Data Structures and Algorithms
CS-206

Recursion

Recursive Thinking

- ***Recursion*** is:
  - A *problem-solving **approach***, that can ...
  - Generate *simple solutions* to ...
  - *Certain kinds* of problems that ...
  - Would be *difficult to solve in other ways*
- Recursion *splits a problem*:
  - Into one or more *simpler versions of **itself***

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

Recursive Definitions of Mathematical Formulas

- Mathematicians often use *recursive definitions*
- These lead very naturally to *recursive algorithms*
- Examples include:
  - Factorial
  - Powers
  - Greatest common divisor

Terminating Condition

- The recursive functions always contains one or more **terminating conditions**.
  - A condition when a recursive function is processing a simple case instead of processing recursion.
- Without the terminating condition, the recursive function may run forever.

Properties of Recursive Functions

Problems that can be solved by recursion have these characteristics:

- One or more terminating cases have a simple, nonrecursive solution
- The other cases of the problem can be reduced (using recursion) to problems that are closer to terminating cases
- Eventually the problem can be reduced to only terminating cases, which are relatively easy to solve

Follow these steps to solve a recursive problem:

- Try to express the problem as a simpler version of itself
- Determine the terminating cases
- Determine the recursive steps

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

Calculate Factorial

Example: Factorial

- The factorial is defined in this way:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \cdots \times n & n > 0 \end{cases}$$

This can be computed by a loop.

- We can also define the factorial as:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

So, if we know how to compute the factorial of $n-1$, we know how to compute the factorial of n.

This assumes that N >= 0 and that the factorial of 0 is 1 and the factorial of 1 is also 1 (a non-recursive definition).

