CS 2213
Advanced Programming
Ch 4 – Recursion

Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

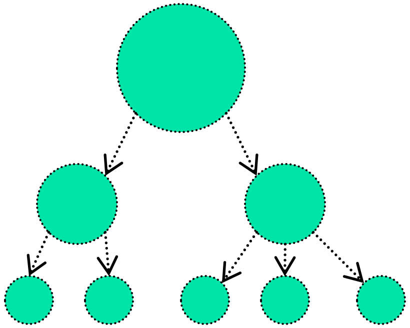
Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.

Objectives

- To be able to define the concept of ***recursion*** *as a* ***programming strategy*** distinct from other forms of algorithmic decomposition.
- To recognize the ***paradigmatic form*** of a recursive function.
- To understand the internal implementation of recursive calls.
- To appreciate the importance of the ***recursive leap of faith***.
- To understand the concept of ***wrapper functions*** in writing recursive programs.
- To be able to write and debug simple recursive functions at the level of those presented in this chapter.

Recursion:

One of the most important “Great Ideas”



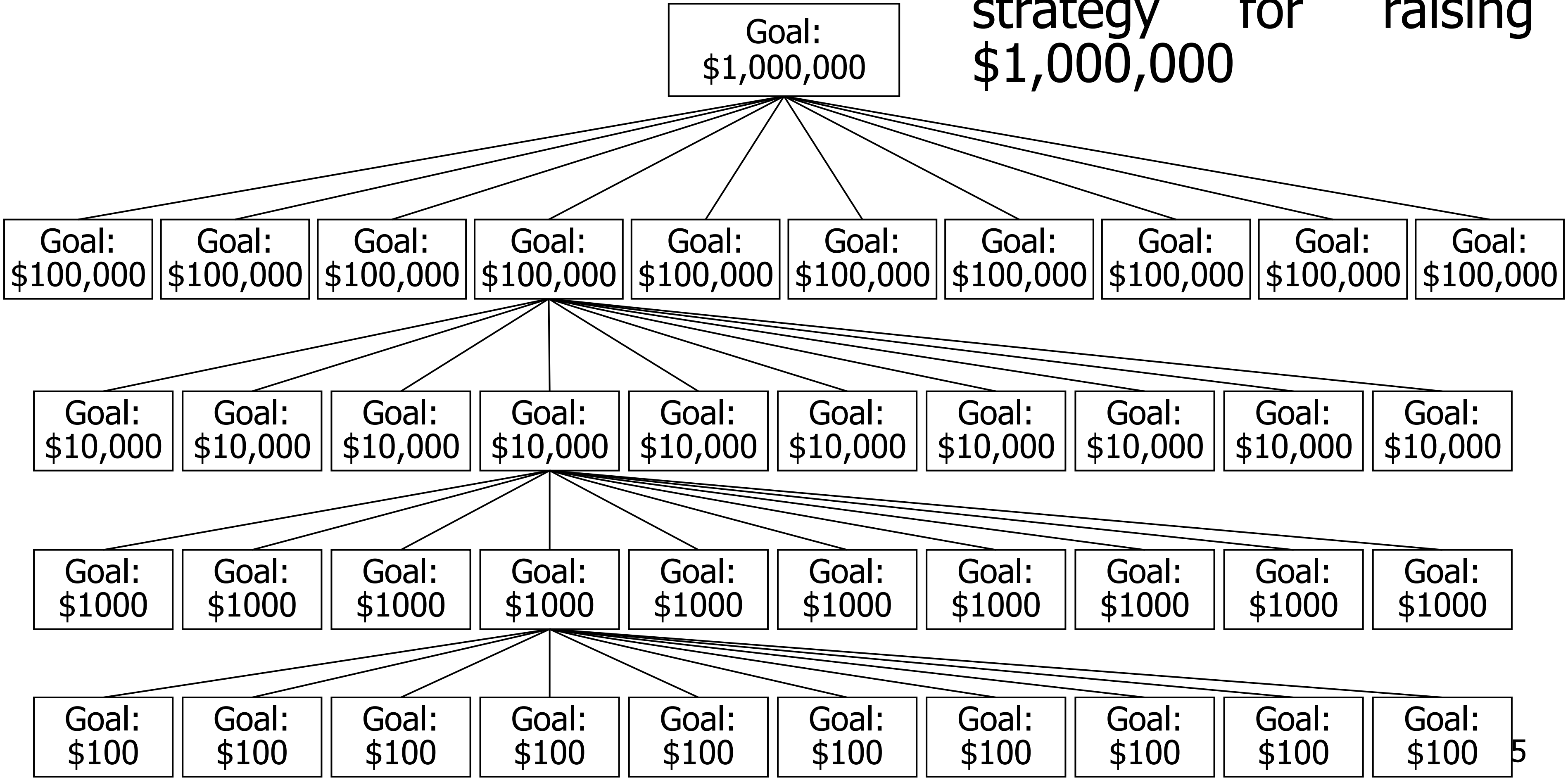
- Recursion is the process of solving a problem by dividing it into smaller ***sub-problems of the same form***.
- The italicized phrase is the essential characteristic of recursion; without it, all you have is a description of stepwise refinement of the solution.
- Since the recursive decomposition generates sub-problems that have the same form as the original problem, **we can use the same function** or method to solve the generated sub-problems at different levels.
- In terms of the structure of the code, the defining characteristic of recursion is **having functions that call themselves**, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution.
  - Individuals with that much money are difficult to find.
  - Donors are much more likely to donate in the $100 range.
- Another strategy would be to ask 10,000 friends for $100 each. But, most of us don't have 10,000 friends.
- There are, however, more promising strategies.
  - You could, for example, find ten regional coordinators and charge each one with raising $100,000.
  - Those regional coordinators could in turn delegate the task to local coordinators, each with a goal of $10,000, continuing the process reached a manageable contribution level.

