Chapter 13

RECURSION

Learning Objectives

- Recursive void Functions
  - Tracing recursive calls
  - Infinite recursion, overflows
- Recursive Functions that Return a Value
  - Powers function
- Thinking Recursively
  - Recursive design techniques
  - Binary search

Introduction to Recursion

- A function that "calls itself"
  - Said to be ***recursive***
  - In function definition, call to same function
- C++ allows recursion
  - As do most high-level languages
  - Can be useful programming technique
  - Has limitations

Recursive void Functions

- Divide and Conquer
  - Basic design technique
  - Break large task into subtasks
- Subtasks could be smaller versions of the original task!
  - When they are → recursion

Recursive void Function Example

- Consider task:
- Search list for a value
  - Subtask 1: search 1st half of list
  - Subtask 2: search 2nd half of list
- Subtasks are smaller versions of original task!
- When this occurs, recursive function canbe used.
  - Usually results in "elegant" solution

Recursive void Function: Vertical Numbers

- Task: display digits of number vertically, one per line
- Example call:
  writeVertical(1234);
  Produces output:
  1
  2
  3
  4

Vertical Numbers: Recursive Definition

- Break problem into two cases
- Simple/base case: if n<10
  - Simply write number n to screen
- Recursive case: if n>=10, two subtasks:
  1- Output all digits except last digit
  2- Output last digit
- Example: argument 1234:
  - 1^{st} subtask displays 1, 2, 3 vertically
  - 2^{nd} subtask displays 4

writeVertical Function Definition

- Given previous cases:

```
void writeVertical(int n)
{
    if (n < 10)                    //Base case
        cout << n << endl;
    else
    {                              //Recursive step
        writeVertical(n/10);
        cout << (n%10) << endl;
    }
}
```

```
Example call:
writeVertical(123);
    writeVertical(12);  (123/10)
            writeVertical(1);  (12/10)
                cout << 1 << endl;
        cout << 2 << endl;
cout << 3 << endl;
```

writeVertical Trace

- Example call:

```
writeVertical(123);
→ writeVertical(12);   (123/10)
        → writeVertical(1);  (12/10)
            → cout < < 1 < < endl;
        cout < < 2 < < endl;
cout < < 3 < < endl;
```

- Arrows indicate task function performs
- Notice 1st two calls call again (recursive)
- Last call (1) displays and "ends"

Recursion—A Closer Look

- Computer tracks recursive calls
  - Stops current function
  - Must know results of new recursive call before proceeding
  - Saves all information needed for current call
    - To be used later
  - Proceeds with evaluation of new recursive call
  - When THAT call is complete, returns to "outer" computation

Recursion Big Picture

- Outline of successful recursive function:
  - One or more cases where function accomplishes it’s task by:
    - Making one or more recursive calls to solve smaller versions of original task
    - Called "recursive case(s)"
  - One or more cases where function accomplishes it’s task without recursive calls
    - Called "base case(s)" or stopping case(s)

Infinite Recursion

- Base case MUST eventually be entered
- If it doesn't → infinite recursion
  - Recursive calls never end!
- Recall writeVertical example:
  - Base case happened when down to 1-digit number
  - That's when recursion stopped

Infinite Recursion Example

- Consider alternate function definition:

```
void newWriteVertical(int n)
{
	newWriteVertical(n/10);
	cout << (n%10) << endl;
}
```

- Seems "reasonable" enough
- Missing "base case"!
- Recursion never stops

Stacks for Recursion

- A stack
  - Specialized memory structure
  - Like stack of paper
    - Place new on top
    - Remove when needed from top
  - Called "last-in/first-out" memory structure
- Recursion uses stacks
  - Each recursive call placed on stack
  - When one completes, last call is removed from stack

Stack Overflow

- Size of stack limited
  - Memory is finite
- Long chain of recursive calls continually adds to stack
  - All are added before base case causes removals
- If stack attempts to grow beyond limit:
  - Stack overflow error
- Infinite recursion always causes this

Recursion Versus Iteration

- Recursion not always "necessary"
- Not even allowed in some languages
- Any task accomplished with recursion can also be done without it
  - Nonrecursive: called iterative, using loops
- Recursive:
  - Runs slower, uses more storage
  - Elegant solution; less coding

Recursive Functions that Return a Value

- Recursion not limited to void functions
- Can return value of any type
- Same technique, outline:
  1. One+ cases where value returned is computed by recursive calls
     - Should be "smaller" sub-problems
  2. One+ cases where value returned computed without recursive calls
     - Base case

Return a Value Recursion Example: Powers

- Recall predefined function pow():
  result = pow(2.0,3.0);
  - Returns 2 raised to power 3 (8.0)
  - Takes two double arguments
  - Returns double value
- Let’s write recursively
  - For simple example

