Recursion

"To understand recursion, one must first understand recursion."

-*Stephen Hawking*

What is recursion?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first
- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

Recursion

More than programming technique:

- a way of describing, defining, or specifying things.
- a way of designing solutions to problems (divide and conquer).

Basic Recursion

1. Base cases:
   – Always have at least one case that can be solved without using recursion.

2. Make progress:
   – Any recursive call must make progress toward a base case.

Mathematical Examples

- Power Function
- Fibonacci Sequence
- Factorial Function

Power Function

There are recursive definitions for many mathematical problems:

- The function **`Power`** (used to raise the number *y* to the *x*th power).
- Assume *x* is a non-negative integer:

$\mathbf{y^x = 1}$, if x is 0 // base case

$\mathbf{y^x = y*y^{(x-1)}}$, otherwise // make progress

Power Function

$2^3 = 2*\mathbf{2^2} \qquad = 2 * \mathbf{4} = 8$

$2^2 = 2*\mathbf{2^1} \qquad = 2 * \mathbf{2} = 4$

$2^1 = 2*\mathbf{2^0} \qquad = 2 * \mathbf{1} = 2$

$2^0 = \mathbf{1}$

Fibonacci Sequence

Fibonacci Sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Function:

- `Fib(0) = 1         // base case`
- `Fib(1) = 1         // base case`
- `Fib(n) =  Fib(n-1) + Fib(n-2)  // n>1`

Unlike most recursive algorithms:

- two base cases, not just one
- two recursive calls, not just one

Factorial

Factorial Function

- `factorial(0) = 1`
- `factorial(n) = n * factorial(n-1) // n > 0`

Compute factorial(3).

Factorial

Factorial Function

- `factorial(0) = 1`
- `factorial(n) = n * factorial(n-1) // n > 0`

Compute factorial(3)

