Introduction to Programming (in C++)

Recursion

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

Recursion

- A subprogram is recursive when it contains a call to itself.
- Recursion can substitute iteration in program design:
  - Generally, recursive solutions are simpler than (or as simple as) iterative solutions.
  - There are some problems in which one solution is much simpler than the other.
  - Generally, recursive solutions are slightly less efficient than the iterative ones (if the compiler does not try to optimize the recursive calls).
  - There are *natural* recursive solutions that can be extremely inefficient. Be careful !

Factorial

```
// Pre: n >= 0
// Returns n!

int factorial(int n) { // iterative solution
    int f = 1;
    int i = 0;
    // Invariant: f = i! and i ≤ n
    while (i < n) {
        i = i + 1;
        f = f*i;
    }
    return f;
}
```


Factorial

- Definition of factorial:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

- Recursive definition:

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

Factorial

```
// Pre: n >= 0
// Returns n!

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


Recursive design

In the design of a recursive program, we usually follow a sequence of steps:

1. Identify the basic cases (those in which the subprogram can solve the problem directly without recurring to recursive calls) and determine how they are solved.

   For example, in the case of factorial, the only basic case used in the function is n=0. Similarly, we could have considered a more general basic case (e.g., n ≤ 1). In both cases, the function should return 1.

Recursive design

2. Determine how to resolve the non-basic cases in terms of the basic cases, which we assume we can already solve.

   In the case of a factorial, we know that the factorial of a number n greater than zero is n∗factorial(n-1).

3. Make sure that the parameters of the call move closer to the basic cases at each recursive call. This should guarantee a finite sequence of recursive calls that always terminates.

   In the case of a factorial, n-1 is closer to 0 than n. Therefore, we can guarantee that this function terminates.

Recursive design

- For example, it is not clear whether the following function terminates:

```
// Pre: n >= 1
// Returns the number of steps of the Collatz sequence
// that starts with n.

int Collatz(int n) { // recursive solution
    if (n == 1) return 0;
    else if (n%2 == 0)  return 1 + Collatz(n/2);
    else return 1 + Collatz(3*n + 1);
}
```

- The reason is that 3∗n+1 is not closer to 1 than n

Recursion: behind the scenes

```
...
f = factorial(4);
...
```

```
int factorial(int 4)
  if (4 <= 1) return 1;
  else return 4 * factorial( 3 );
```

```
int factorial(int 3)
  if (3 <= 1) return 1;
  else return 3 * factorial( 2 );
```

```
int factorial(int 2)
  if (2 <= 1) return 1;
  else return 2 * factorial( 1 );
```

```
int factorial(int 1)
  if (1 <= 1) return 1;
  else return 1 * factorial(n-1);
```


Recursion: behind the scenes

```
...
f = 24;
...
```

```
24 factorial(int 4)
  if (4 <= 1) return 1;
  else return 24;
```

```
6 factorial(int 3)
  if (3 <= 1) return 1;
  else return 6;
```

```
2 factorial(int 2)
  if (2 <= 1) return 1;
  else return 2;
```

```
1 factorial(int 1)
  if (1 <= 1) return 1;
  else return 1 * factorial(n-1);
```


Recursion: behind the scenes

- Each time a function is called, a new *instance* of the function is *created*. Each time a function "returns", its instance is *destroyed*.
- The creation of a new instance only requires the allocation of memory space for data (parameters and local variables).
- The instances of a function are destroyed in reverse order to their creation, i.e. the first instance to be created will be the last to be destroyed.

Write the binary representation

Design a procedure that, given a number *n*, writes its binary representation.

```
// Pre:  n > 0
// Post: the binary representation of has been written.
void base2(int n) {
```

- Basic case (n=1) → write "1"
- General case (n>1) → write n/2 and then write n%2

Write the binary representation

```
// Pre:  n > 0
// Post: the binary representation of n has been written.

void base2(int n) {
    if (n == 1) cout << n;
    else {
        base2(n/2);
        cout << n%2;
    }
}
```

The procedure always terminates since n/2 is closer to 1 than n. Note that n/2 is never 0 when n > 1. Therefore, the case n = 1 will always be found at the end of the sequence call.

Fibonacci numbers

- Design a function that, given a number n, returns the Fibonacci number of order n.

  The Fibonacci numbers are:

order	0	1	2	3	4	5	6	7	8	9
fib	1	1	2	3	5	8	13	21	34	55

- In general, except for n = 0 and n = 1, the Fibonacci number of order n is equal to the sum of the two previous numbers.

Fibonacci numbers