Function Definition for power()

```
int power(int x, int n)
{
        if (n<0)
        {
                cout << "Illegal argument";
                exit(1);
        }
        if (n>0)
                return (power(x, n-1)*x);
        else
                return (1);
}
```

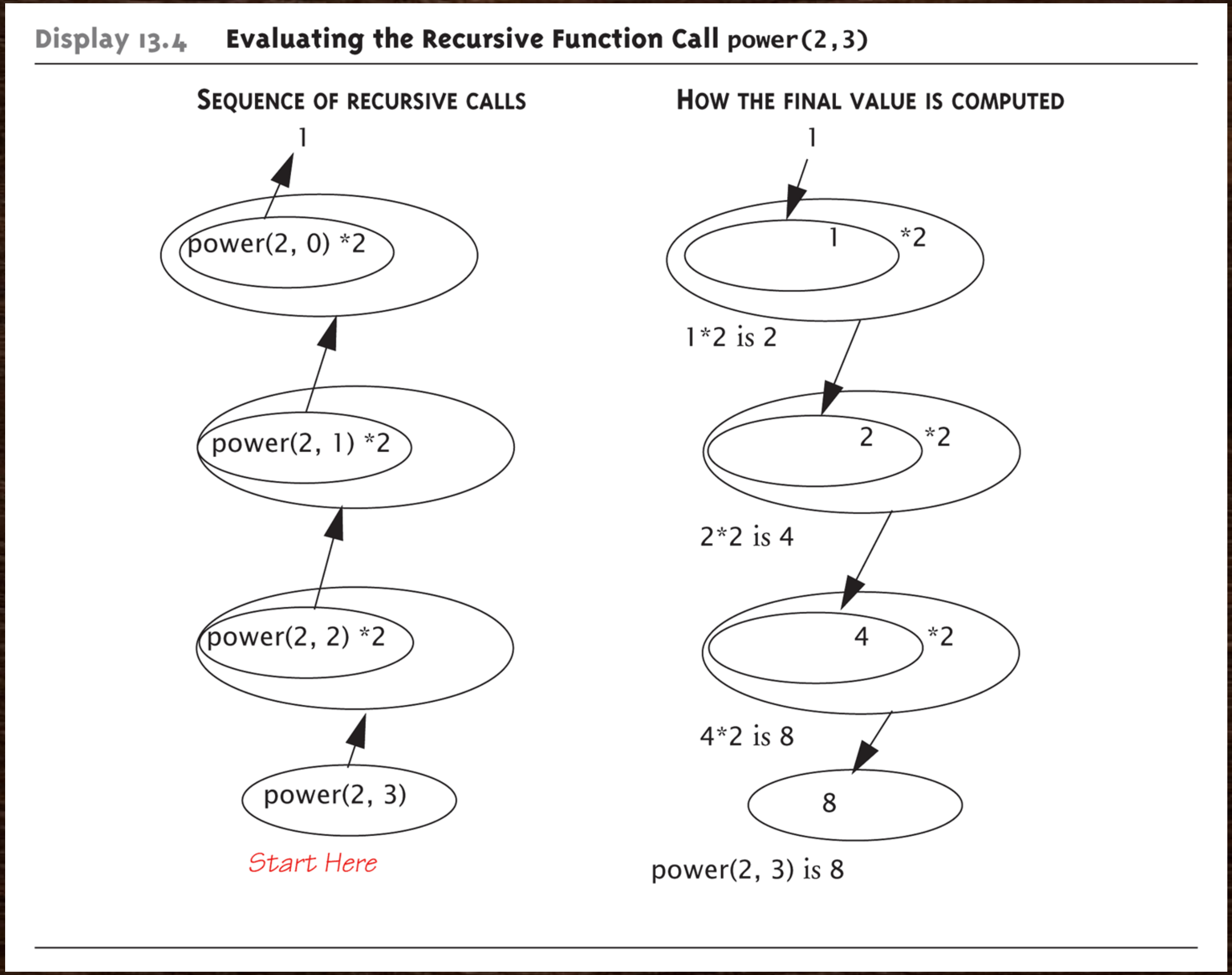
Calling Function power()

- Example calls:
- power(2, 0);
  → returns 1
- power(2, 1);
  → returns (power(2, 0) * 2);
  → returns 1
  - Value 1 multiplied by 2 & returned to original call

Calling Function power()

- Larger example:
  power(2,3);
  → power(2,2)*2
  → power(2,1)*2
  →power(2,0)*2
  →1
  - Reaches base case
  - Recursion stops
  - Values "returned back" up stack

Tracing Function power():
Display 13.4 Evaluating the Recursive Function Call power(2,3)



Thinking Recursively

- Ignore details
  - Forget how stack works
  - Forget the suspended computations
  - Yes, this is an "abstraction" principle!
  - And encapsulation principle!
- Let computer do "bookkeeping"
  - Programmer just think "big picture"

