Recursion

Jordi Cortadella
Department of Computer Science
Recursion
Recursion

• Principle:
  – Reduce a complex problem into a simpler instance of the same problem

• Recursion is a concept tightly related to mathematical induction (typically used for proofs on natural numbers):
  – Proof a base case for the first natural number
  – Inductive step: proof that validity for \( n \) also implies validity for \( n+1 \) (domino effect)
Mathematical induction: example

• The sum of the first $n$ odd numbers is $n^2$:

$$
\sum_{i=1}^{n} (2i - 1) = n^2
$$

• Informal proof:
Mathematical induction: example

\[ \sum_{i=1}^{n} (2i - 1) = n^2 \]

- **Base case**: it works for \( n = 1 \) \( \Rightarrow \) \( 2 \cdot 1 - 1 = 1^2 \)
- **Inductive step**: let us assume it works for \( n \)

\[ \sum_{i=1}^{n+1} (2i - 1) = \sum_{i=1}^{n} (2i - 1) + 2(n + 1) - 1 \]

\[ = n^2 + 2n - 1 \]

\[ = (n + 1)^2 \]
Factorial

Definition of factorial:

\[ n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1 \]

// Pre: n ≥ 0
// Returns n!
int factorial(int n) {
    // iterative solution
    int f = 1;
    for (int i = n; i > 0; --i) f = f * i;
    return f;
}
Factorial

Recursive definition:

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

// Pre: n \geq 0
// Returns n!
int factorial(int n) { // recursive solution
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
Factorial: recursive solution

factorial(n) = n \times factorial(n-1)

factorial(1) = 1

factorial(2) = 2

factorial(n) = n! (n factorial)

Diagram: 
- node for factorial(n)
- recursive calls to factorial(n-1), factorial(n-2), etc.
- base cases factorial(2) = 2 and factorial(1) = 1
- final result n! (n factorial)
Recursion

• 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!
Recursive design: 3 steps

1. Identify the **basic cases** in which a simple non-recursive solution can be provided.

   Example: \(0! = 1! = 1\).

2. **Recursive cases**: solve the complex cases in terms of simpler instances of the same problem (domino effect).

   Example: \(\text{factorial}(n) = n\times\text{factorial}(n-1)\).

3. **Termination**: guarantee that the parameters of the call move closer to the basic cases at each recursive call (the domino chain is not infinite).

   Example: In the case of a factorial, \(n-1\) is closer to 0 than \(n\). Therefore, we can guarantee that this function terminates.
Recursive design: termination

• It is not clear whether the following function terminates:

```c
// 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 $3n+1$ is not closer to 1 than $n$
Recursion: behind the scenes

... 

f = factorial(4);  
...

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

```
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 = factorial(4);

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

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

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

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 of their creation, i.e. the first instance to be created will be the last to be destroyed.
Print a number $n$ in base $b$

- Design a procedure that prints a number $n$ in base $b$ to cout.

- Examples:

  - $1024$ is $100000000000$ in base $2$
  - $1101221$ in base $3$
  - $2662$ in base $7$
  - $1024$ in base $10$
Print a number $n$ in base $b$

- **Basic case:** $n < b$ $\rightarrow$ only one digit. Print it.

- **General case:** $n \geq b$
  - Print the leading digits ($n/b$)
  - Print the last digit ($n\%b$)

```
// Pre: n \geq 0, 2 \leq b \leq 10
// Prints the representation of n in base b
void print_base(int n, int b);
```
Write a number \( n \) in base \( b \)

// Pre: \( n \geq 0 \), \( 2 \leq b \leq 10 \)
// Prints the representation of \( n \) in base \( b \)

```cpp
void print_base(int n, int b) {
    if (n < b) cout << n;
    else {
        print_base(n/b, b);
        cout << n%b;
    }
}
```
void print_base(int n, int b) {
    if (n >= b) print_base(n/b, b);
    cout << n%b;
}
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.

- 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.
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)
// Pre: n is the number of disks (n ≥ 0).
// from, to and aux are the names of the pegs.
// Post: solves the Tower of Hanoi 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);
    }
}
```c
int main() {
    int Ndisks;

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

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

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 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 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 L to M
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 L to M
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
Tower of Hanoi

Hanoi(3,L,R,M)

Hanoi(2,L,M,R)

Hanoi(1,L,R,M)

Hanoi(0,L,M,R)

L → R

Hanoi(0,M,R,L)

Hanoi(1,R,M,L)

Hanoi(0,R,L,M)

R → M

Hanoi(0,L,M,R)

L → R

Hanoi(2,M,R,L)

Hanoi(2,L,M,R)

Hanoi(0,M,R,L)

M → L

Hanoi(0,R,L,M)

M → R

Hanoi(1,L,R,M)

Hanoi(0,L,M,R)
Tower of Hanoi

• How many moves do we need for \( n \) disks?

\[
\text{Moves}(n) = 1 + 2 \times \text{Moves}(n-1)
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{Moves}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>( n )</td>
<td>( 2^{n-1} )</td>
</tr>
</tbody>
</table>
**Tower of Hanoi**

- Let us assume that we can move one disk every second.

- How long would it take to move $n$ disks?

<table>
<thead>
<tr>
<th>n</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1s</td>
</tr>
<tr>
<td>5</td>
<td>31s</td>
</tr>
<tr>
<td>10</td>
<td>17m 3s</td>
</tr>
<tr>
<td>15</td>
<td>9h 6m 7s</td>
</tr>
<tr>
<td>20</td>
<td>12d 3h 16m 15s</td>
</tr>
<tr>
<td>25</td>
<td>1y 23d 8h 40m 31s</td>
</tr>
<tr>
<td>30</td>
<td>&gt; 34y</td>
</tr>
<tr>
<td>40</td>
<td>&gt; 34,000y</td>
</tr>
<tr>
<td>50</td>
<td>&gt; 35,000,000y</td>
</tr>
<tr>
<td>60</td>
<td>&gt; 36,000,000,000y</td>
</tr>
</tbody>
</table>