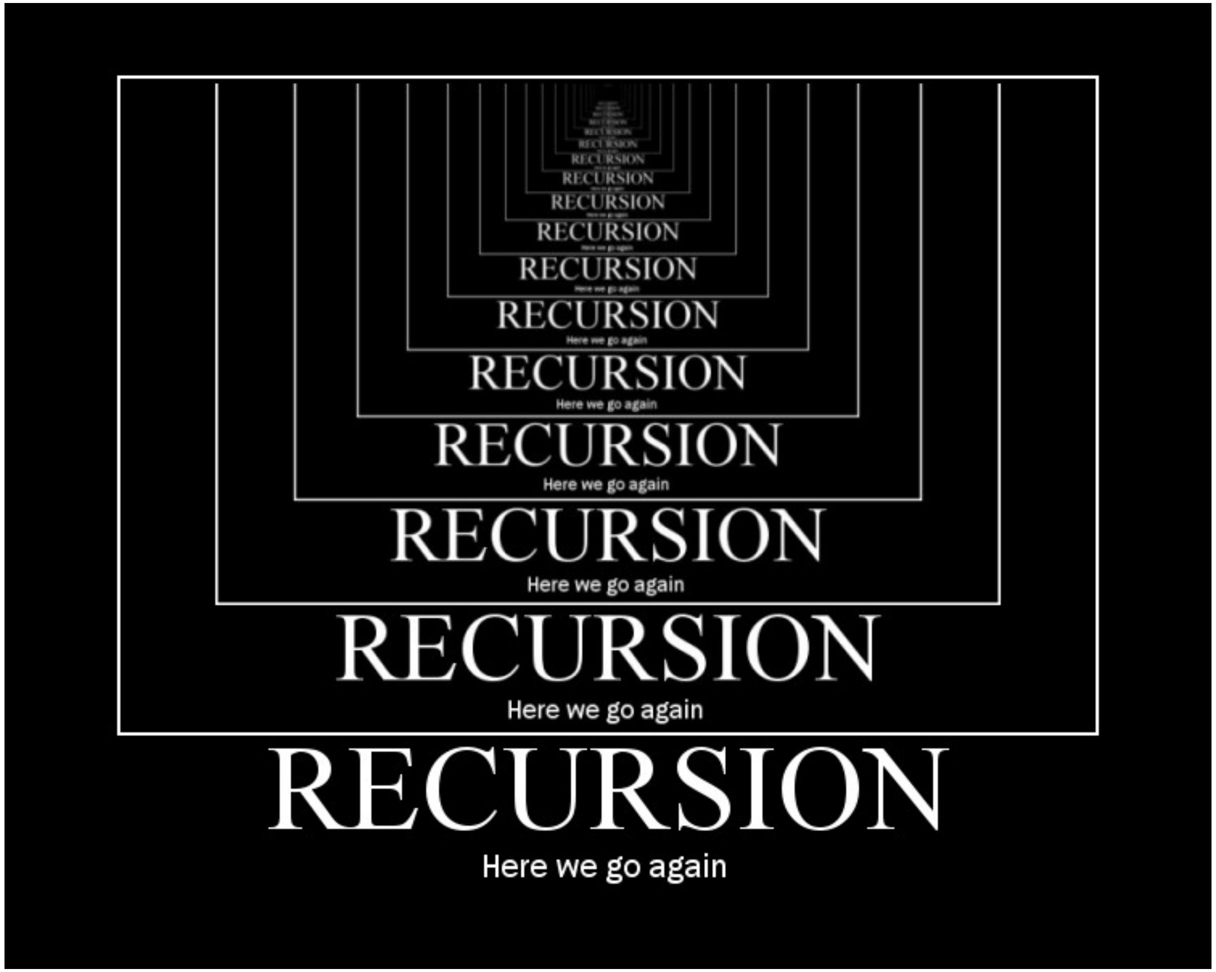
Problem Solving & Software Development

Lecture 3. Recursive approach

Recursive approach



In order to understand recursion, you must first understand recursion.

Recursive approach

Recursion - a method of defining a function in terms of its own definition.

Why write a method that calls itself?

- Recursion is a good problem solving approach.
- Recursive solutions are often shorter.
- Solve a problem by reducing the problem to smaller sub-problems; this results in recursive calls.

