CSE 142
Computer Programming I

Recursion

(Includes additional slides for Wi01)

Overview

Review
- Function calls in C

Concepts
- Recursive definitions and functions
- Base and recursive cases

Reading
- Read textbook sec. 10.1-10.3 & 10.7
- Optional: sec. 10.6 (Towers of Hanoi, a classic example)
- Skip sec. 10.4-10.5

Factorial Function

Factorial is an example of a mathematical function that is defined *recursively*, i.e., it is partly defined in terms of itself.

$$n! = \begin{cases} 1 & n \leq 1 \\ n*(n-1)! & otherwise \end{cases}$$

Factorial Revisited

We've already seen an implementation of factorial using a loop

```
int factorial ( int n ) {
    int product, i ;
    product = 1 ;
    for ( i = n ; i > 1 ; i = i - 1 ) {
        product = product * i ;
    }
    return product ;
}
```

1! is 1
2! is 1 * 2
3! is 1 * 2 * 3
4! Is 1 * 2 * 3 * 4
5! Is 1 * 2 * 3 * 4 * 5
. .

Factorial, Recursively

But we can use the recursive definition directly to get a different version

```
/* = n factorial – the product of the first
   n integers, 1 * 2 * 3 * 4 . . . * n  */
int factorial(int n){
    int result;
    if (n <= 1)
        result = 1;
    else
        result = n * factorial(n - 1);
    return result;
}
```


Trace

factorial(4) =
4* factorial(3) =
4* 3 * factorial(2) =
4* 3 * 2 * factorial(1) =
4* 3 * 2 * 1 =
4* 3 * 2 =
4* 6 = **24**

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

What is Recursion?

Definition: A function is recursive if it calls itself

```
int foo(int x) {
  ...
  y = foo(...);
  ...
}
```

How can this possibly work???

W-7

Function Calls

Answer: there's nothing new here!

Remember the steps for executing a function call in C:
- Allocate space for called function's parameters and local variables
- Initialize parameters
- Begin function execution

Recursive function calls work exactly the same way
- New set of parameters and local variables for each (recursive) call

W-8

Trace

main k 24

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

int main(void) {
   ...
   k = factorial(4);
   ...
}
```

W-9

Recursive & Base Cases

A recursive definition has two parts
- One or more recursive cases where the function calls itself
- One or more *base cases* that return a result without a recursive call

There *must* be at least one base case

Every recursive case *must* make progress towards a base case

Forgetting one of these rules is a frequent cause of errors with recursion

W-10

Recursive & Base Cases

Base case

Recursive case

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

W-11

Does This Run Forever?

Check:
- Includes a base case? Yes
- Recursive calls make progress? Hmmm…

```
int f (int x) {
   if (x == 1)
      return 1;
   else if (x % 2 == 0)
      return 1 + f(x/2);
   else
      return 1 + f(3*x + 1);
}
```

Answer: Not known!!! In tests, it always gets to the base case eventually, but nobody has been able to *prove* that this must be so!

W-12

3N + 1 function

f(5) = 1 + f(16) = 2 + f(8) = 3 + f(4)
= 4 + f(2) = 5 + f(1) = 6

f(7) = 1 + f(22) = 2 + f(11) = 3 + f(34)
= 4 + f(17) = 5 + f(52) = 6 + f(26)
= 7 + f(13) = 8 + f(40) = 9 + f(20)
= 10 + f(10) = 11 + f(5) = 12 + f(16)
= 13 + f(8) = 14 + f(4) = 15 + f(2)
= 16 + f(1) = 17

Iteration *vs.* Recursion

Turns out *any* iterative algorithm can be reworked to use recursion instead (and vice versa).

There are programming languages where recursion is the only choice(!)

Some algorithms are more naturally written with recursion

But *naïve* applications of recursion can be inefficient

Example: Array Sum

Problem: Write a function that returns the sum of a section of an integer arrray

Solution?

