Programming with Recursion

Recursive Function Call

- **A recursive call is a function call in which the called function is the same as the one making the call.**

- **In other words, *recursion occurs when a function calls itself!***

- **We must avoid making an infinite sequence of function calls (infinite recursion).**

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.

General format for many recursive functions

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

else                                    // general case

     recursive function call
```

SOME EXAMPLES . . .

Writing a recursive function to find n factorial

DISCUSSION

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;
```

Writing a recursive function to find Factorial(n)

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) * (n-2) * (n-3) * . . . *3 * 2 * 1

or, n! = n * (n - 1)! = n * (n-1) * (n-2)! = ….. = ……1! = …….1*0!

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

Recursive Solution

```
public static int Factorial (  int   n )
//   Pre:   number is assigned and  number >= 0.
{
    if  ( n == 0)                                    //  base case, for termination/stoping
            return  1 ;
    else                                             // general case, for recursion
            return  n * Factorial ( n - 1 )  ;
}
```

Three-Question Method of verifying recursive functions

- **Base-Case Question:** Is there a nonrecursive way out of the function?

  The answer should be "yes"
- **Smaller-Caller Question:** Does each recursive function call involve a smaller case of the original problem leading to the base case?

  The answer should be "yes"
- **General-Case Question:** Assuming each recursive call works correctly, does the whole function work correctly?

  The answer should be "yes"

Another example where recursion comes naturally

- From mathematics, we know that

  $2^0 = 1$ and $2^5 = 2 * 2^4$

- In general,

  $x^0 = 1$ and $x^n = x * x^{n-1} = x*x*x^{n-2}$

  for integer x, and integer $n > 0$.

- Here we are defining x^n recursively, in terms of x^{n-1}
- $x^n = x * x * x * x* \ldots\ldots$ n times

```
// Recursive definition of power function
 public static int Power (  int   x,   int   n )

    //  Pre:     n >= 0.   x, n are not both zero
    //  Post:   Function value = x raised to the power n.

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

Of course, an alternative would have been to use looping instead of a recursive call in the function body.

How recursion works?

Power: x^n

$$x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0)))$$
$$= x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x))$$
$$= x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$$

call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x \cdot x^1$	$= x \cdot x$
call 4	$x \cdot x^0$	$= x \cdot 1 = x$
call 5	1	

or alternatively, as

call 1	`power(x,4)`
call 2	`power(x,3)`
call 3	`power(x,2)`
call 4	`power(x,1)`
call 5	`power(x,0)`
call 5	1
call 4	x
call 3	$x \cdot x$
call 2	$x \cdot x \cdot x$
call 1	$x \cdot x \cdot x \cdot x$

How recursion works?

Power: x^n

```
            static public void main(String args[]) {
            { ...
/* 136 */    y = power(5.6,2);
              ...
            }
```

A trace of the recursive calls is relatively simple, as indicated by this diagram

call 1	power(5.6,2)		
call 2		power(5.6,1)	
call 3			power(5.6,0)
call 3			1
call 2		5.6	
call 1	31.36		

Non-recursive solution

Power: x^n

// Pseudo code. Remember: x^n = x * x * x * x* …… n times = 1*x*x*….x n times

```
NonRecursivePower(x,n)
{
        Result =1;
        For i = 1 to n
                result = result * x
}
```

Jave code

```
double nonRecPower(double x, int n) {
  double result = 1;
  if (n > 0)
  for (result = x; n > 1; --n)
        result *= x;
  return result;
}
```

Recursive function to determine if value is in list

PROTOTYPE

public boolean ValueInList (int 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