A Simple Illustration of Recursion (cont'd)

The following diagram illustrates the recursive strategy for raising $1,000,000



A Pseudocode for Fundraising Strategy

```
void CollectContributions(int n) {
    if (n <= 100) {
        Collect the money from a single donor.
    } else {
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
    }
}
```

What makes this strategy recursive is that the line

Get each volunteer to collect n/10 dollars.

will be implemented using the following recursive call:

```
CollectContributions(n / 10);
```

Recursive Paradigm: Writing a Recursive Function

```
if (test for simple case) {
  Compute a simple solution without using recursion
} else {
  Break the problem down into sub-problems of the same form.
  Solve each of the sub-problems by calling this function recursively.
  Reassemble the solutions to the sub-problems into a solution for the whole.
}
```

Finding a recursive solution is mostly a matter of figuring out **how to break it down** so that it fits the paradigm. When you do so, you must do two things:

1. Identify simple case(s) that can be solved without recursion.
2. Find a recursive decomposition that breaks each instance of the problem into simpler sub-problems of the same type, which you can then solve by applying the method recursively.

The recursive formulation of Factorial

- *n! = n x (n – 1)!*
- Thus, 4! is 4 x 3!, 3! is 3 x 2!, and so on.
- To make sure that this process stops at some point, mathematicians define 0! to be 1.
- Thus, the conventional mathematical definition of the factorial looks like

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

Recursive vs. iterative implementation

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

```
int FactIteration(int n)
{
  int product;
  product = 1;
  for (int i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

Tracing Factorial Function

Local variables and return addresses are stored in a stack.

```
int main() {
  int Fact(int n) {
    int Fact(int n) {
      int Fact(int n) {
        int Fact(int n) {
          if (n == 0) {
            return 1;
          } else {
            return n * Fact(n - 1);
          }
        }
```

n: 0

Fact