Calculate Factorial using Recursion

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

Calculate Factorial using Recursion

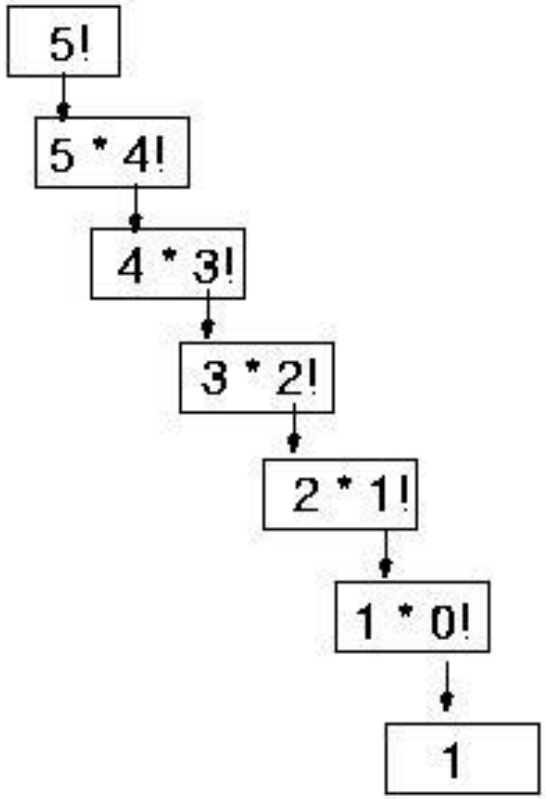
Example: Factorial

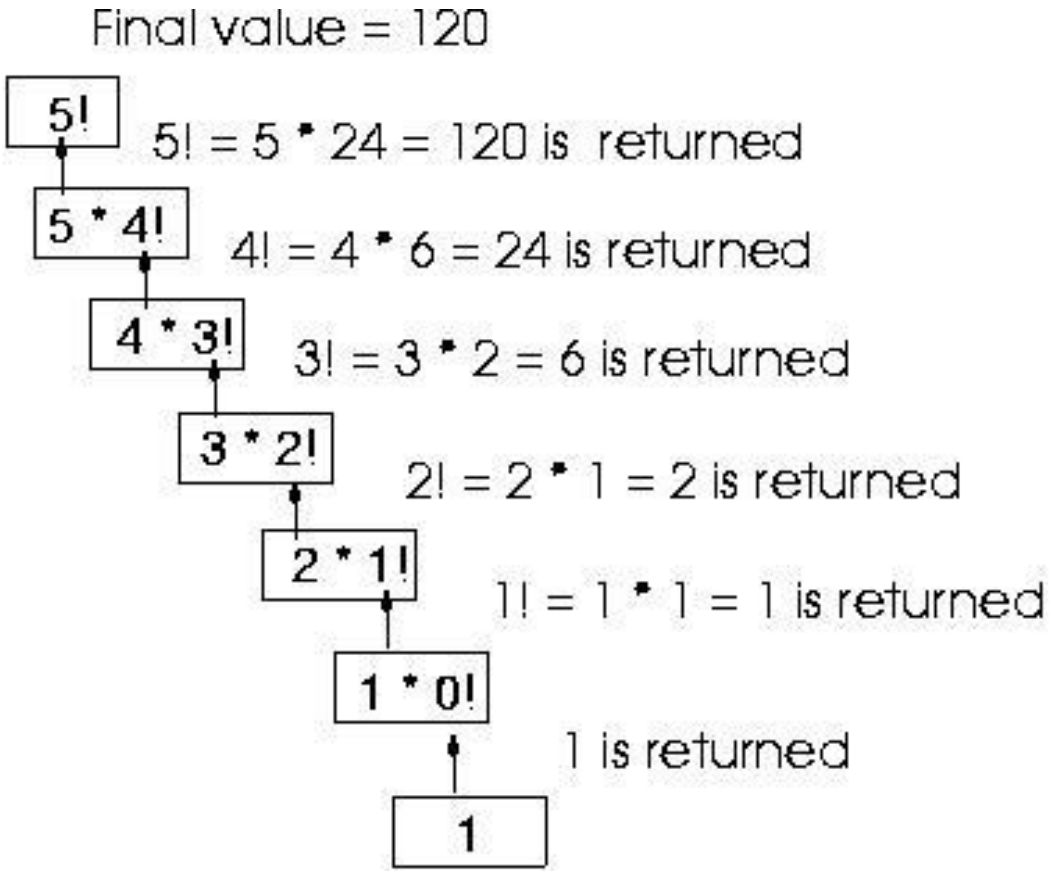
```
int factorial(int n)  // assumes n >= 0
{
  if (n == 0)
    return 1;   //base case
  else
    return n * factorial(n-1);
}
```

To see how the computation is done, trace `factorial(3)`:

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

Calculate 5!


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1


Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1! = 1 * 1 = 1 is returned
1 is returned

Recursion in Action: *factorial(n)*

```
factorial (5) = 5 x factorial (4)
              = 5 x (4 x factorial (3))
              = 5 x (4 x (3 x factorial (2)))
              = 5 x (4 x (3 x (2 x factorial (1))))
              = 5 x (4 x (3 x (2 x (1 x factorial (0)))))
              = 5 x (4 x (3 x (2 x (1 x 1))))
              = 5 x (4 x (3 x (2 x 1)))
              = 5 x (4 x (3 x 2))
              = 5 x (4 x 6)
              = 5 x 24
              = 120
```

Base case arrived
Some concept from elementary maths: Solve the inner-most bracket, first, and then go outward

Coding the factorial function (cont.)