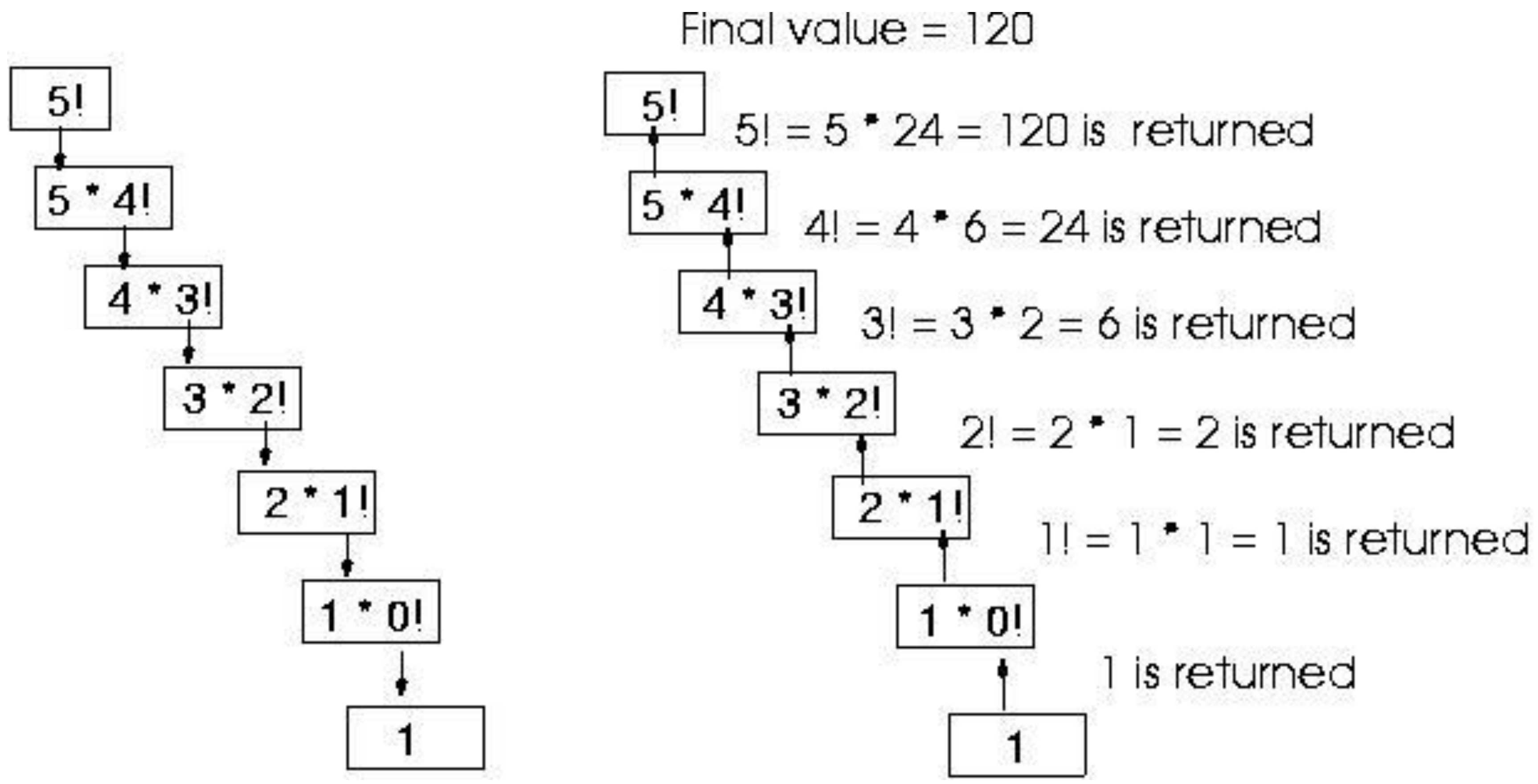
```
factorial(3) = 3 * factorial(2)
             = 3 * ( 2 * factorial(1) )
             = 3 * ( 2 * ( 1 * factorial(0) ) )
             = 3 * ( 2 * ( 1 *      1      ) ))   = 6
```

Coding the Factorial Function

Recursive Implementation

```
int factorial(int n)
{
  if (n==0)  // base case
        return 1;
  else
        return n * factorial(n-1);
}
```

Recursive Call Stack



Implementing Recursion

What happens when a function gets called?

```
// a method
int b(int x)
{
 int z,y;

 ……………… // other statements

 z = a(x) + y;

 return z;
}

// another method
int a(int w)
{
 return w+w;
}
```

When a Function is Called

- Stop executing function ***b***
- So can return to function ***b*** later, need to store everything about function **b**
  - Create **activation** record
  - Includes values of variables **x, y, z**
  - The place to start executing upon return
- Push **activation** record onto the **call stack**
- Then, **a** is bounded to **w** from **b**
- Control is transferred to function **a**

When a Function is Called

After function **a** is executed, the activation record is popped out off call stack

- Values of the parameters and variables in function **b** are restored
- Return value of function **a** replaces **a(x)** in the assignment statement

Recursion vs. Iteration

- *Recursion* is based upon calling the same function over and over.
- *Iteration* simply `jumps back' to the beginning of the loop.

A function call is usually more expensive than a jump.

Recursion vs. Iteration

- *Iteration* can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- *Recursive* solutions are often less efficient
  - in terms of both *time* and *space*
- *Recursion* may simplify the solution
  - shorter, more easily understood source code

Recursion to Iteration Conversion

- Most recursive algorithms can be translated into iterative algorithms.
- Sometimes this is very straightforward
  - most compilers detect a special form of recursion, called **tail** recursion, and translate into iteration automatically.
- Sometimes, the translation is more involved
  - May require introducing an explicit stack with which to ‘fake’ the effect of recursive calls.

