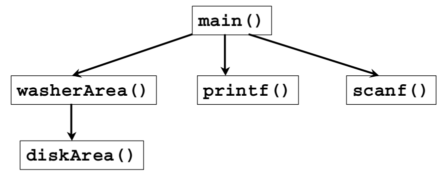
CSE 142
Programming I

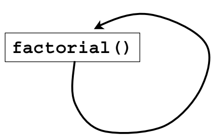
Recursion

Remember Static Call Graphs



What is Recursion

- A program contains recursion if there is a cycle in the static call graph
  - We call the functions involved recursive



Huh?!

- You probably have a few questions:
  - How can this work?
  - Why would I want to do this?
  - Isn't there another way?

Factorial Function

- How do we think of a factorial?
  - "N factorial is 1*2*3*… all the way up to N"
  - "N factorial is N times N-1 factorial"

Iterative Factorial

```
int factorial(int n){
  int product, i;

  product = 1;
  for (i=1; i<=n; i++)
     product = product * i;

  return product;
}
```

Recursive Factorial

```
int factorial(int n){
  int product;

  if (n <= 1) product = 1;
  else product = n * factorial(n-1);

  return product;
}
```

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-7

Yikes!

- Tracing the recursion is easy as long as we keep in mind our function basics:
  - ➢Parameters are local to a function—created when the function starts, and destroyed when the function ends
  - ➢Parameters are initialized by *copying* the arguments into the parameters

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-8

Important Point!

- EVERY recursion must have a base case and an iterative case
  - ➢The iterative case is what makes our program loop—without it it would not be a recursion
  - ➢The base case is what stops our program from looping—without it, we will loop forever!

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-9

Another Example

- Fibonacci numbers
- Recall: each Fibonacci number is the sum of the previous two

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, etc.

- Problem: Find the nth Fibonacci number

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-10

We Can Do This Iteratively

```
int fibonacci(int n){
  int prev, cur, next;

  prev = 1; cur = 1;
  for (n = n-1; n > 0; n--){
     next = prev + cur;
     prev = cur;
     cur = next;
  }
  return cur;
}
```

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-11

We *Can* Do This Recursively

```
int fibonacci(int n){
  int result;

  if (n <= 1) result = 1;
  else result = fibonacci(n-1) +
     fibonacci(n-2);

  return result;
}
```

9. August, 2000 CSE 142 Summer 2000 — Isaac Kunen R-12

Let's Examine Our Choices

- Which solution is easier to understand?
- Which solution makes more work for the computer?
- Which method makes sense for each of our two problems?

Warning!

The Surgeon General has determined that computing the Fibonacci sequence recursively is hazardous to your computer's performance.

What *Can* We Do Recursively?

- It turns out that anything we can do iteratively we can do recursively.
- Likewise, anything we can do recursively, we can do iteratively.
- So, why do we care?

Why Would We Use Recursion?

- Some problems are much easier to formulate as a recursion, so it makes our life easier (we'll see examples)
- Some programming languages don't have loops!
  - Some don't even have variables! (!!!!)

More Problems With Recursion

- It turns out that calling a function takes a *little* bit of time
- It's actually slightly more efficient to do things iteratively than recursively!
- So...

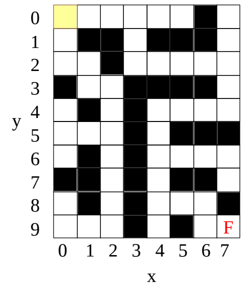
Why on Earth Would We Do It?

- Some problems are very hard to solve without recursion
  - In this case, we may give up a little bit of performance (not much!) and use a recursion
- It's fun! (Really, it is!)

How to Spot Recursion

- A good candidate for a recursive function is one in which the problem can be split apart into two smaller problems of the same kind
- Be careful! Fibonacci sounds like a good candidate by that criteria!

Example: Path planning



Problem: Can we get through the maze?

Store the board as an array

F = finish
X = blocked
‘ ’ = open

Base Cases

- If we are at the end (‘F’) then return true
- If we have nowhere to go, then return false

Recursive Cases

- What should we do if we aren’t blocked, and we aren’t at the end?
  - Pretend that the current space is blocked, and try to continue the path in every direction
  - If any one of those recursive cases succeed, then we succeed, otherwise we fail

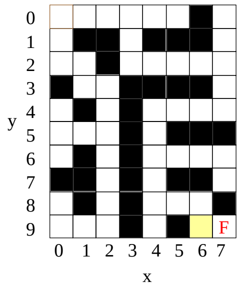
A Helper Function

```
int legalMove(char
  m[MAXX][MAXY], int x, int y){

  return (x >=0 && x < MAXX &&
          y >= 0 && y < MAXY &&
          m[x][y] != ‘X’);
}
```

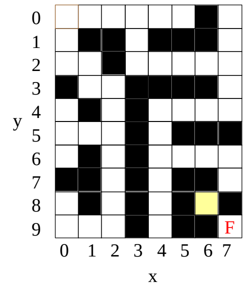

Elegant Solution

```
int isPath(char m[MAXX][MAXY], int x, int y){

  if (m[x][y] == ‘F’) return TRUE;

  m[x][y] = ‘X’;
  return (legalMove(m,x+1,y) &&
  isPath(m,x+1,y))
     ||(legalMove(m,x-1,y) && isPath(m,x-1,y))
     ||(legalMove(m,x,y+1) && isPath(m,x,y+1))
     ||(legalMove(m,x,y-1) && isPath(m,x,y-1));
}
```

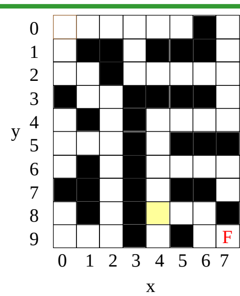
Example:



Example:



Example:



Summary:

- Recursion is a useful programming technique if used properly
  - Can make hard problems easier
  - Can make some problems worse
- Recursion requires you think differently about your program
  - Split up the problem into smaller pieces