```
Enter n: 3
3! = 6
```

skip simulation

The Recursive “Leap of Faith”

- The purpose of going through the complete decomposition of factorial is to convince you that the **process works** and that recursive calls are in fact **no different from other method calls**, at least in their internal operation.
- The danger with going through these details is that it might encourage you to do the same when you write your own recursive programs. As it happens, **tracing through the details of a recursive program almost always makes such programs harder to write.**
- Writing recursive programs becomes natural only after you have enough confidence in the process that **you don’t need to trace them fully**.
- As you write a recursive program, it is important to believe that **any recursive call will return the correct answer** as long as the arguments define a simpler sub-problem.
- Believing that to be true—even before you have completed the code—is called **the recursive leap of faith**.

t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
0	1	1	2	3	5	8	13	21	34	55	89	144

The Fibonacci function

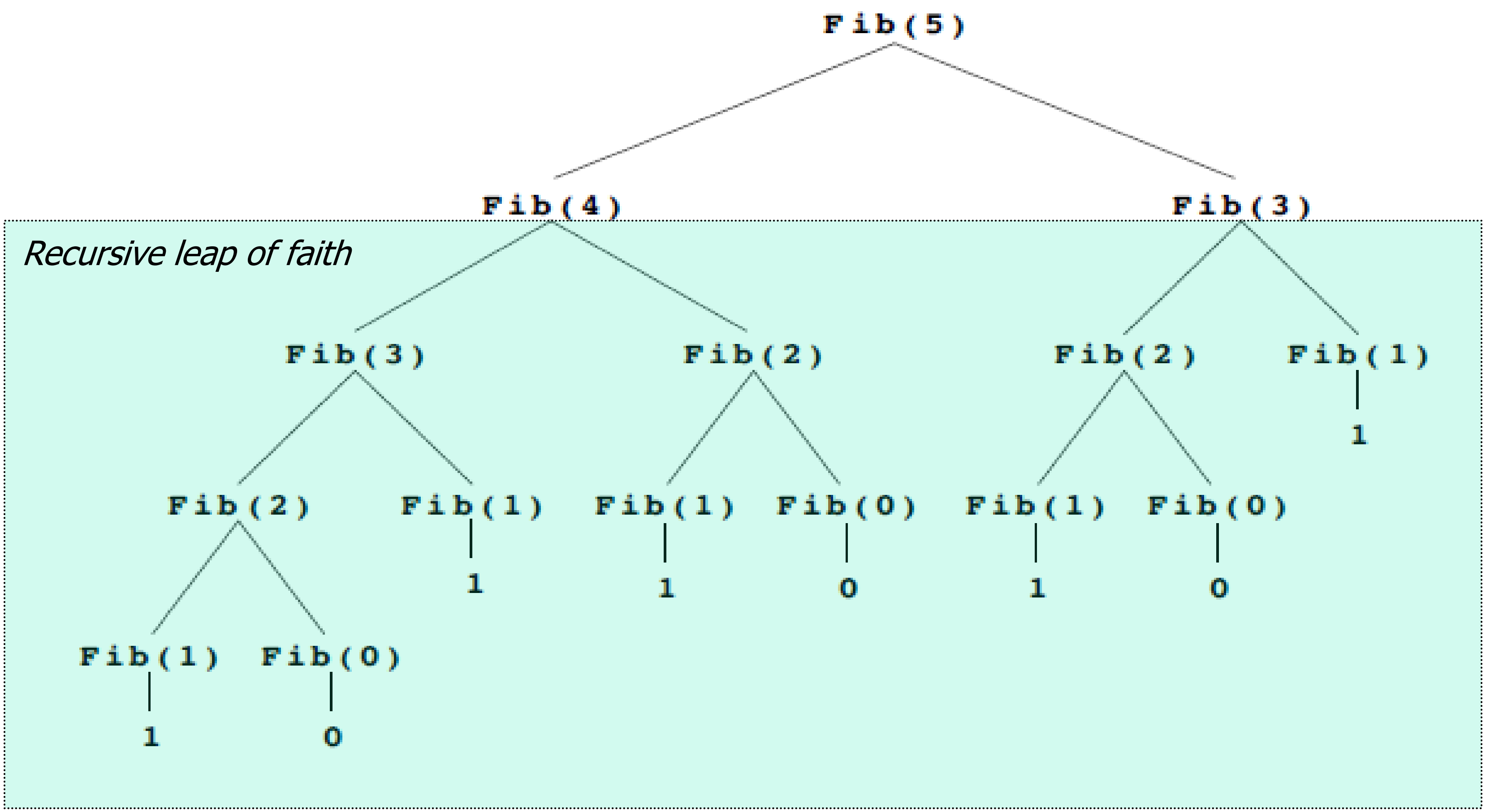
$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

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

How about