```
/* = sum of A[m]…A[n] */
int asum(int A[ ], int m, int n) {
  int k;
  int sum = 0;
  for (k = m; k <= n; k++) {
    sum = sum + A[k];
  }
  return sum;
}
```


Array Sum Thinking Recursively

A different way to think about this: We can use asum to calculate the sum of any section of the array, so…

```
/* = sum of A[m]…A[n] */
int asum(int A[ ], int m, int n) {
  …
  return A[m] + asum(A,m+1,n);
}
```

Any problems?

Answer: Need a base case

Otherwise, the recursion runs forever… Corrected version…

```
/* = sum of A[m]…A[n] */
int asum(int A[ ], int m, int n) {
  if (m > n) {
    return 0;
  } else {
    return A[m] + asum(A,m+1,n);
  }
}
```


When to Use Recursion?

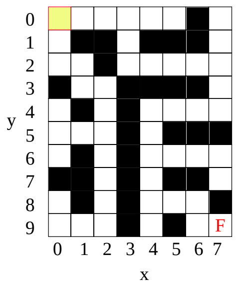
Problem has one or more simple cases

These have a straightforward nonrecursive solution, and:

Other cases can be redefined in terms of problems that are closer to simple cases

By repeating this redefinition process one gets to one of the simple cases

Example: Path planning




```
/* 'F' means finished!
   'X' means blocked
   ' ' means ok to move */
char maze[MAXX][MAXY];
int x =0, y=0;   /* start in yellow */
```

Unless blocked, can move up, down, left, right

Objective: determine if there is a path?

Simple Cases

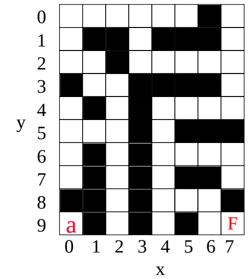
Suppose at x,y

If maze[x][y]=='F'

Then "yes!"

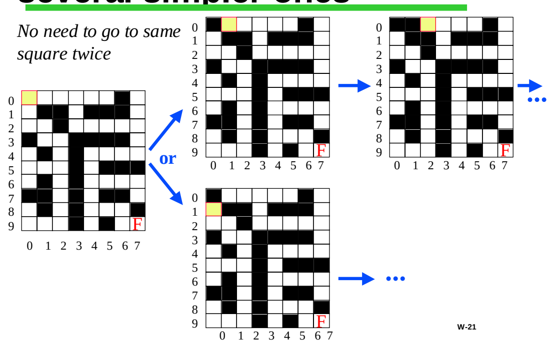
If no place to go

Then "no!"



Redefining a hard problem as several simpler ones

No need to go to same square twice



Helper function

```
/* Return true if <x,y> is a legal move
   given the maze, otherwise returns false */
int legal_mv (char m[MAXX ][MAXY],
                                    int x, int y) {
   return(x >= 0 && x < MAXX &&
            y >= 0 && y < MAXY &&
             m [x][y] != 'X');
}
```


Elegant Solution

```
/* Return true if there is a path from <x,y> to an element of maze
   containing 'F' otherwise returns false */
int is_path(char m[MAXX][MAXY ], int x, int y) {
   if (m [x][y] == 'F')
      return(TRUE);
   else {
      m[x][y] = 'X';
      return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
             (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
             (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
             (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
 }}
```

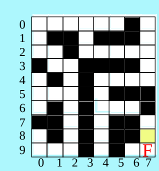

Example

is_path(maze, 7, 8)

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
          (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
          (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
          (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```

x: 7

y: 8



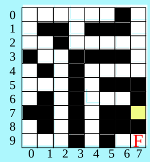
Example Cont

is_path(maze, 7, 7)

x 7

y 7

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```



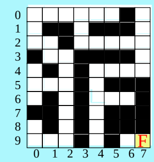
Example Cont

is_path(maze, 7, 9)

x 7

y 9

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```



Recursion Wrap-up

- **Recursion is a programming technique**
  - **It works because of the way function calls and local variables work**
- **Recursion is more than a programming technique**