```
// Pre: n >= 0
// Returns the Fibonacci number of order n.
int fib(int n);
```

- Basic case:

  n = 0 ⇒ Return 1.

  n = 1 ⇒ Return 1.

- General case:

  n > 1 ⇒ Return fib(n - 1) + fib(n - 2)

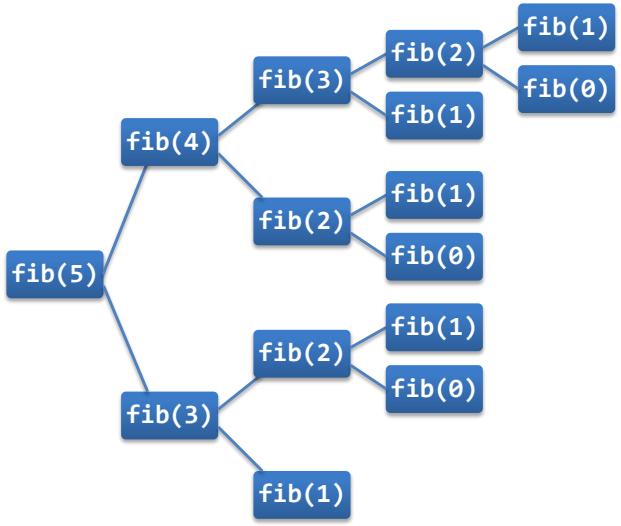
Fibonacci numbers

```
// Pre: n >= 0
// Returns the Fibonacci number of order n.

int fib(int n) { // Recursive solution
    if (n <= 1) return 1;
    else return fib(n - 2) + fib(n - 1);
}
```

The function always terminates since the parameters of the recursive call (n-2 and n-1) are closer to 0 and 1 than n.

Fibonacci numbers

The tree of calls for fib(5) would be:



Fibonacci numbers

- When fib(5) is calculated:
  - fib(5) is called once
  - fib(4) is called once
  - fib(3) is called twice
  - fib(2) is called 3 times
  - fib(1) is called 5 times
  - fib(0) is called 3 times
- When fib(n) is calculated, how many times will fib(1) and fib(0) be called?
- Example: fib(50) calls fib(1) and fib(0) about $2.4 \cdot 10^{10}$ times

Fibonacci numbers

```
// Pre: n >= 0
// Returns the Fibonacci number of order n.

int fib(int n) { // iterative solution
    int i = 1;
    int f_i = 1;
    int f_i1 = 1;
    // Inv: f_i is the Fibonacci number of order i.
    //      f_i1 is the Fibonacci number of order i-1.
    while (i < n) {
        int f = f_i + f_i1;
        f_i1 = f_i;
        f_i = f;
        i = i + 1;
    }
    return f_i;
}
```


Fibonacci numbers

- With the iterative solution, if we calculate fib(5), we have that:
  - fib(5) is calculated once
  - fib(4) is calculated once
  - fib(3) is calculated once
  - fib(2) is calculated once
  - fib(1) is calculated once
  - fib(0) is calculated once

Counting a's

- We want to read a text represented as a sequence of characters that ends with '.'
- We want to calculate the number of occurrences of the letter 'a'
- We can assume that the text always has at least one character (the last '.')
- Example: the text

  Programming in C++ is amazingly easy !.

  has 4 a's

Counting a's

```
// Input:  a sequence of characters that ends with '.'
// Returns the number of times 'a' appears in the
// sequence (and the sequence has been read)
```

- Basic case:
  We have a '.' at the input → return 0
- General case:
  We have something different from '.' at the input → calculate the number of remaining 'a' at the input and add 1 if the current char is an 'a'

Counting a's

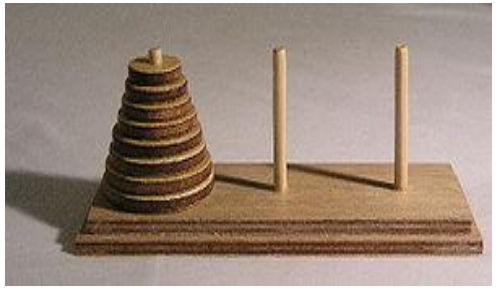
```
// Input: a sequence of characters that ends with '.'
// Returns the number of times 'a' appears in the
// sequence (and the sequence has been read)

