CSE 142
Computer Programming I

Recursion

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

Overview

Review
- Function calls in C

Concepts
- Recursive definitions and functions
- Base and recursive cases

Factorial Function

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

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

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

```
/* Compute 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???

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

---

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);
   ...
}
```

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

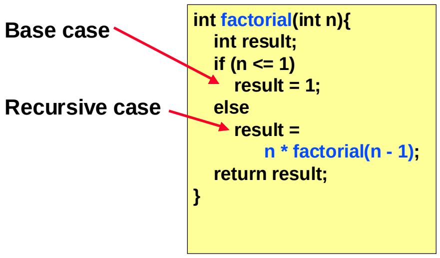
Forgetting one of these rules is a frequent cause of errors with recursion

---

Autumn 2000

Slides past this point not covered in both lecture sections

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;
}
```

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!

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

Recursive Binary Search

Binary Search
Recursive Algorithm
Iteration vs. Recursion

A Familiar Search Algorithm

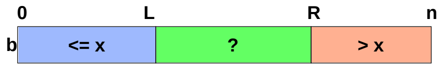
Binary search works if the array is sorted

1. Look for the target in the middle.
2. If you don't find it, you can ignore half of the array, and repeat the process with the other half.

Example: Find first page of pizza listings in the yellow pages

Let's solve this again, recursively

Binary Search Strategy



Values in b[0..L] <= x
Values in b[R..n-1] > x
Values in b[L+1..R-1] are unknown

mid = (L + R) / 2

Compare b[mid] and x

Replace either L or R by mid

Recursive Binary Search

Key idea – do a little bit of work, and make recursive call to do the rest

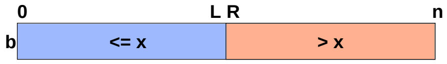
Binary search has value restricted to a range

Look at midpoint, and decide which half of the range is of interest

Use binary search to find value in reduced range. Recursion.

Base Case

No remaining unknown area:

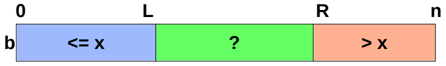


We recognize the base case when

L+1 == R

Recursive Case

Situation while searching



Step: Look at b[(L+R)/2]. Move L or R to the middle depending on test

Each recursive call is given L and R as parameters

The Search Function

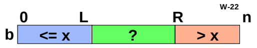
The original search problem called for a function with 3 parameters:

```
int bsearch (int b[ ], int n, int x);
```

Our recursive approach requires L and R as parameters

Let's call this function by a different name:

```
int bsearchHelper (int b[ ], int L, int R, int x) {
...
}
```



Recursive Search Function

```
int bsearchHelper (int a[ ], int L, int R, int x) {
   int mid;
   if (L+1 == R)       /*base case*/
        return L;

   mid = (L+R)/2;    /*recursive case*/
   if (a[mid] <= x)
        L = mid;
   else
        R = mid;
   return bsearchHelper(a, L, R, x);
}
```

Initialization Dilemma

The proper initial values for L and R are:

L = -1;

R = n;

These initializations cannot be inside the bsearchHelper function, since L and R are parameters!

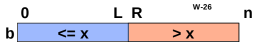
Termination Dilemma

After the base case is reached, we must make the final decision about what value to return: -1 if not found, L if found

This decision cannot be placed inside bsearchHelper!

Solution: A "Wrapper" Function

1. It sets the recursion in motion
   Calls the recursive function with the correct initial parameters
2. After the recursion completes, determines the correct final action



Non-Recursive Wrapper

```
int bsearch (int a[ ], int asize, int x) {
   int L = -1;
   int R = asize;

   L = bsearchHelper (a, L, R, x); /*kickoff*/

   if (a[L] == x)       /* final */
        return L;
   else
        return -1;
}
```

Trace

-17 -5 3 6 12 21 45 142

bSearch(a, 8, 5) -1
   bsearchHelper(a, -1, 8, 5) 2
      bsearchHelper(a, -1, 3, 5) 2
         bsearchHelper(a, 1, 3, 5) 2
            bsearchHelper(a, 2, 3, 5) 2

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

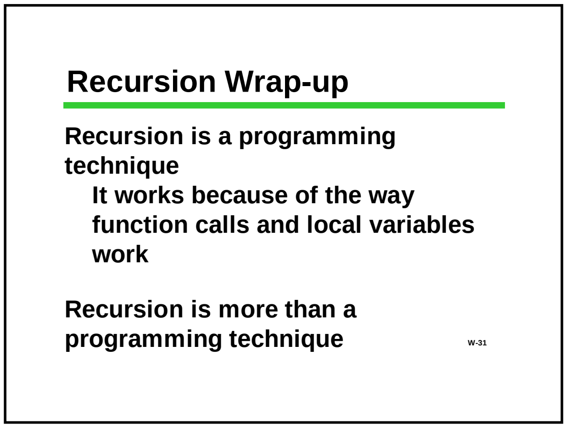
When to Use Recursion?

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



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