```
int FibIteration(int n)
{
… // dynamic programming
}
```

Steps in the calculation of Fib(5)



Efficiency of the Recursive implementation of Fib

- Can you implement an iterative version of Fib, say `int IterativeFib(int n)`?
- Which one will be faster Recursive or Iterative?
- Look at the details of Fib(5) in previous slide:
  - you will see that it is extremely inefficient
  - the same Fib term is computed many times (redundant calls to Fib())
- Should we blame Recursion!
- Can we fix this?

Wrapper function and a **subsidiary function** for the more general case

- Suppose we have the following function

```
int AdditiveSequence(int n, int t0, int t1)
{
  if (n == 0) return t0;
  if (n == 1) return t1;
  return AdditiveSequence(n-1, t1, t0+t1);
}
```

subsidiary function for the more general case

- Then we can simply implement Fib(n) as

```
int Fib(int n) {
    return AdditiveSequence(n, 0, 1);
}
```

Wrapper function

Trace and Efficiency of Fib

Fib(5)
= AdditiveSequence(5, 0, 1)
= AdditiveSequence(4, 1, 1)
= AdditiveSequence(3, 1, 2)
= AdditiveSequence(2, 2, 3)
= AdditiveSequence(1, 3, 5)
= 5

- The new implementation is entirely recursive, and it is comparable in efficiency to the standard iterative version of the Fib() function.

Common Errors

- Recursive function may not terminate if the stopping case is not correct or is incomplete
  - stack overflow: run-time error
- Make sure that each recursive step leads to a situation that is closer to a stopping case.

(problem size gets smaller and smaller and smaller and smaller)

Iteration vs. Recursion

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. Why?
  - This is because the overhead involved in entering and exiting a function is avoided in iterative version.
- However, a recursive solution can be sometimes the most natural and logical way of solving a problem (tree traversal).
- Conflict:
  - machine efficiency vs. programmer efficiency
- It is always true that recursion can be replaced with iteration and a stack.

Mutual Recursion

- So far, the recursive functions have called themselves directly
- But, the definition is broader:
  - To be recursive, a function must call itself at some point during its evaluation.
  - For example, if a function *f* calls a function *g, which in turn calls f, the function calls are still considered to be* recursive.
- The recursive call is actually occurring at a deeper level of nesting.

Mutual Recursion Example

```
bool IsEven(unsigned int n) {
  if (n == 0) {
    return true;
  } else {
    return IsOdd(n - 1);
  }
}

bool IsOdd(unsigned int n) {
   return !IsEven(n);
}
```

Study Other examples in Section 4.4

- A **palindrome** is a string that reads identically backward and forward, such as "level” or "noon".
  - it is easy to check whether a string is a palindrome by iterating through its characters,
  - Palindromes can also be defined recursively.
  - Insight: any pali
    must contain a s
  - For example, **"le**
    with an **"l"** at ea
- Binary Search

```
bool IsPalindrome(string str) {
   int len = strlen(str);
   if (len <= 1) {
      return true;
   } else {
     return (str[0] == str[len – 1]  &&
                 IsPalindrome(SubString(str,1, len - 2)));
   }
}  // see the textbook using wrapper function
```

More Recursive Examples in Ch 5

- Tower of Hanoi (Self-Study)
- **Generating Permutations**
- Graphical applications (Self-Study)

Exercise: A Recursive **GCD** Function

One of the oldest known algorithms that is worthy of the title is Euclid's algorithm for computing the greatest common divisor (GCD) of two integers, *x* and *y*. Euclid's algorithm is usually implemented iteratively using code that looks like this:

```
int GCD(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

Rewrite this method so that it uses recursion instead of iteration, taking advantage of Euclid's insight that the greatest common divisor of *x* and *y* is also the greatest common divisor of *y* and the remainder of *x* divided by *y*.