int count_a() {
    char c;
    cin >> c;
    if (c == '.') return 0;
    else if (c == 'a') return 1 + count_a();
    else return count_a();
}
```

Even though it has no parameters, we can see that the function terminates if we consider that the input is an implicit parameter. At every recursive call, a new char is read. Therefore, each call moves closer to reading the final dot.

Tower of Hanoi

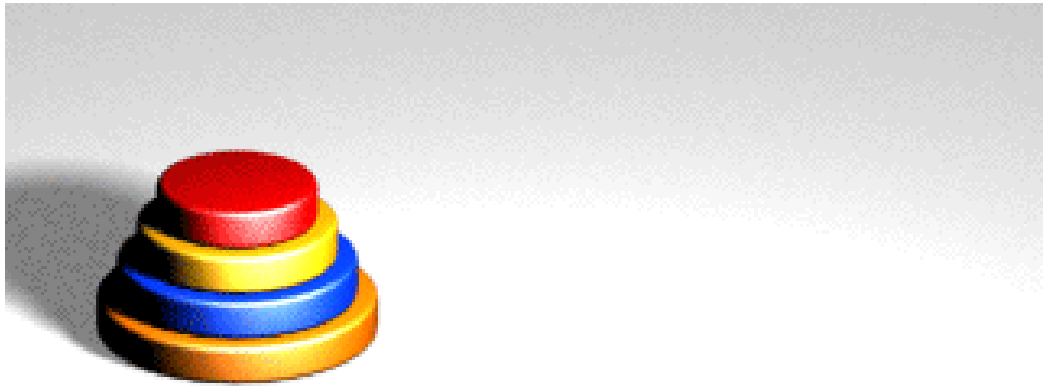
- The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a legend about an Indian temple that contains a large room with three time-worn posts in it, surrounded by 64 golden disks. To fulfil an ancient prophecy, Brahmin priests have been moving these disks, in accordance with the rules of the puzzle, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move in the puzzle is completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.
  (from http://en.wikipedia.org/wiki/Tower_of_Hanoi)
- Rules of the puzzle:
  - A complete tower of disks must be moved from one post to another.
  - Only one disk can be moved at a time.
  - No disk can be placed on top of a smaller disk.

Not allowed !

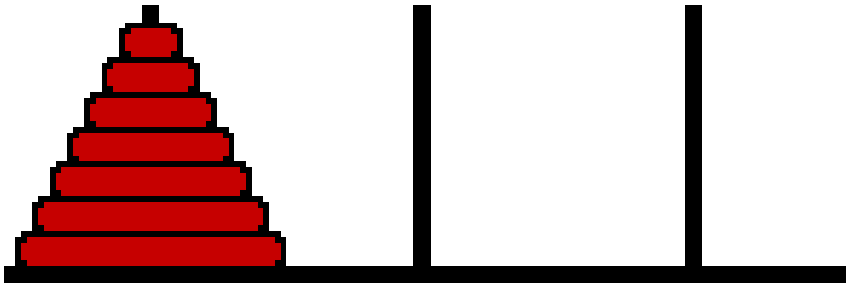


Tower of Hanoi

Tower of Hanoi

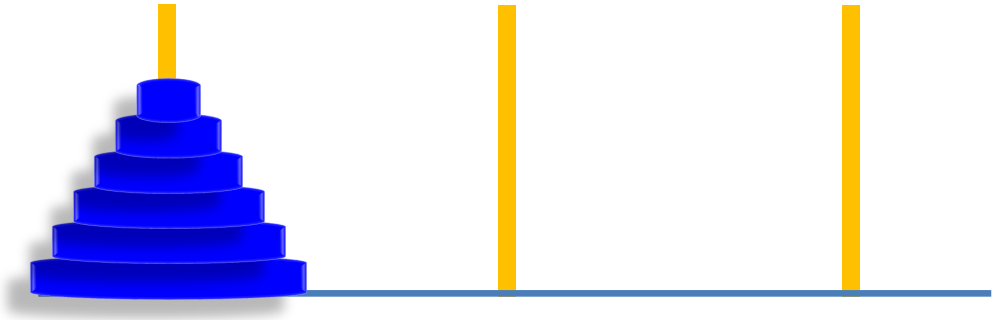


Tower of Hanoi



- What rules determine the next move?
- How many moves do we need?
- There is no trivial iterative solution.

Tower of Hanoi



Inductive reasoning: assume that we know how to solve Hanoi for n-1 disks

- **Hanoi(n-1) from left to middle (safe: the largest disk is always at the bottom)**
- **Move the largest disk from the left to the right**
- **Hanoi(n-1) from the middle to the right (safe: the largest disk is always at the bottom)**

Tower of Hanoi