However

- Good recursive solutions may be more difficult to design and test.
- Recursive calls can result in an infinite loop of calls

Recursive algorithms

To solve a problem recursively

- break into smaller problems;
- solve sub-problems recursively;
- assemble sub-solutions.

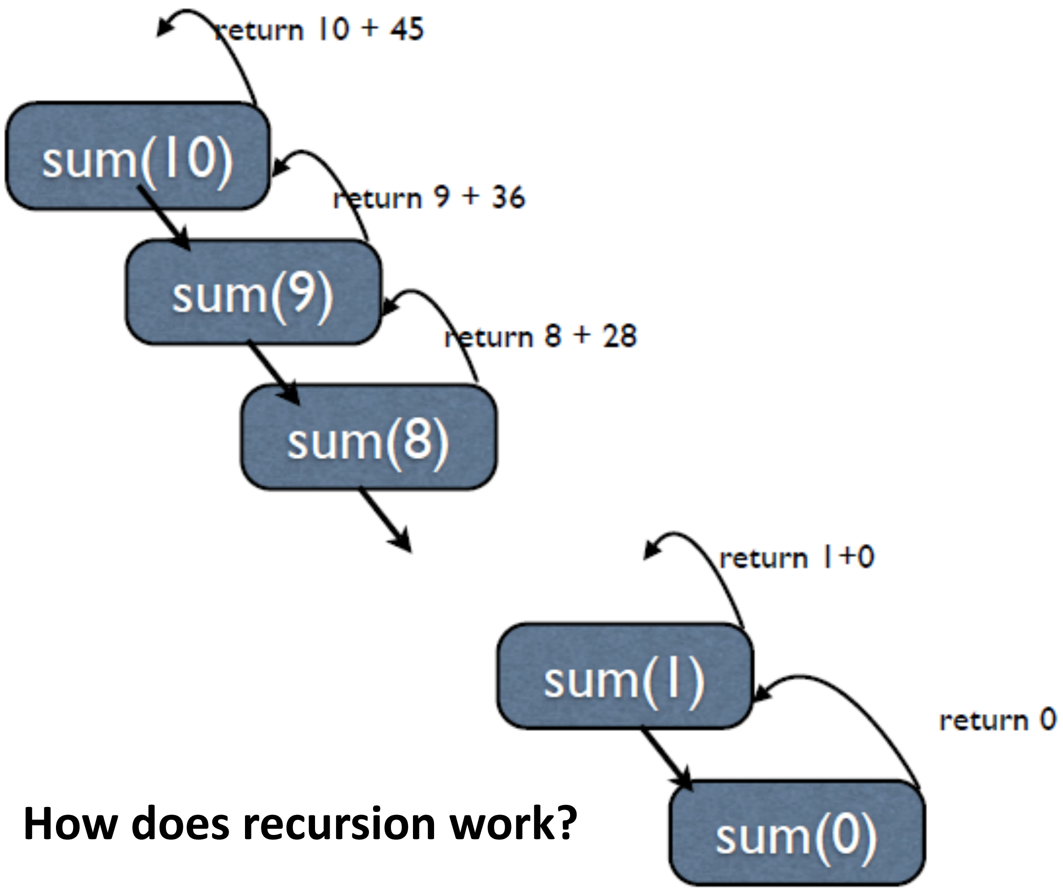
```
recursive-algorithm(input) {
// base-case
if (isSmallEnough(input))
   compute the solution and return it
else
// recursive case
   break input into simpler instances input1, input 2,...
   solution1 = recursive-algorithm(input1)
   solution2 = recursive-algorithm(input2)
   ...
   figure out solution to this problem from solution1, solution2,...
   return solution
}
```

Simple Example

Write a function that computes the sum of numbers from 0 to n
(i) using a loop; (ii) recursively.

```
// with a loop
int sum (int n) {
   int s = 0;
   for (int i=0; i<=n; i++)
          s+= i;
   return s;
}
```

```
// recursively
int sum (int n) {
   // base case
   if (n == 0) return 0;
   // else
   return n + sum(n-1);
}
```



How does recursion work?