```
public boolean ValueInList (int list[ ] , int  value ,  int  startIndex  )
//    Searches list for value between positions startIndex
//    and  list.length-1
//    Pre:   list [ startIndex ] . . list [ list.length - 1 ]
//             contain values to be searched
//   Post:  Function value =
//        ( value exists in list [ startIndex ] . . list [ list.length - 1 ] )
{
     if  ( list[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 ) ;
}

main()
{
ValueInList(list, value, 0);
}
```

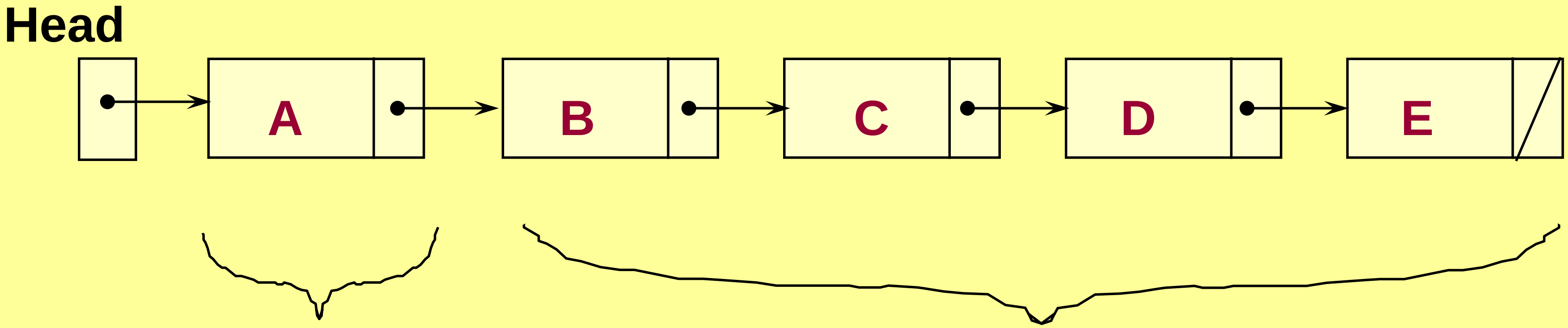
"Why use recursion?"

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. This often occurs when pointer variables are used.

• Recursion is easy to code.

Example: Print link list data in reverse order



THEN, print
this element

FIRST, print out this section of list, backwards

So, the output is: E D C B A

Example Continued
Base Case and General Case

A base case may be a solution in terms of a "smaller" list. Certainly for a list with 0 elements, there is no more processing to do.

Our general case needs to bring us closer to the base case situation. That is, the number of list elements to be processed decreases by 1 with each recursive call. By printing one element in the general case, and also processing the smaller remaining list, we will eventually reach the situation where 0 list elements are left to be processed.

In the general case, we will print the elements of the smaller remaining list in reverse order, and then print the current pointed to element.

Example Continued: Solution using recursion

```
public static void reversePrint (LN   pointer)
 {
            if (pointer == null)
                        return;
            else {
                        reversePrint(pointer.next);
                        System.out.println(pointer.value);
             };
}

main()
{
reversePrint(Head);
}
```

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.**

```
found = BinarySearch(info, 25, 0, 14 );
```

item fromLoc toLoc

indexes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
info	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28
									16	18	20	22	24	26	28
													24	26	28
													24		

NOTE: denotes element examined

```
// Recursive definition

public boolean BinarySearch (int info[ ] , int 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 ;

                  if ( info [ mid ] == item )      // base case-- found at mid

                       return  true  ;

                 else  if ( item < info [ mid ] )         //  search lower half
                       return BinarySearch ( info, item, fromLoc, mid-1 ) ;
                 else                                        // search upper half
                       return  BinarySearch( info, item, mid + 1, toLoc ) ;
                  }

}
```

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.

Some Examples

Tail Recursion

```
void tail (int i) {
  if (i > 0) {
    System.out.print (i + "");
    tail(i-1);
  }
}
```

Non-Tail Recursion

```
void nonTail (int i) {
  if (i > 0) {
    nonTail(i-1);
    System.out.print (i + "");
    nonTail(i-1);
  }
}
```

Iteration/Loop