```
// Pre:  n is the number of disks (n≥0).
//       from, to and aux are the names of the pegs.
// Post: Tower of Hanoi solved by moving n disks
//       from peg from to peg to using peg aux

void Hanoi(int n, char from, char to, char aux) {
    if (n > 0) {
        Hanoi(n - 1, from, aux, to);
        cout << "Move disk from " << from
             << " to " << to << endl;
        Hanoi(n - 1, aux, to, from);
    }
}
```


Tower of Hanoi

```
// Main program to solve the Tower of Hanoi
// for any number of disks

int main() {
    int Ndisks;

    // Read the number of disks
    cin >> Ndisks;

    // Solve the puzzle
    Hanoi(Ndisks, 'L', 'R', 'M');
}
```

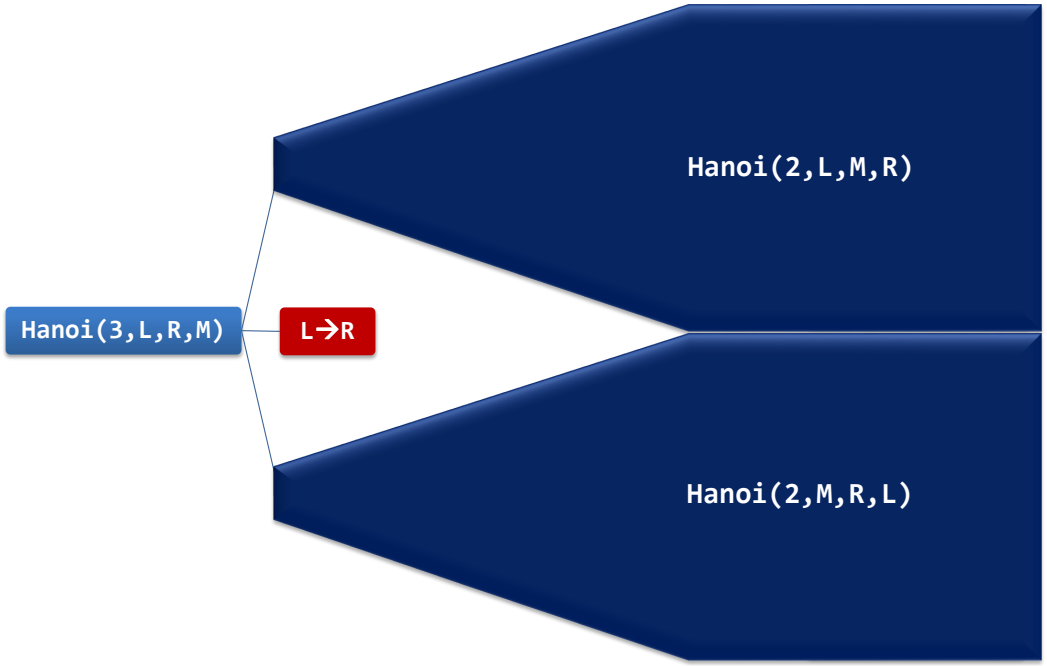

Tower of Hanoi