How it works

- Recursion is no different than a function call.
- The system keeps track of the sequence of method calls that have been started but not finished yet (active calls). **Order matters!**

Recursion pitfalls:

- **Missed base-case**: infinite recursion, stack overflow.
- **No convergence**: solve recursively a problem that is not simpler than the original one.
- Recursion has an ***overhead*** (keep track of all active frames). Modern compilers can often optimize the code and eliminate recursion.

Unless you write super-duper optimized code, recursion is good.

Fibonacci: recursive vs iterative version

Recursive is not always better!

```
// Fibonacci: recursive version
int Fibonacci_R(int n) {
   if(n<=0) return 0;
   else if(n==1) return 1;
   else return Fibonacci_R(n-1)+Fibonacci_R(n-2);
}
```

This takes $O(2^n)$ steps! Unusable for large n.

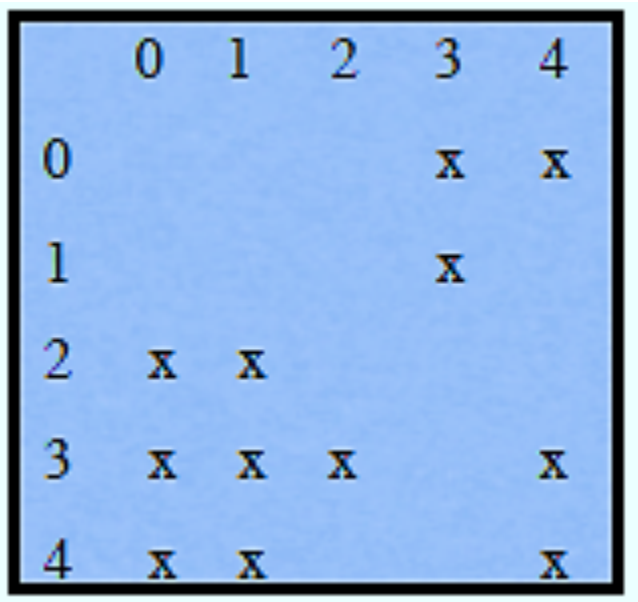
```
// Fibonacci: iterative version
int Fibonacci_I(int n) {
   int fib[] = {0,1,1};
   for(int i=2; i<=n; i++) {
      fib[i%3] = fib[(i-1)%3] + fib[(i-2)%3];
   }
   return fib[n%3];
}
```

This iterative approach is “linear”; it takes $O(n)$ steps.

Example: Blob Check

Problem: you have a 2-dimensional grid of cells, each of which may be filled or empty. Filled cells that are connected form a “blob” (for lack of a better word).

Write a recursive method that returns the size of the blob containing a specified cell (i,j).



BlobCount(0,3) = 3

BlobCount(0,4) = 3

BlobCount(3,4) = 2

BlobCount(4,0) = 7

Example: Blob Check

Solution: essentially you need to check the current cell,
its neighbors, the neighbors of its neighbors, and so on.

When calling BlobCheck(i,j)

- (i,j) may be outside of grid
- (i,j) may be EMPTY
- (i,j) may be FILLED

When you write a recursive method, always start from the base case

Given a call to BlobCkeck(i,j): when is there no need for recursion,
and the function can return the answer immediately?

Example: Blob Check

Solution: essentially you need to check the current cell,
its neighbors, the neighbors of its neighbors, and so on.

When calling BlobCheck(i,j)

- (i,j) may be outside of grid
- (i,j) may be EMPTY
- (i,j) may be FILLED

When you write a recursive method, always start from the base case

Given a call to BlobCkeck(i,j): when is there no need for recursion,
and the function can return the answer immediately?

- (i,j) is outside grid
- (i,j) is EMPTY

Example: Blob Check

```
blobCheck(i,j):
   if (i,j) is FILLED  -> add 1 (for the current cell)
                       -> count its 8 neighbors
```

```
// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
   for (k = -1; k <= 1; k++)
      if (l==0 && k==0) continue;  // skip of middle cell
      if (grid[i+l][j+k] == FILLED) blobc++; // count neighbors that are FILLED
```

Does this work?

Example: Blob Check

```
blobCheck(i,j):
   if (i,j) is FILLED  -> add 1 (for the current cell)
                       -> count its 8 neighbors

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
   for (k = -1; k <= 1; k++)
      if (l==0 && k==0) continue;  // skip of middle cell
      if (grid[i+l][j+k] == FILLED) blobc++; // count neighbors that are FILLED
```

- It does not count the neighbors of the neighbors, and their neighbors, and so on.
- Instead of adding +1 for each neighbor that is filled, need to count its blob recursively.

