CMPSC 24: Lecture 13
Recursion

Divyakant Agrawal
Department of Computer Science
UC Santa Barbara

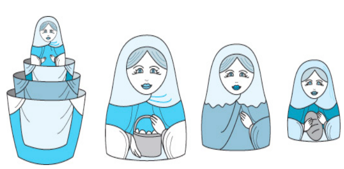
5/12/10 1

Lecture Plan

- Recursion
  - General structure of recursive solutions
  - Why do recursive solutions terminate?
  - How do recursive programs manage the stack?
  - Tail recursion
  - When to use recursion?

2

What Is Recursion?



Recursion like a set of Russian dolls.

3

What Is Recursion?

- **Recursive call** A method call in which the method being called is the same as the one making the call
- **Direct recursion** Recursion in which a method directly calls itself
  – example
- **Indirect recursion** Recursion in which a chain of two or more method calls returns to the method that originated the chain
  – example

4

Recursion

- You must be careful when using recursion.
- Recursive solutions can be less efficient than iterative solutions.
- Still, many problems lend themselves to simple, elegant, recursive solutions.

5

Some Definitions

- **Base case** The case for which the solution can be stated non-recursively
- **General (recursive) case** The case for which the solution is expressed in terms of a smaller version of itself
- **Recursive algorithm** A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case

6

Finding a Recursive Solution

- Each successive recursive call should bring you closer to a situation in which the answer is known.
- A case for which the answer is known (and can be expressed without recursion) is called a base case.
- Each recursive algorithm must have at least one base case, as well as the general (recursive) case

7

General format for many recursive functions

```
if   (some condition for which answer is known)
                         // base case
     solution statement

else                              // general case

     recursive function call
```

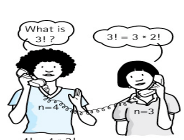
8

Computing Factorial

Recursive definition

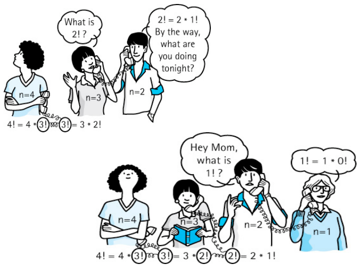
A definition in which something is defined in terms of a smaller version of itself

What is 3 factorial?



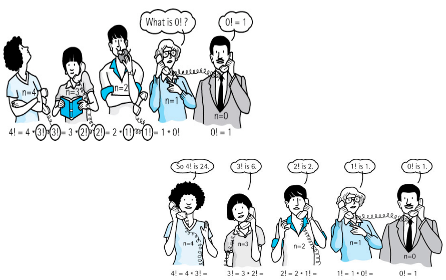
9

Computing Factorial



10

Recursive Computation



11

Factorial Program

The function call Factorial(4) should have value 24, because that is 4 * 3 * 2 * 1 .

For a situation in which the answer is known, the value of 0! is 1.

So our base case could be along the lines of

```
if  ( number == 0 )
return 1;
```

12

Factorial Program

Now for the general case . . .

The value of Factorial(n) can be written as n * the product of the numbers from (n - 1) to 1,
that is,

n * (n - 1) * . . . * 1

or, n * Factorial(n - 1)

And notice that the recursive call Factorial(n - 1) gets us "closer" to the base case of Factorial(0).

13

Recursive Factorial

```
int Factorial ( int number )
// Pre: number >= 0.
{
     if ( number == 0)          //  base case
       return  1 ;
     else                    // general case
     return  number * Factorial ( number - 1 ) ;
}
```

Why is this correct?

14

Three-Questions for Verifying Recursive Functions

- **Base-Case Question: Is there a non-recursive way out of the function?**
- **Smaller-Caller Question: Does each recursive function call involve a smaller case of the original problem leading to the base case?**
- **General-Case Question: Assuming each recursive call works correctly, does the whole function work correctly?**

15

Computing Exponentiation Recursively

- From mathematics, we know that
  $2^0 = 1$ and $2^5 = 2 * 2^4$
- In general,
  $x^0 = 1$ and $x^n = x * x^{n-1}$
  for integer x, and integer n > 0.
- Here we are defining x^n recursively, in terms of x^{n-1}

16

```
// Recursive definition of power function

int  Power (  int   x,   int   n )

{
   if ( n == 0 )
        return  1;        //  base case
   else                          // general case
        return (  x * Power ( x , n-1 ) ) ;
}
```

Can you compute multiplication recursively?
How about addition?

17

Fibonacci Sequence

Shall we try it again?

Problem: Calculate Nth item in Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

- *What is the next number?*
- *What is the size of the problem?*
- *Which case do you know the answer to?*
- *Which case can you express as a smaller version of the size?*

18

Fibonacci Program

```
int Fibonacci(int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return Fibonacci(n-2) +
Fibonacci(n-1);
)
```

That was easy, but it is not very efficient. Why?