```
> Hanoi
5
Move disk from L to R
Move disk from L to M
Move disk from R to M
Move disk from L to R
Move disk from M to L
Move disk from M to R
Move disk from L to R
Move disk from L to M
Move disk from R to M
Move disk from R to L
Move disk from M to L
Move disk from R to M
Move disk from L to R
Move disk from L to M
Move disk from R to M
```

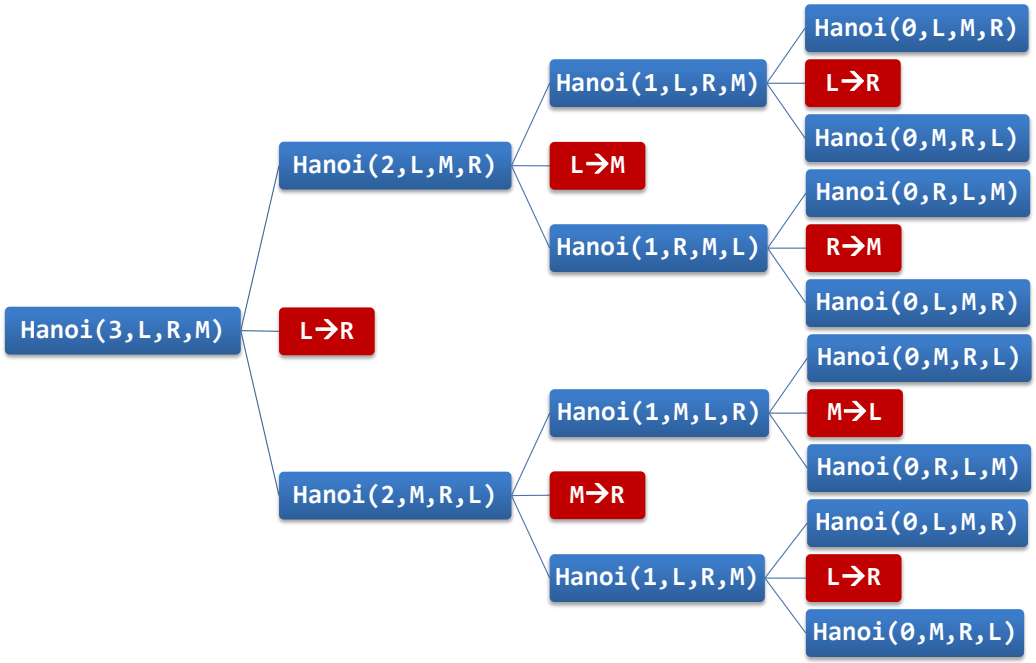
↓

```
Move disk from L to R
Move disk from M to L
Move disk from M to R
Move disk from L to R
Move disk from M to L
Move disk from R to M
Move disk from R to L
Move disk from M to L
Move disk from M to R
Move disk from L to R
Move disk from L to M
Move disk from R to M
Move disk from L to R
Move disk from M to L
Move disk from M to R
Move disk from L to R
```


Tower of Hanoi



Tower of Hanoi



Tower of Hanoi

- How many moves do we need for n disks?

Moves(n) = 1 + 2*Moves(n-1)

n	Moves(n)
1	1
2	3
3	7
4	15
5	31
6	63
n	2^n-1

Tower of Hanoi

- Let us assume that we can move one disk every second.
- How long would it take to move n disks?

n	time
1	1s
5	31s
10	17m 3s
15	9h 6m 7s
20	12d 3h 16m 15s
25	1y 23d 8h 40m 31s
30	> 34y
40	> 34,000y
50	> 35,000,000y
60	> 36,000,000,000y

Digital root

- The *digital root* (or the *repeated digital sum*) of a number is the number obtained by adding all the digits, then adding the digits of that number, and then continuing until a single-digit number is reached.
- For example, the digital root of 65536 is 7, because 6 + 5 + 5 + 3 + 6 = 25 and 2 + 5 = 7.

Digital root

- Basic case: *n* can be represented as a single-digit number → return *n*
- General case: *n* has more than one digit
  - Calculate the sum of the digits
  - Calculate the digital root of the sum

Digital root

```
// Assume we have a function (to be defined)
// that calculates the sum of the digits of a number

// Pre: n ≥ 0
// Returns the sum of the digits of n
// (represented in base 10)
int sumdigits(int n);

// Pre: n ≥ 0
// Returns the digital root of n
int digital_root(int n) {
    if (n < 10) return n;
    else return digital_root(sumdigits(n));
}
```


Write a number *n* in base *b*

- Design a program that writes a number *n* in base *b.*
- Examples:

```
1024 is 10000000000 in base 2
        1101221 in base 3
           2662 in base 7
           1024 in base 10
```


Write a number *n* in base *b*

- Basic case: n = 0 → do not write anything (avoid writing numbers with a leading 0). Treat the zero as a special case outside the function.
- General case: n > 0
  - Write the leading digits of the number (n/b)
  - Write the last digit of the number (n%b)

Write a number *n* in base *b*

```
// Writes the representation of n in base b (n ≥ 0, 2 ≤ b ≤ 10)
// No digits are written when n = 0.
void write_base(int n, int b) {
    if (n > 0) {
        write_base(n/b, b);
        cout << n%b;
    }
}

// Input:  read two numbers, n and b, with n ≥ 0 and 2 ≤ b ≤10
// Output: write the representation of n in base b.
int main() {
    int n, b;
    cin >> n >> b;
    if (n == 0) cout << "0";
    else write_base(n, b);
    cout << endl;
}
```