Thinking Recursively: power

- Consider power() again
- Recursive definition of power:
  power(x, n)

  returns:

  power(x, n – 1) * x
  - Just ensure "formula" correct
  - And ensure base case will be met

Recursive Design Techniques

- Don't trace entire recursive sequence!
- Just check 3 properties:

1. No infinite recursion
2. Stopping cases return correct values
3. Recursive cases return correct values

Recursive Design Check: power()

- Check power() against 3 properties:
  1. No infinite recursion:
     - 2nd argument decreases by 1 each call
     - Eventually must get to base case of 1
  2. Stopping case returns correct value:
     - power(x,0) is base case
     - Returns 1, which is correct for x^0
  3. Recursive calls correct:
     - For n>1, power(x,n) returns power(x,n-1)*x
     - Plug in values → correct

```
int power(int x, int n)
{
        if (n<0)
        {
          cout << "Illegal argument";
                exit(1);
        }
        if (n>0)
                return (power(x, n-1)*x);
        else
                return (1);
}
```

determine if an input is prime

```
bool isPrime(int p, int i=2)
{
    if (i==p) return 1;        // i*i > p for faster
    if (p%i == 0) return 0;
    return isPrime (p, i+1);
}
```

adding up numbers from 1 to any given number

```
1) int sum (int num)
2) {
3)     if (num==0)      return 0;
4)     return (sum(num-1)+(num));
5) }
```

Design a faster version for power()

- Analysis

```
int power(int x, int n)
{
    if (n>0)
        return (power(x, n-1)*x);

    else   return (1);
}
```

```
int power(int x, int n)
{
        if (n<0)
        {
                cout << "Illegal argument";
                exit(1);
        }
        if (n>0)
                return (power(x, n-1)*x);
        else
                return (1);
}
```

Think about more efficient version!

Do it, thx!

Algorithm Fast-Exponentiate(x, n)

```
if n = 0 then  return 1
else if n is even then
      return Fast-Exponentiate(x^2, n / 2)
else
      return x * Fast-Exponentiate(x^2, (n - 1) / 2)
```

Tail recursion

- A function that is tail recursive
  - if it has the property that no further computation occurs after the recursive call
  - I.e. the function immediately returns.
- Tail recursive functions can easily be converted to a more efficient iterative solution
  - May be done automatically by your compiler

```
function bar(data) {
    if ( a(data) ) {
        return b(data);
    }
    return c(data);
}
```

Mutual Recursion

- When two or more functions call each other it is called mutual recursion
- Example
  - Determine if a string has an even or odd number of 1's by invoking a function that keeps track if the number of 1's seen so far is even or odd
  - Would result in stack overflow for long strings