Example: Blob Check

```
blobCheck(i,j):
   if (i,j) is FILLED  -> add 1 (for the current cell)
                       -> count blobs of its 8 neighbors
```

```
// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
   for (k = -1; k <= 1; k++)
      if (l==0 && k==0) continue;  // skip of middle cell
      blobc += blobCheck(i+k, j+l);
```

Does this work?

Example: Blob Check

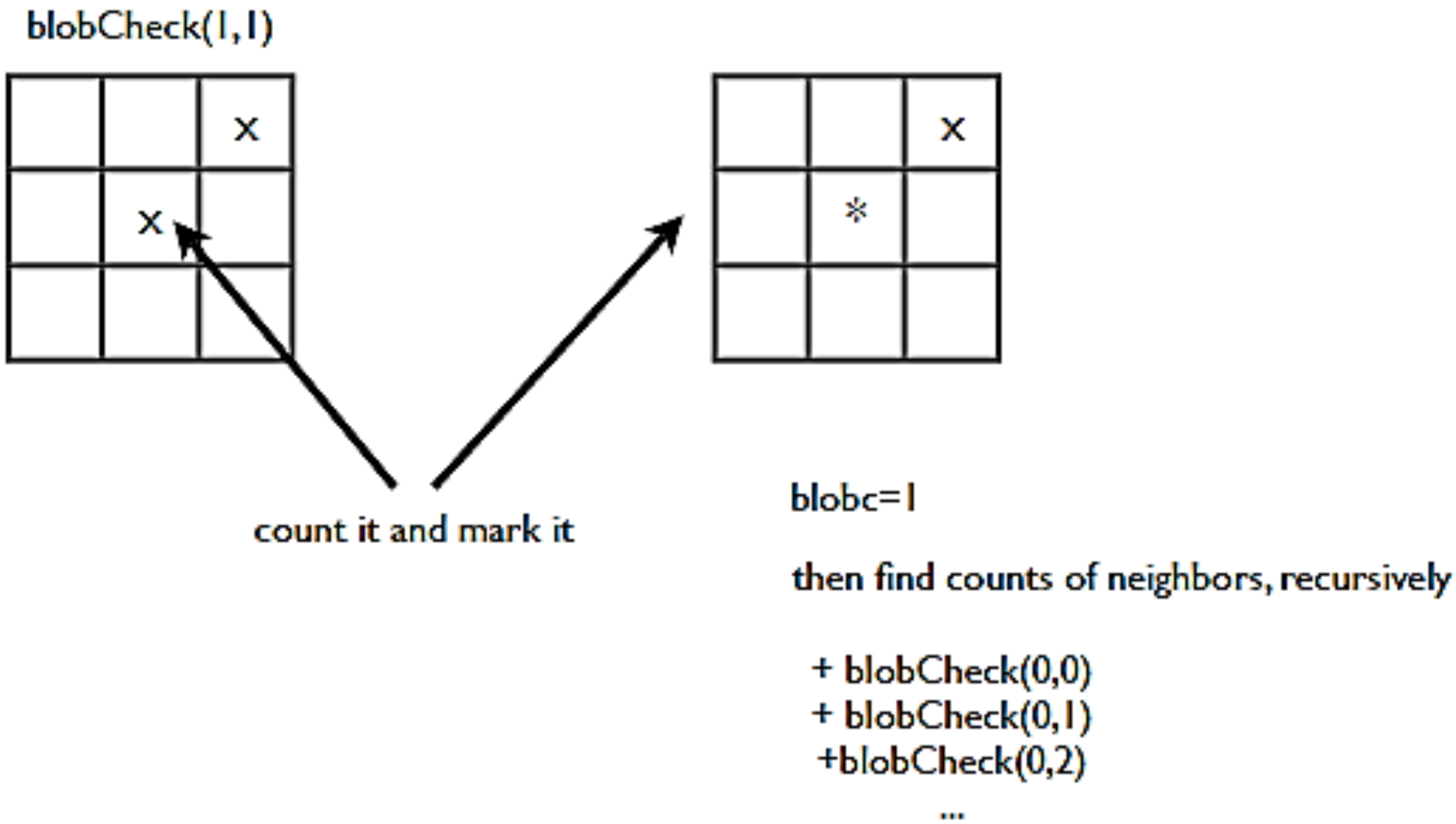
```
blobCheck(i,j):
   if (i,j) is FILLED  -> add 1 (for the current cell)
                       -> count blobs of its 8 neighbors

// first check base cases
if (outsideGrid(i,j)) return 0;
if (grid[i][j] != FILLED) return 0;
blobc = 1
for (l = -1; l <= 1; l++)
   for (k = -1; k <= 1; k++)
      if (l==0 && k==0) continue;  // skip of middle cell
      blobc += blobCheck(i+k, j+l);
```