```
void iterativeEquivalentOfTail (int i) {
  for ( ; i > 0; i--)
  System.out.print(i+ "");

}
```

Another Example: press keyboard characters, finish by pressing "Enter", print the characters in reverse order

By Non-Tail Recursion

```
void reverse() {
    char ch = getChar();
    if (ch != '\n') {
        reverse();
        System.out.print(ch);
    }
}
```

Another Example: Search Value

By Tail Recursion

```
public boolean ValueInList (int list[ ] , int  value ,  int  startIndex  )

//    Searches list for value between positions startIndex
//    and  list.length-1
//    Pre:   list [ startIndex ] . . list [ list.length - 1 ]
//               contain values to be searched
//   Post:  Function value =
//         ( value exists in list [ startIndex ] . . list [ list.length - 1 ] )
{
    if  ( list[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 ) ;
}
```

By Loop/Iteration

```
// ITERATIVE SOLUTION
public boolean ValueInList (int 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 ] )
{ boolean   found;
  found =  false ;
     while  ( !found  &&  startIndex < list.length )
       {         if ( value == list [ startIndex ] )
                                found = true ;
                 else           startIndex++ ;
       }
       return  found ;
}
```

Recursion is not always good

- It may be very slow when excessive recursion calls
- Example: Compute Fibonacci Number

$$\mathrm{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1) & \text{otherwise} \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Compute Fibonacci Number: Continue…

Recursive solution:

```
int Fib (int n) {
  if (n < 2)
    return n;
  else return Fib(n-2) + Fib(n-1);
}
```

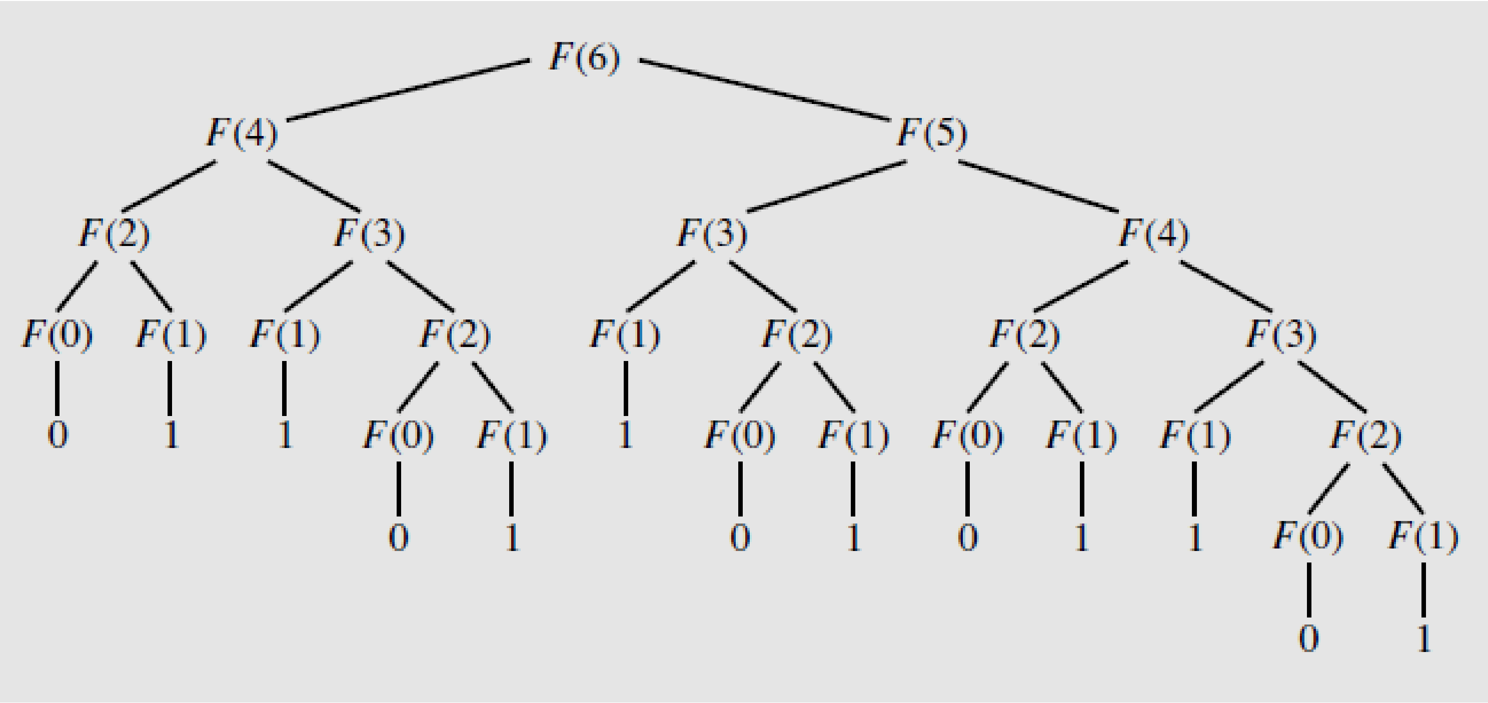
```
Fib(6) =              Fib(4)                      + Fib(5)
       =    Fib(2)        +        Fib(3)         + Fib(5)
       = Fib(0)+Fib(1)    +        Fib(3)         + Fib(5)
       =   0    +   1     +        Fib(3)         + Fib(5)
       =        1         + Fib(1)+   Fib(2)      + Fib(5)
       =        1         + Fib(1)+Fib(0)+Fib(1) + Fib(5)
```

etc.

Compute Fibonacci Number: Continue…

Recursive solution:

The tree of calls for `Fib(6)`.



- Do you see what is the problem here?
- 25 calls **!!!** to compute F(6)
- The problem is: Same function is repeated again and again. For example, F(1) has been computed 8 times!

Compute Fibonacci Number: Continue…

Recursive solution:

Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Compute Fibonacci Number: Continue…

Iterative solution:

An iterative algorithm may be produced rather easily as follows:

```
int iterativeFib (int n) {
    if (n < 2)
        return n;
    else {
        int i = 2, tmp, current = 1, last = 0;
        for ( ; i <= n; ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

Compute Fibonacci Number: Continue…

Iterative Solution Vs Recursion

Comparison of iterative and recursive algorithms for calculating Fibonacci numbers.

		Assignments	
n	Number of Additions	Iterative Algorithm	Recursive Algorithm
6	5	15	25
10	9	27	177
15	14	42	1,973
20	19	57	21,891
25	24	72	242,785
30	29	87	2,692,537

Non-recursive solution

Use a recursive solution when:

- The depth of recursive calls is relatively "shallow" compared to the size of the problem.
- The recursive version does about the same amount of work as the nonrecursive version.
- The recursive version is shorter and simpler than the nonrecursive solution.

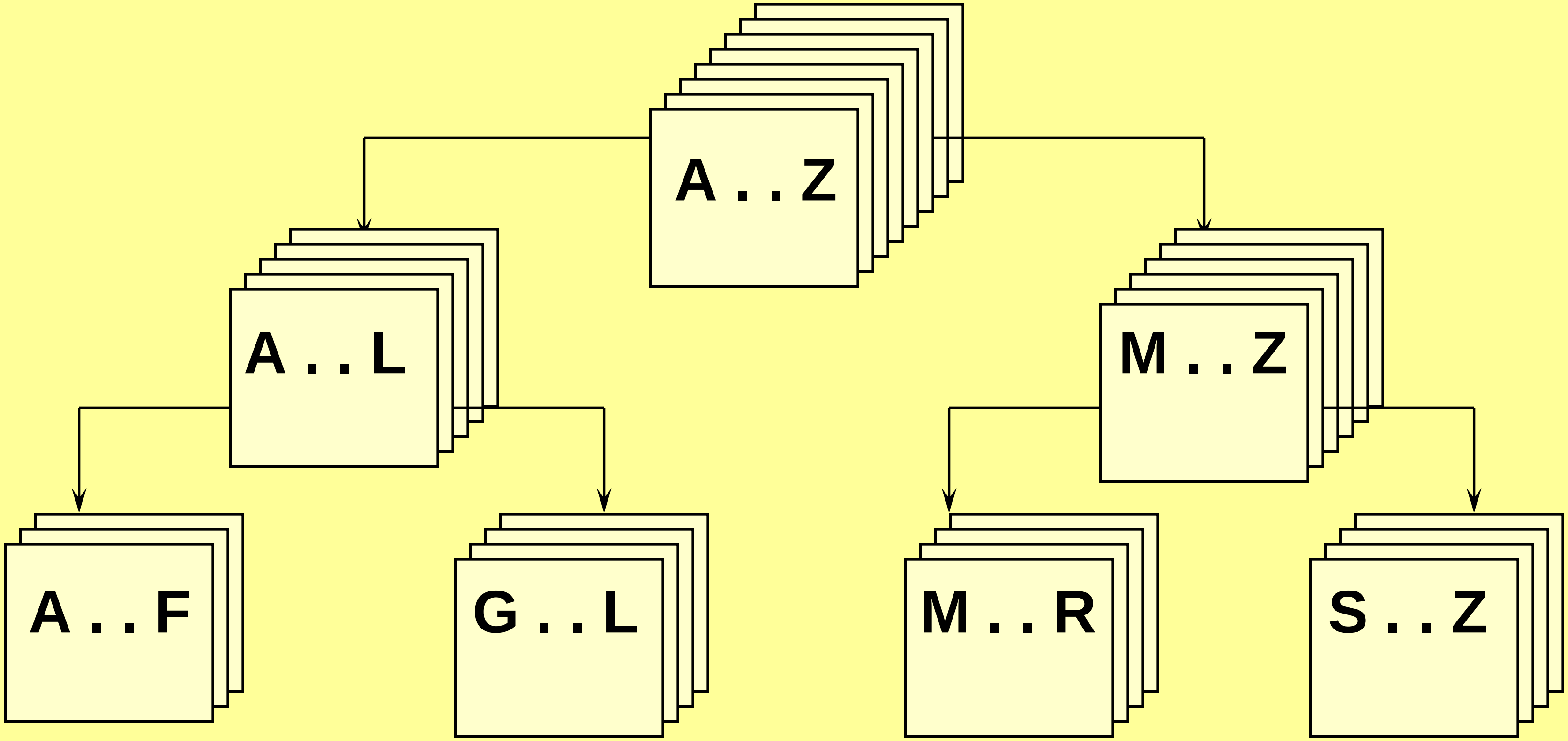
SHALLOW DEPTH

EFFICIENCY

CLARITY

Example: Quick sort

- Simple and recursive, but fastest sorting algorithm



Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

9	*20*	*6*	*10*	*14*	*3*	*60*	*11*

values[first] **[last]**

After call to function Split

splitVal = 9

smaller values **larger values**

6	***3***	***9***	***10***	***14***	***20***	***60***	***11***

values[first] **[splitPoint]** **[last]**

```
// Recursive quick sort algorithm
public void QuickSort (int values[ ], int first , int last )

//  Pre:   first <= last
//  Post: Sorts array values[ first .  . last ] into ascending order
{
    if  ( first < last )                               //  general case

    {       int  splitPoint ;

            Split ( values, first, last, splitPoint ) ;

            // values [ first ] . . values[splitPoint - 1 ] <= splitVal
            // values  [ splitPoint ] = splitVal
            // values [ splitPoint + 1 ] . . values[ last ] > splitVal

            QuickSort( values,  first,  splitPoint - 1 ) ;
            QuickSort( values,  splitPoint + 1,  last );
    }

} ;
```

Remember: Quick Sort is the best sorting algorithm