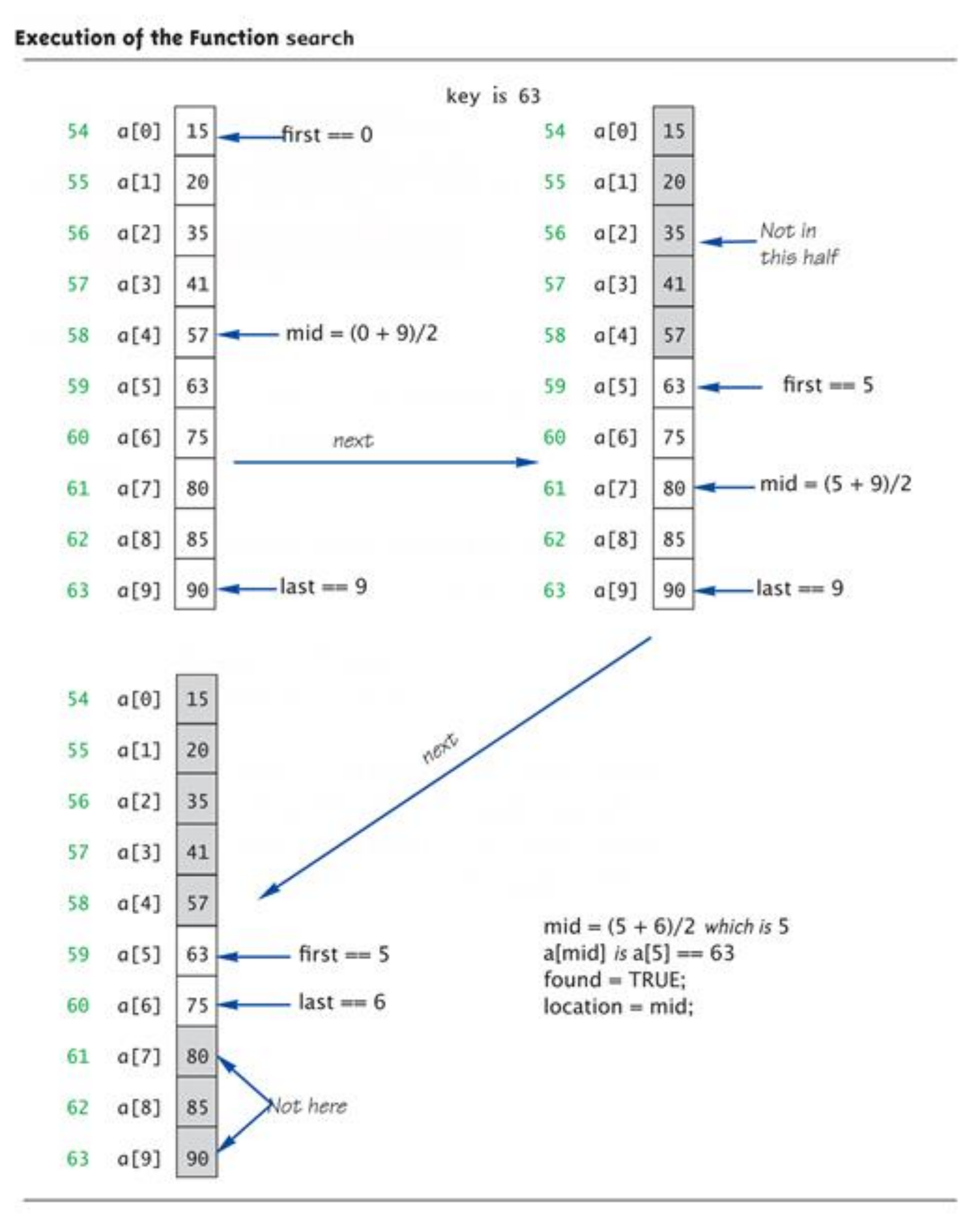
Mutual Recursion Example (1 of 2)

```
// Function prototypes
bool evenNumberOfOnes(string s);
bool oddNumberOfOnes(string s);

// If the recursive calls end here with an empty string
// then we had an even number of 1's.
bool evenNumberOfOnes(string s)
{
        if (s.length() == 0)
                return true; // Is even
        else if (s[0]=='1')
                return oddNumberOfOnes(s.substr(1));
        else
                return evenNumberOfOnes(s.substr(1));
}
```

```
// if the recursive calls end up here with an empty string
// then we had an odd number of 1's.
bool oddNumberOfOnes(string s)
{
	if (s.length() == 0)	return false; // Not even
	else if (s[0]=='1')	return evenNumberOfOnes(s.substr(1));
	else	return oddNumberOfOnes(s.substr(1));
}
int main()
{
	string s = "10011";
	if (evenNumberOfOnes(s))
		cout << "Even number of ones." << endl;
	else
		cout << "Odd number of ones." << endl;
	return 0;
}
```

Binary Search

Execution of Binary Search: **Display 13.8** Execution of the Function search



```
void search(const int a[], int lowEnd, int highEnd, int key, bool& found, int& location)
{
    int first = lowEnd;
    int last = highEnd;
    int mid;

    found = false;//so far
    while ( (first <= last) && !(found) )
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])  last = mid - 1;
        else if (key > a[mid])  first = mid + 1;
    }
}
```

Binary Search

- Recursive function to search array
  - Determines IF item is in list, and if so: Where in list it is
- Assumes array is sorted
- Breaks list in half
  - Determines if item in 1^{st} or 2^{nd} half
  - Then searches again just that half
    - Recursively (of course)!

Display 13.6 Pseudocode for Binary Search

Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
//Precondition:
//a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
```

TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
 }
```

Checking the Recursion

- Check binary search against criteria:
  1. No infinite recursion:
     - Each call increases first or decreases last
     - Eventually first will be greater than last
  2. Stopping cases perform correct action:
     - If first > last → no elements between them, so key can't be there!
     - IF key == a[mid] → correctly found!
  3. Recursive calls perform correct action
     - If key < a[mid] → key in 1st half – correct call
     - If key > a[mid] → key in 2nd half – correct call

Efficiency of Binary Search

- Extremely fast
  - Compared with sequential search
- Half of array eliminated at start!
  - Then a quarter, then 1/8, etc.
  - Essentially eliminate half with each call
- Example:
  Array of 100 elements:
  - Binary search never needs more than 7 compares!
    - Logarithmic efficiency (log n)

Recursive Solutions

- Notice binary search algorithm actually solves "more general" problem
  - Original goal: design function to search an entire array
  - Our function: allows search of any interval of array
    - By specifying bounds *first* and *last*
- Very common when designing recursive functions

Summary 1

- Reduce problem into smaller instances of same problem -> recursive solution
- Recursive algorithm has two cases:
  - Base/stopping case
  - Recursive case
- Ensure no infinite recursion
- Use criteria to determine recursion correct
  - Three essential properties
- Typically solves "more general" problem