- **Example:** blobCheck(1,1)
  - blobCount(1,1) calls blobCount(0,2)
  - blobCount(0,2) calls blobCount(1,1)

- **Problem:** infinite recursion because of the multiple counting of the same cell.

Example: Blob Check

Idea: once you count a cell, mark it so that it is not counted again by its neighbors.



Example: Blob Check (Correctness)

- **blobCheck(i,j) works correctly if the cell (i,j) is not filled**
- **blobCheck(i,j) works correctly if the cell (i,j) is filled**
  - mark the cell
  - the blob of this cell is 1 plus the blobCheck of all neighbors
  - because the cell is marked, the neighbors will not see it as FILLED

    => a cell is counted only once
- **Why does this stop?**
  - blobCheck(i,j) will generate recursive calls to neighbors
  - recursive calls are generated only if the cell is FILLED
  - when a cell is marked, it is NOT FILLED anymore,
    so the size of the blob of filled cells is one smaller

    => the blob when calling blobCheck(neighbor of i,j) is smaller that blobCheck(i,j)

Problem type: Recursion

- How to identify if a problem can be solved recursively?
  - Problems in which the solution "builds up"
  - **Multiple Related Decisions**
    - 1 decision k elements - have a case statement (k)
    - 2 decisions k elements - a nested loop (k^2)
    - 14 decisions – cannot use brute force

    But we can take one decision at a time

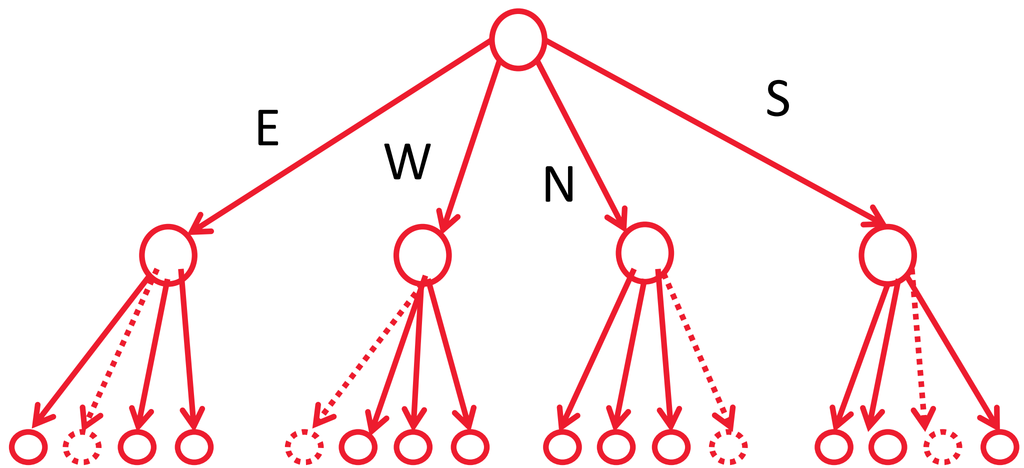
Recursion – call tree

CrazyBot

4 choices per step

  4-ary tree

Depth = number of steps



Thinking recursively

Finding the recursive structure of the problem is the hard part.

- **Common patterns:**
  - divide in half, solve one half
  - divide in sub-problems, solve each sub-problem recursively, “merge”
  - solve one or several problems of size n-1
  - process first element, recurse on the remaining problem
- **Recursion**
  - functional: function computes and returns result
  - procedural: no return result (function returns void)
    The task is accomplished during the recursive calls.
- **Recursion**
  - exhaustive
  - non-exhaustive: stops early