- Iterative implementation

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

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

 return fact;
}
```

Iterative vs Recursive factorial

Iterative

1. 2 local variables
2. 3 statements
3. Saves solution in an intermediate variable

Recursive

1. No local variables
2. One statement
3. Returns result in single expression

Recursion simplifies factorial by making the computer do the work.

What work and how is it done?

How does it work?

Each recursive call creates an “activation record” which contains

local variables and parameters

return address

and is saved on the stack.

Another example:
n choose *k* (combinations)

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

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

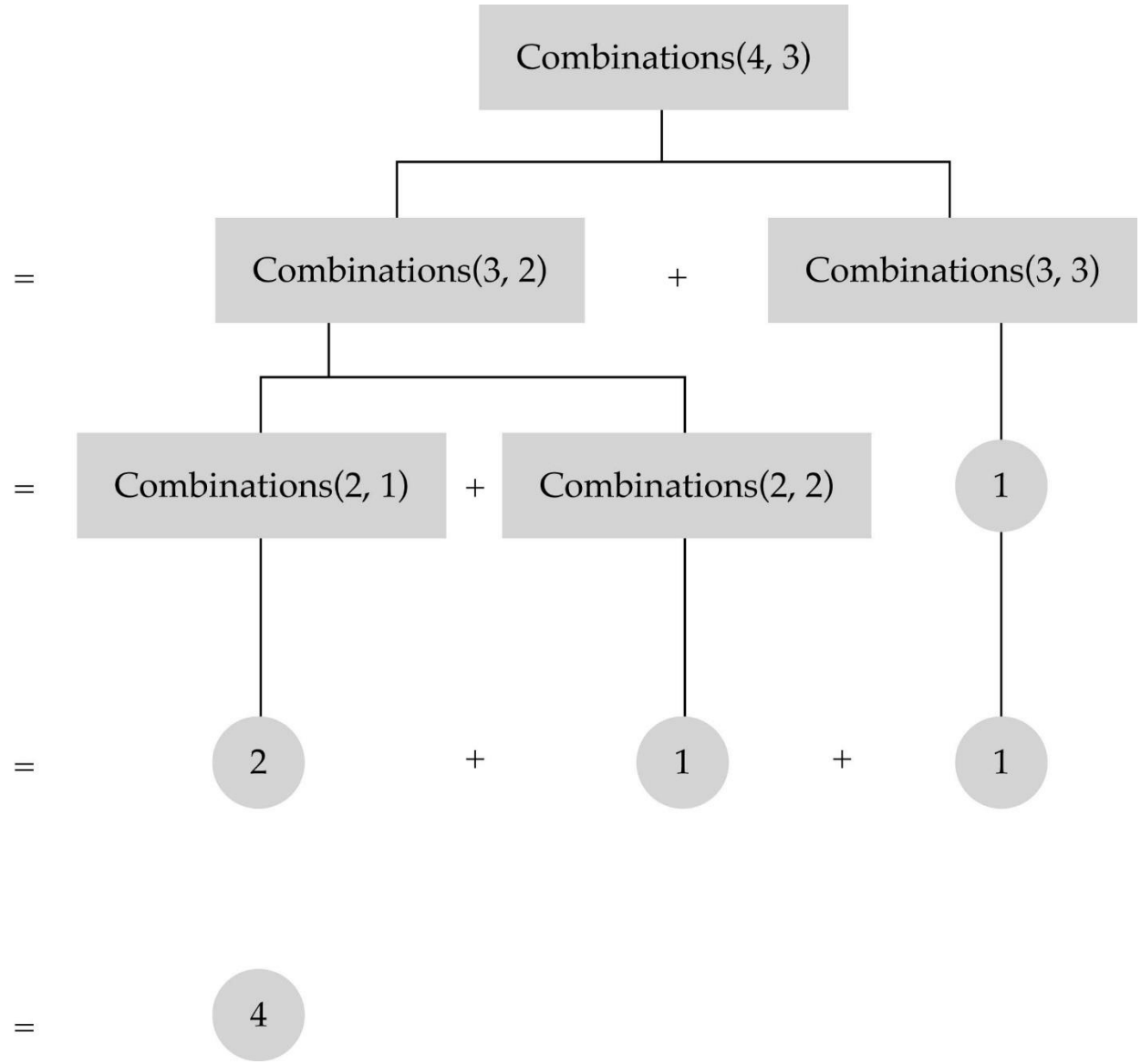
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

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

n choose *k* (combinations)

```
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(4, 3)
=
Combinations(3, 2)
+
Combinations(3, 3)
=
Combinations(2, 1)
+
Combinations(2, 2)
1
=
2
+
1
+
1
=
4

- Control structures may be categorized as either *sequential* structures, *selection* structures or *repetition* structures.
- Recursion is a technique by which to achieve repetition.
  - Iterative loops such as while, do while and for are the other technique.
- Any algorithm that can be expressed iteratively can also be expressed recursively, and visa versa.

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*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How do I write a recursive function?

- Determine the size factor
- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

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

  (use the "Three-Question-Method")

Three-Question Verification Method

The Base-Case Question:

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

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?

The General-Case Question:

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

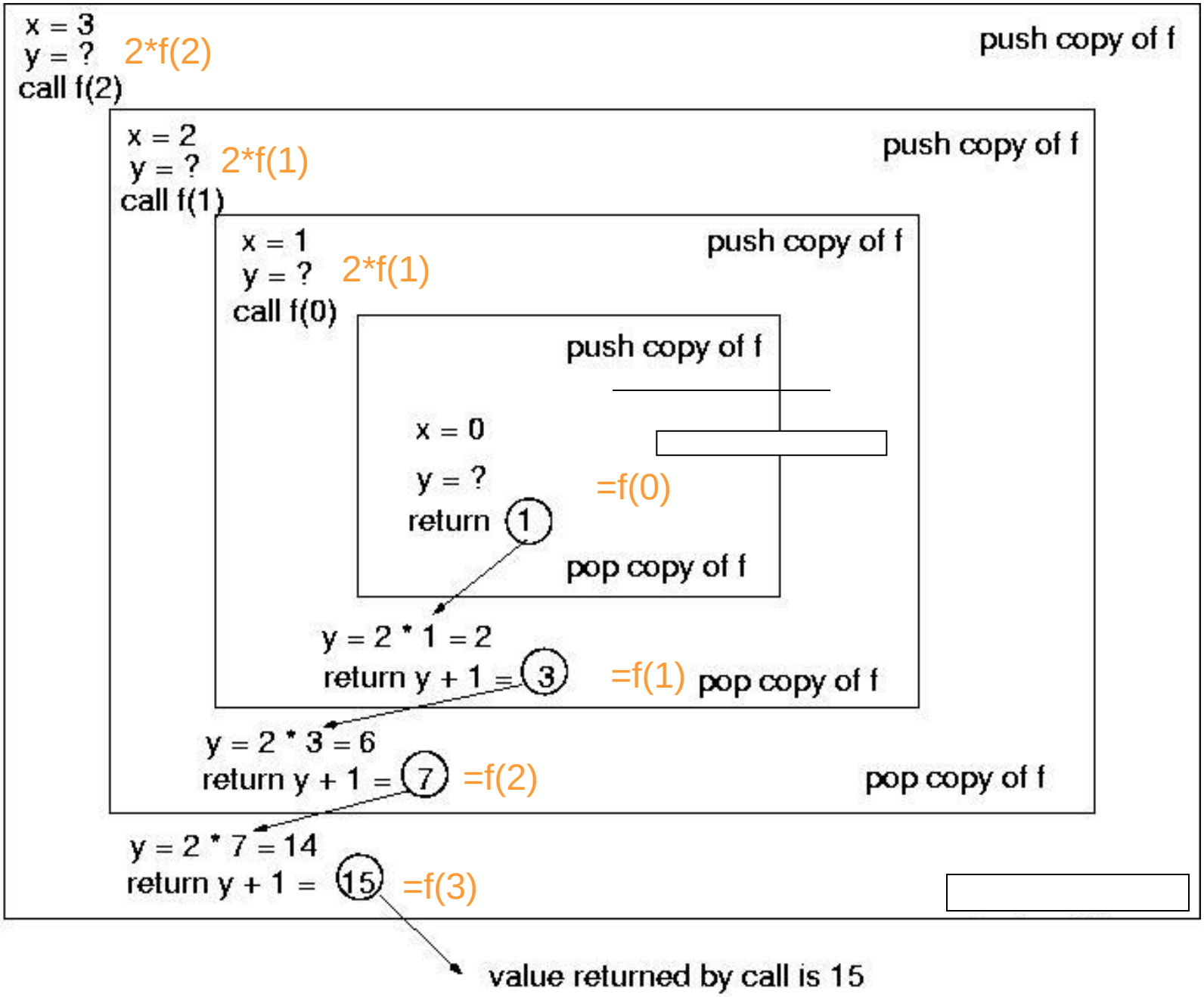
- The recursive call must not skip over the base case.

What happens when a recursive function is called?

- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
 int y;

 if(x==0)
  return 1;
 else {
   y = 2 * f(x-1);
   return y+1;
}
}
```


x = 3
y = ?
call f(2)
2*f(2)
push copy of f
x = 2
y = ?
call f(1)
2*f(1)
push copy of f
x = 1
y = ?
call f(0)
2*f(1)
push copy of f
x = 0
y = ?
return 1
=f(0)
pop copy of f
y = 2 * 1 = 2
return y + 1 = 3
=f(1)
pop copy of f
y = 2 * 3 = 6
return y + 1 = 7
=f(2)
pop copy of f
y = 2 * 7 = 14
return y + 1 = 15
=f(3)
value returned by call is 15

Towers of Hanoi

- This is a standard problem where the recursive implementation is trivial but the non-recursive implementation is almost impossible.

Practice Exercise

Write program on

- Summing the elements of an array recursively
- Finding the maximum element in an array A of n elements using recursion
- Write a recursive function to compute first N Fibonacci numbers. Test and trace for N = 6