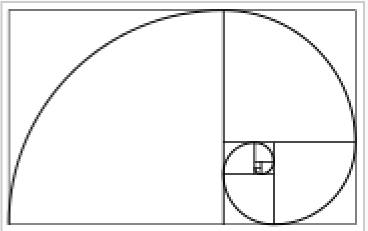
Fibonacci sequence

- Fibonacci sequence: 0,1,1,2,3,5,8,13,21,34,55,89,144,…

$$F_n = F_{n-1} + F_{n-2}, \text{ where } F_0 = 0,\ F_1 = 1$$

- Implementation:

```
1) function fib(n)
2)     if n <= 1 return n
3)     return fib(n – 1) + fib(n – 2)
```



The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling;[4] this one uses squares of sizes 1, 1, 2, 3, 5, 8, 13, 21, and 34.

Trace it and find some terms recalculated again and again!

Problem with fib()

- fib(5), produce a call tree that calls the function on the same value many different times:

- fib(5)
- fib(4) + fib(3)
- (fib(3) + fib(2)) + (fib(2) + fib(1))
- ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
- (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

fib(n) only O(n) time but O(n) space

```
var m := map(0 → 0, 1 → 1)
  function fib(n)
      if key n is not in map m
          m[n] := fib(n − 1) + fib(n − 2)
      return m[n]
```

bottom-up approach: O(n) time and O(1) space

```
function fib(n)
    if (n = 0)          return 0
    else
        var previousFib := 0, currentFib := 1
        repeat n – 1 times // loop is skipped if n = 1
            var newFib := previousFib + currentFib
            previousFib := currentFib
            currentFib  := newFib
    return currentFib
```

Example of Recursive Function: GCD

- GCD
- Using Euclid's method ($m \geq n > 0$):
- GCD(m, n) = n, if m%n = 0,
- = GCD(n, m%n), otherwise

- Dijkstra's method (assuming $m > n > 0$) G
  – CD(m, n) is same as GCD(m – n, n):
- GCD(m, n) = n, if m = n
- = GCD(m – n, n), if m > n
- = GCD(m, n – m), if n > m

GCD: Euclid’s Method

```
#include <stdio.h>
int GCD(int m, int n)
{
    if ((m % n) == 0)  return ;
    return GCD(n, m % n);
}
int main()
{
    int m, n;
    printf(” Enter m,n ” );
    scanf(” %d %d” , &m, &n);
    if (m < n)          printf(” GCD(%d , %d) = %d\n” , m, n , GCD(n , m));
    else     printf(” GCD(%d , %d) = %d\n” , m, n , GCD(m, n));
}
```

GCD: Dijkstra’ Method

```
#include <stdio.h>
int GCD(int m, int n)
 {
       if (m == n) return m;
       if (m > n) return GCD(m–n, n);
       return GCD(m, n–m);
}
int main()
{
       int m, n;
       printf(” Enter m and n:” );
       scanf(” %d %d” , &m, &n);
       printf(” GCD(%d, %d) = %d\n” , m, n, GCD(m, n));
}
```

Example of Recursive Function: Binomial Coefficient

$$\binom{n}{r} = \begin{cases} 1, \text{ if } r = 0 \\ 1, \text{ if } n = r \\ \binom{n-1}{r} + \binom{n-1}{r-1} \text{ otherwise} \end{cases}$$

Binomial Coefficient

```c
#include <stdio.h>
int binom ( int n , int r )
{
    if ( r == 0 || n == r )        return 1 ;
    return    binom ( n–1, r ) + binom ( n–1, r –1);
}
int main ( )
{
    int n , r ;
    printf ( "   Entern , r : "    ) ;
    scanf ( "  %d %d"   , &n , &r ) ;
    printf ( "   binom(%d , %d ) = %d\n"   , n , r , binom ( n , r ) ) ;
}
```

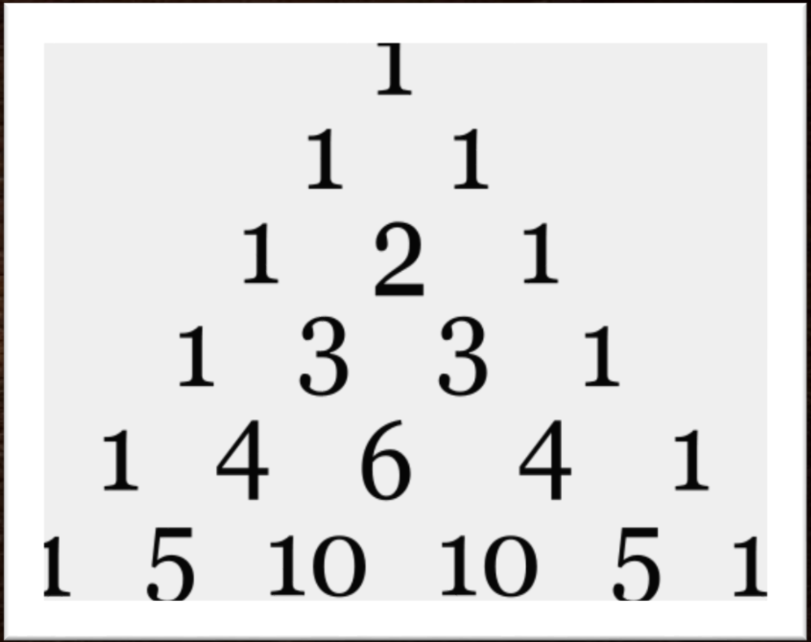
```c
int binomial(int n, int k)
{
    int num, den ;
    if ( n < k )
    {
        return(0) ;
    }
    else
    {
        den = 1;
        num = 1 ;
        for (int i =  1  ; i <= k   ; i = i+1)
            den =    den * i;
        for (int j = n-k+1; j<=n; j=j+1)
            num = num * j;
        return(num/den);
    }
}
```