19

Recursive Linear Search

```
struct  ListType
{
   int  length ;   // number of elements in the list

   int  info[ MAX_ITEMS ] ;

} ;

ListType   list ;
```

20

Problem Instance

PROTOTYPE

bool ValueInList(ListType list , int value , int startIndex) ;

74	36	. .	95	75	29	47	. . .

list[0]   [1]   [startIndex]   [length -1]

Already searched

index of current element to examine

Needs to be searched

21

```
bool ValueInList ( ListType  list , int  value, int startIndex )

// Searches list for value between positions startIndex
// and  list.length-1
// Pre: list.info[ startIndex ] . . list.info[ list.length - 1 ]
//       contain values to be searched
//  Post:  Function value =
//    ( value exists in list.info[ startIndex ] . .
//   list.info[ list.length - 1 ] )
{
   if  ( list.info[startIndex] == value )    // one base case   return
   true ;
   else  if  (startIndex == list.length -1 ) // another base case
     return  false ;
   else                              // general case      return
   ValueInList( list, value, startIndex + 1 ) ;
}
```

22

"Why Use Recursion?"

- Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement.
- However, for certain problems the recursive solution is the most natural solution. Build a prototype. A more efficient iterative solution can be developed later.
- Recursive solutions are easier to reason about.
- The *Functional Programming* paradigm adopts recursion.

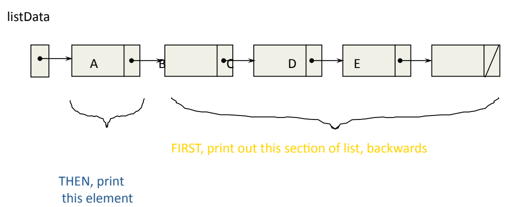
23

Printing List in Reverse

```
struct  NodeType
{
   int  info ;
   NodeType*  next ;
}
class  SortedType
{
public :

         .  .  .

private :
    NodeType*  listData  ;
} ;
```

24

RevPrint(listData)



25

Base Case and General Case

Base case: list is empty

Do nothing

General case: list is non-empty

Extract the first element;

Print rest of the list (may be empty);

Print the first element

26

Printing in Reverse

```
void   RevPrint ( NodeType*  listPtr )
{
   if  ( listPtr != NULL )        // general case
   {
     RevPrint ( listPtr-> next ) ;  //process the rest
     std::cout << listPtr->info << std::endl ;
                  // print this element
   }
  // Base case : if the list is empty, do nothing
}
```

How would this work without recursion?

27

Function `BinarySearch( )`

- `BinarySearch` takes sorted array `info`, and two subscripts, `fromLoc` and `toLoc`, and `item` as arguments. It returns false if `item` is not found in the elements `info[fromLoc…toLoc]`. Otherwise, it returns true.
- `BinarySearch` can be written using iteration, or using recursion.

28

Function `BinarySearch( )`

```
bool  BinarySearch( ItemType  info[ ],  ItemType  item,
                    int fromLoc , int  toLoc )
   // Pre: info [ fromLoc . . toLoc ] sorted in ascending order
   // Post: Function value = ( item  in info [ fromLoc .. toLoc] )

{  int  mid;
    if  ( fromLoc > toLoc )        //  base case -- not found
             return  false ;
    else {
         mid = ( fromLoc + toLoc ) / 2 ;
       switch ( item.ComparedTo( info [ mid ] ) )
          { case  EQUAL:  return true; //base case-- found at mid
           case   LESS:   return BinarySearch ( info, item, fromLoc, mid-1 );
           case  GREATER: return BinarySearch( info, item, mid + 1, toLoc );
       }
}
```

Which version is easier: compare to iterative version presented next

29

Iterative `BinarySearch( )`

```
bool  BinarySearch( ItemType info[], ItemType item,  int fromLoc, int toLoc)
{  int mid ;
   int  first  =  fromLoc;
   int   last  = toLoc;
   bool found = false ;
   while (( first  <=  last )  &&  !found )
   { mid  =  ( first + last ) / 2 ;
     switch ( item.ComparedTo( info [ mid ] ) )
        {  case   LESS:  last = mid - 1 ;
          break ;
        case  GREATER:  first = mid + 1 ;
            break ;
        case  EQUAL:  found = true  ;
          break ;
        }
   }
   return found;
}
```

30

When a function is called...

- A transfer of control occurs from the calling block to the code of the function. It is necessary that there be a return to the correct place in the calling block after the function code is executed. This correct place is called the return address.
- When any function is called, the run-time stack is used. On this stack is placed an activation record (stack frame) for the function call.
  - This stores all the variables local to the called function.

31

Stack Activation Frames

- The activation record stores the return address for this function call, and also the parameters, local variables, and the function's return value.
- The activation record for a particular function call is popped off the run-time stack when the final closing brace in the function code is reached, or when a return statement is reached in the function code.
- At this time the function's return value, if non-void, is brought back to the calling block return address for use there.