Coding Factorial Function

Iterative implementation

```
int factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Other Recursive Examples

- Combinations
- Euclid's Algorithm
- Binary Search

Combinations: *n* choose *k*

Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \quad \textit{(recursive solution)}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \quad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \ \ (k = 1), \quad \binom{n}{n} = 1 \ \ (k = n)$$

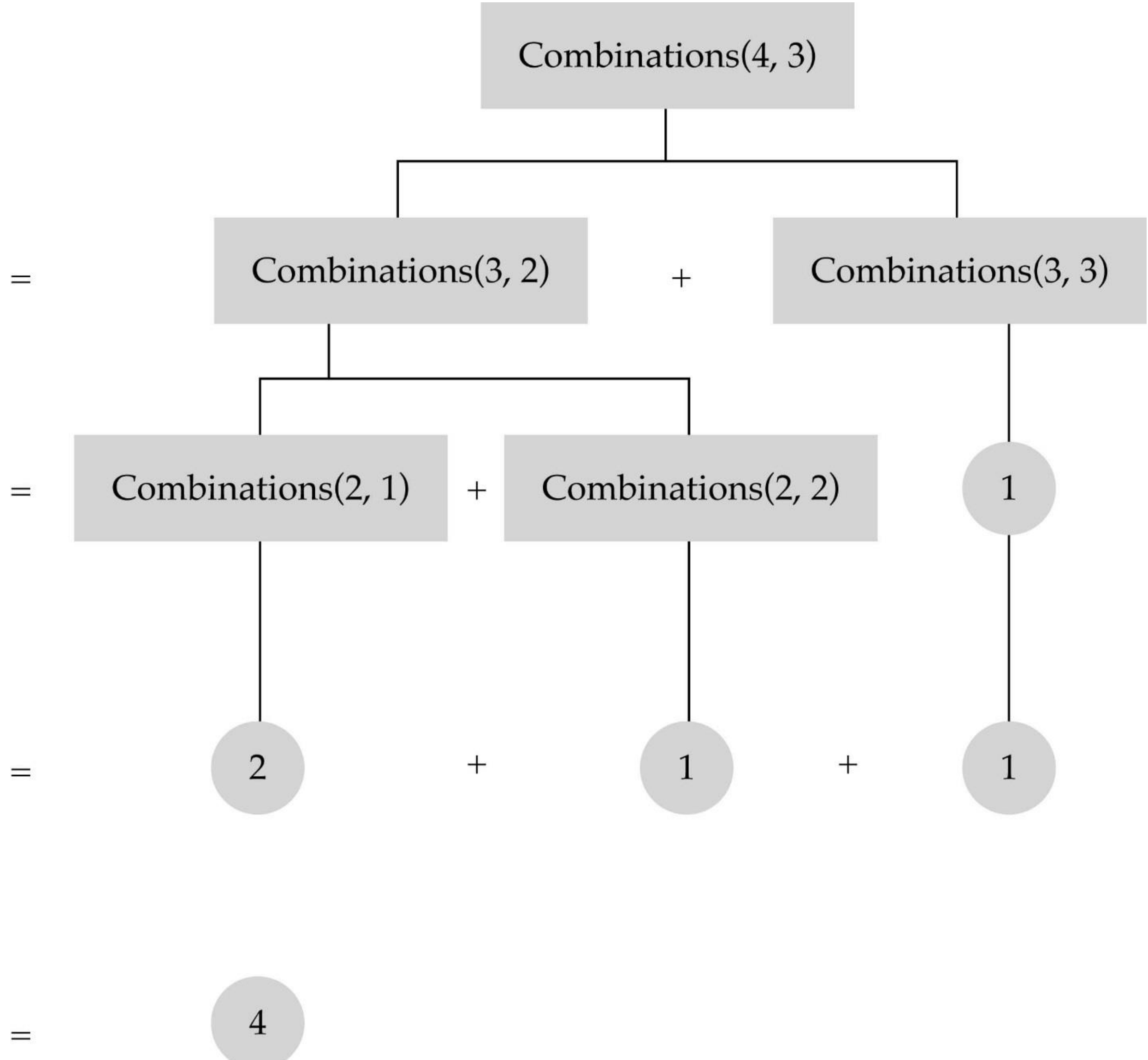
					1					
				1		1				
			1		2		1			
		1		3		3		1		
	1		4		6		4		1	
1		5		10		10		5		1

Pascal's Triangle

Combinations: *n* choose *k*