Can you
List all combinations of C(n,r)

Pascal’ s Triangle

- Construction:
  - In row 0, the entry is C(0,0) = 1 (the entry is in the zeroth row and zeroth column)
  - Then, to construct the elements of the following rows, add the number above and to the left with the number above and to the right of a given position
    - If either the number to the right or left is not present, substitute a zero in its place
- Can state that the binomial coefficient C(n,k) appears in the *n*th row and *k*th column of Pascal's triangle.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



Pascal’s Triangle

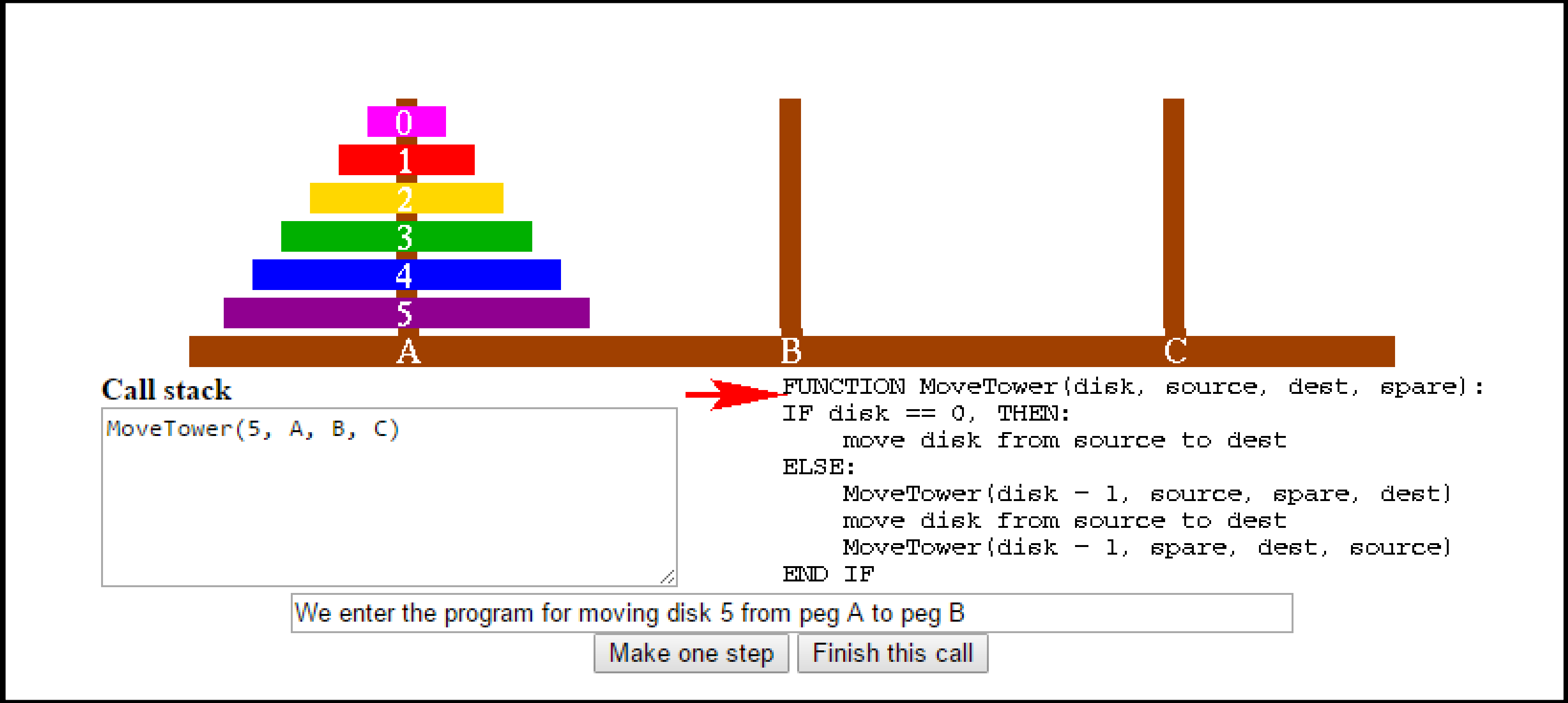
```
int main()
{
   int i, n, c;
    printf("Enter the number of rows you wish to see in pascal triangle\n");
   scanf("%d",&n);
    for (i = 0; i < n; i++)
   {
      for (c = 0; c <= (n - i - 2); c++)         printf(" ");
      for (c = 0 ; c <= i; c++)          printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));   // c(i, c)
      printf("\n");
   }
    return 0;
}

 long factorial(int n)
{
   int c;
   long result = 1;
   for (c = 1; c <= n; c++)         result = result*c;
   return result;
}
```