32

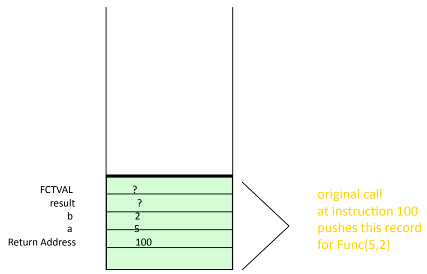
Mystery Recursive Function

```
// Another recursive function
int  Func   (  int  a, int b )

{
   int  result;
   if  ( b == 0 )                        //  base case
     result = 0;
   else if  ( b > 0 )                  //  first  general  case

     result = a +  Func ( a , b - 1 ) ) ;   // instruction  50

   else                     //  second general case
     result = Func ( - a , - b ) ;   // instruction 70

   return  result;
}
```

33

Run-Time Stack Activation Records

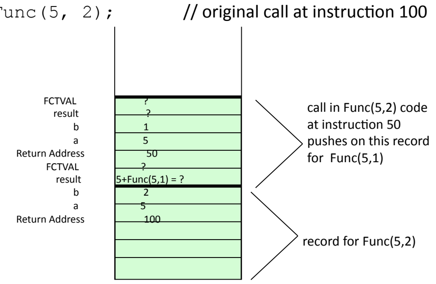
`x = Func(5, 2);` // original call is instruction 100



34

Second Call

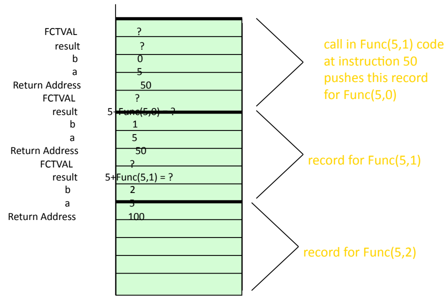
`x = Func(5, 2);` // original call at instruction 100



35

Third Call

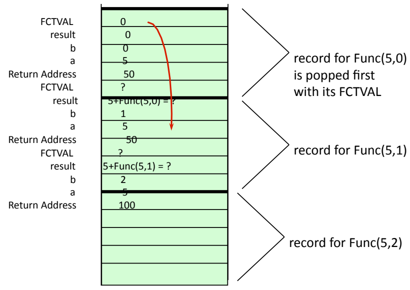
`x = Func(5, 2);` // original call at instruction 100



36

Third Call Completes

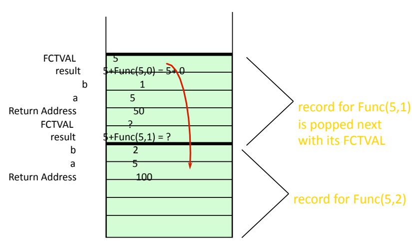
`x = Func(5, 2);` // original call at instruction 100



37

Second Call Completes

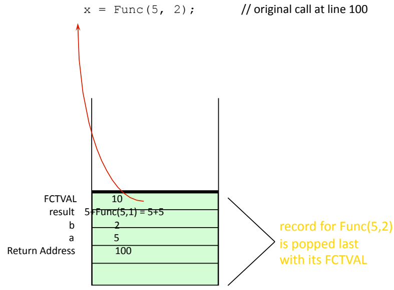
`x = Func(5, 2);` // original call at instruction 100



38

First Call Completes

`x = Func(5, 2);` // original call at line 100



39

Practice: Show Activation Records For These Calls

x = Func(- 5, - 3);

x = Func(5, - 3);

What operation does Func(a, b) simulate?

40

Tail Recursion

- **The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.**
- **Tail recursion can be replaced by iteration to remove recursion from the solution as in the next example.**

41

Tail Recursion Example

```
bool ValueInList ( ListType  list , int  value , int  startIndex  )
{
   if  ( list.info[startIndex] == value )        // one base case
    return  true ;
   else  if  (startIndex == list.length -1 )    // another base case
      return  false ;
   else                        // general case
      return ValueInList( list, value, startIndex + 1 ) ;
}
```

42

Equivalent Iterative Version

```
bool ValueInList (  ListType  list , int value , int startIndex  )
{
   while  (list.info[startIndex] != value &&  startIndex !=
   list.length-1 )
     startIndex++ ;
   if ( value == list.info[ startIndex ] )
            return true ;
     else
          return  false;
}
```

So, what is the general logic?

43

Convert into Iterative Solution

```
int Power(int number, int exponent)
{
  if (exponent == 0)
    return 1
  else
    return number * Power(number,
exponent - 1)
}
```

44

Iterative Equivalent

```
int Power (int number, int exponent)
{
   int val = 1;
   while (exponent != 0)
   {
     val = number*val;
     exponent--;
   }
   return val;
}
```

What is the logic?

45

Tower of Hanoi

Worked out on the blackboard

46