```
int combinations(int n, int k)
{
  if(k == 1)             // base case 1
      return n;
  else if (n == k)       // base case 2
      return 1;
  else
      return(combinations(n-1, k) +
                   combinations(n-1, k-1));
}
```

Combinations:



Euclid's Algorithm

In about 300 BC, Euclid wrote an algorithm to calculate the greatest common divisor (GCD) of two numbers x and y where ($x < y$). This can be stated as:

1. Divide y by x with remainder r.
2. Replace y by x, and x with r.
3. Repeat step 1 until r is zero.

When this algorithm terminates, y is the highest common factor.

GCD(34017, 16966)

Euclid's algorithm works as follows:

- 34,017/16,966 produces a remainder 85
- 16,966/85 produces a remainder 51
- 85/51 produces a remainder 34
- 51/34 produces a remainder 17
- 34/17 produces a remainder 0

The highest common divisor of 34,017 and 16,966 is 17.

Writing a Recursive Function

Determine the <u>base case(s)</u>

(the one for which you know the answer)

Determine the <u>general case(s)</u>

(the one where the problem is expressed as a smaller version of itself)

Verify the algorithm

(use the "Three-Question-Method")

Three-Question Method

1. The Base-Case Question:

   Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?

2. The Smaller-Caller Question:

   Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

3. The General-Case Question:

   Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Binary Search

- Search algorithm
  - Finds a target value within a sorted list.
  - Compares target value to the middle element
    - If the two are equal, done.
    - If target less than middle element, search lower half of list. Otherwise, search upper half of list.
    - Continue dividing list in half until find target or run out of list to search.
- Efficiency:
  - Runs in at worst logarithmic $O(\log n)$ time
  - Takes up linear $O(n)$ space

Recursive Binary Search

What is the *base case(s)*?

1. If *first > last*, return *false*
2. If *item==info[midPoint]*, return *true*

What is the *general case*?

```
    if item < info[midPoint]
        // search the first half
   if item > info[midPoint],
        //search the second half
```

Recursive Binary Search

```
boolean binarySearch(Item info[], Item item, int first, int last)
{
   int midPoint;

   if(first > last)  // base case 1
       return false;
   else
   {
         midPoint = (first + last)/2;
         if(item < info[midPoint])
               return BinarySearch(info, item, first, midPoint-1);
         else if (item == info[midPoint])
         { // base case 2
               item = info[midPoint];
               return true;
         }
         else
               return binarySearch(info, item, midPoint+1, last);
   }
}
```

When to Use Recursion

- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the non-recursive version
- The recursive version is shorter and simpler than the non-recursive solution

Benefits of Recursion

- Recursive functions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.
- The program directly reflects the abstract solution strategy (algorithm).
- Reduces the cost of maintaining the software.

Disadvantages of Recursion

- Makes it easier to write simple and elegant programs, but it also makes it easier to write inefficient ones.
- Use recursion to ensure correctness, not efficiency. My simple, elegant recursive algorithms are inherently inefficient.

Recursion Overhead

- Space:
  - Every invocation of a function call requires:
    - space for parameters and local variables
    - space for return address
  - Thus, a recursive algorithm needs space proportional to the number of nested calls to the same function.

Recursion Overhead

- Time:
  - Calling a function involves
    - allocating, and later releasing, local memory
    - copying values into the local memory for the parameters
    - branching to/returning from the function

All contribute to the time overhead.