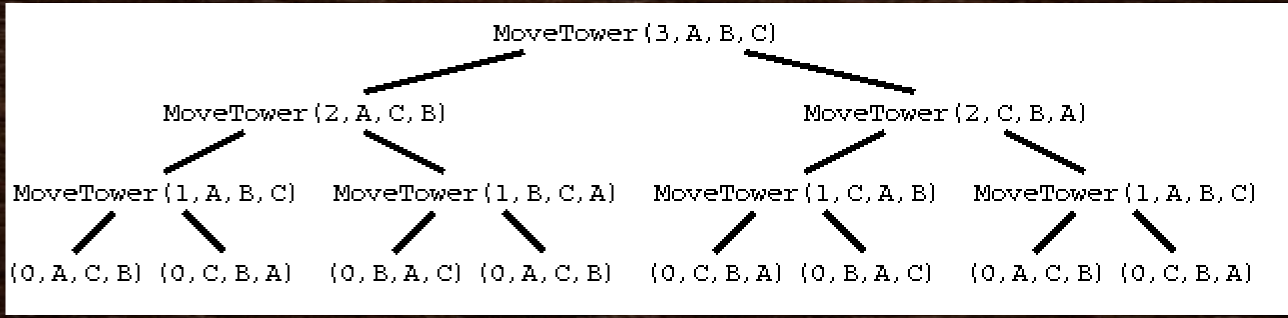
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Tower of Hanoi

- A game consists of three rods, and a number of disks of different sizes which can slide onto any rod.
- Game goal: move the entire stack to another rod
- Mechanics:
  - Starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
  - Move disk to another rod, obeying the following simple rules:
    - Only one disk can be moved at a time.
    - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack
      - i.e. a disk can only be moved if it is the uppermost disk on a stack.
      - No disk may be placed on top of a smaller disk.
- Min(moves) to solve a Tower of Hanoi with n disks is $2^{n} - 1$

Program Trace: https://www.cs.cmu.edu/~cburch/survey/recurse/hanoiex.html



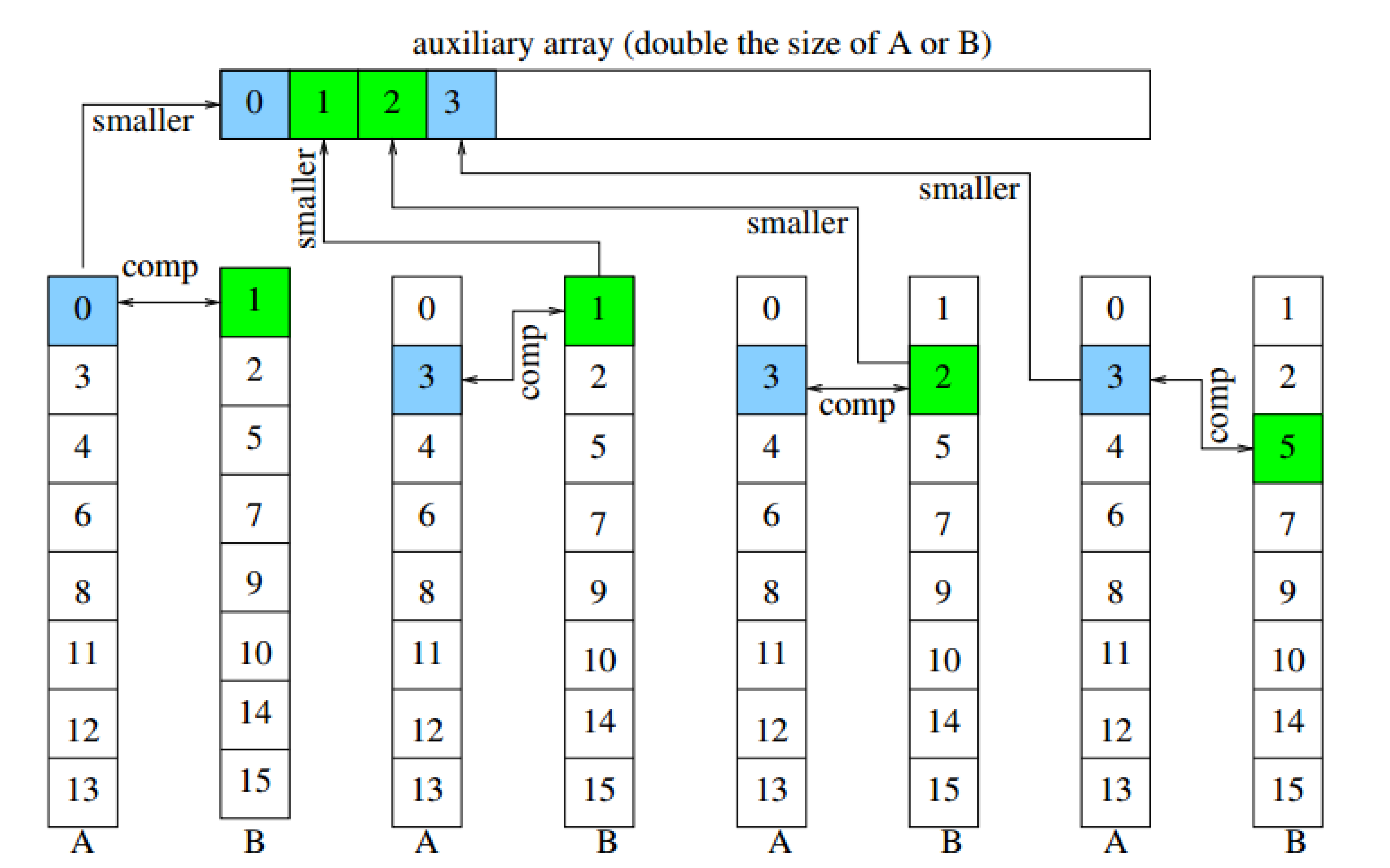


```c
#include<ctype.h>     /*  Character Class Tests  */
#include<stdio.h>     /*  Standard I/O.          */
#include<stdlib.h>    /*  Utility Functions.     */
#define EMPTY 0  /*  Empty disk position.  */
#define DISKS 3  /*  Number of disks.  */

int pos[3][DISKS];  /*  Disk position array, [rows][columns].  */
char code[3] = {'A', 'B', 'C'};  /*  Tower names.  */
void towers( int n, int source, int temporary, int destination );
void moveDisk( int source, int destination );
int main( int argc, char *argv[] )
{
    int i=0, j=0, hold = 0;
    printf( "\n\n  The Towers of Hanoi: %d Disks\n\n", DISKS );
    /*  Initialize disk positions.  */
    for( i = 0; i < 3; ++i )
    {
        for( j = 0; j < DISKS; ++j )
            if( i == 0 ) pos[ i ][ j ] = j + 1;
            else pos[ i ][ j ] = EMPTY;
    }
   towers( DISKS, 0, 1, 2 );
   return 0;
}
```

```c
void moveDisk( int source, int destination )
{
        int i = 0, j= 0;
        /*  Determine source location.  */
        while( pos[ source ][ i ] == EMPTY )    { i++; }
         /*  Determine destination location.  */
        while(( pos[ destination ][ j ] == EMPTY ) && ( j < DISKS ))    { j++; }
        j -= 1;
        /*  Move disk.  */
        printf( "\n  Move disk #%d from %c to %c:\n\n",
        pos[ source ][ i ], code[ source ], code[ destination ] );
        pos[ destination ][ j ] = pos[ source ][ i ];
        pos[ source ][ i ] = EMPTY;
        /*  Print disk positions after move.  */
        printf( "\n\n          A B C\n" );
        printf( "          - - -\n" );
        for( j = 0; j < DISKS; ++j )
        {
           printf( "%11.1d %d %d\n", pos[ 0 ][ j ], pos[ 1 ][ j ], pos[ 2 ][ j ] );
        }
        printf( "\n" );
        return;
}
```

```c
void towers( int n, int source, int temporary, int destination )
{
   if ( n == 1 )  /*  Base case.  */
      moveDisk( source, destination );
    else
    {
            towers( n - 1, source, destination, temporary );
            moveDisk( source, destination );
            towers( n - 1, temporary, source, destination );
    }
   return;
}
```

Merge Sort



auxiliary array (double the size of A or B)
0
1
2
3
smaller
smaller
smaller
smaller
comp
comp
comp
comp
0
3
4
6
8
11
12
13
A
1
2
5
7
9
10
14
15
B
0
3
4
6
8
11
12
13
A
1
2
5
7
9
10
14
15
B
0
3
4
6
8
11
12
13
A
1
2
5
7
9
10
14
15
B
0
3
4
6
8
11
12
13
A
1
2
5
7
9
10
14
15
B