Estruturas de Informação

Recursion

Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

Recursion pattern

A programming technique in which a function calls itself

Recursion is equivalent of **mathematical induction**, which is a way of defining something in terms of itself

Example: exponentiation - y raised to the n power

$$y^n = \begin{cases} 1 & n = 0 \\ y \times y^{n-1} & n > 0 \end{cases}$$

The power of recursion is the possibility of defining elements based on ***simpler versions of themselves***

Iteration

- Problems that require repetition are solved using iteration i.e., some type of loop

Example: calculating the y power of n (**n positive**)

$$y \times y \times \cdots \times y$$

Iterative solution:

```
public int power(int y, int n) {
    int p=1;
    for (int i = 0; i < n; i++)
        p = p*y;
    return p;
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using recursion

Example: calculating the y power of n, where n is positive

$$y^n = \begin{cases} 1 & n = 0 \\ y \times y^{n-1} & n > 0 \end{cases}$$

Recursive solution:

```
public int power(int y, int n) {
   if (n==0)
        return 1;

   return y * power(y, n-1);
}
```

Recursive problem-solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind
- We keep doing this until we reach a problem that is simple enough to be solved directly
- This simplest problem is known as ***the base case***

```
public int power(int y, int n) {
    if (n==0) return 1;    //base case

    return y * power(y, n-1);
}
```

- **The base case stops the recursion**, because it doesn't make another call to the method

Recursive problem-solving

- If the base case hasn't been reached, the recursive case is executed

```
public int power(int y, int n) {
    if (n==0)
        return 1;

    return y * power(y, n-1);
}
```

The recursive case:

- reduces the overall problem to one or more simpler problem of the same kind
- makes recursive calls to solve the simpler problems

Recursive design

Recursive methods/functions **require**:

1. One or more (non-recursive) **base cases** that will cause the recursion to end
2. One or more **recursive cases** that operate on smaller problems **and** get you closer to the base case

```
recursiveMethod (parameters) {
    if (stopping condition) {
       // handle the base case
    }
    // recursive case:
    // possibly do something here
       recursiveMethod(modified parameters);
    // possibly do something here
    }
}
```

Broken recursive power

```
public int brokenpower(int y, int n) {
     return y * brokenpower(y, n-1);
     if (n==0)
       return 1;
}
```

What’s wrong here?

Note: The base case(s) should always be checked before the recursive call

Why do recursive methods work?

Activation Records on the **Run-time Stack** are the key:

- Each time you call a function (any function) you get a new activation record
- Each activation record contains a copy of all local variables and parameters for that invocation
- The activation record remains on the stack until the function returns, then it is destroyed

Tracing a recursive method

```
public int power(int y, int n) {
    if (n==0)
       return 1;

    return y * power(y, n-1); }
}
```

What happens when execute: `power(2,3)`

power(2, 3) -> 8

return 2 * power(2, 2) -> 2*4

return 2 * power(2, 1) -> 2*2

return 2 * power(2, 0); -> 2*1

Backtracking – the power of recursion

In recursive problems that involve ***backtracking***, the steps towards to the problem solution are tested and stored, but if some steps do not lead to a final solution, ***these are broken, that is, we turn back up to the most recent position, and try new possibilities***

The general steps for any problem that involves recursive backtracking, assuming that the number of potential candidates in each step is finite, are:

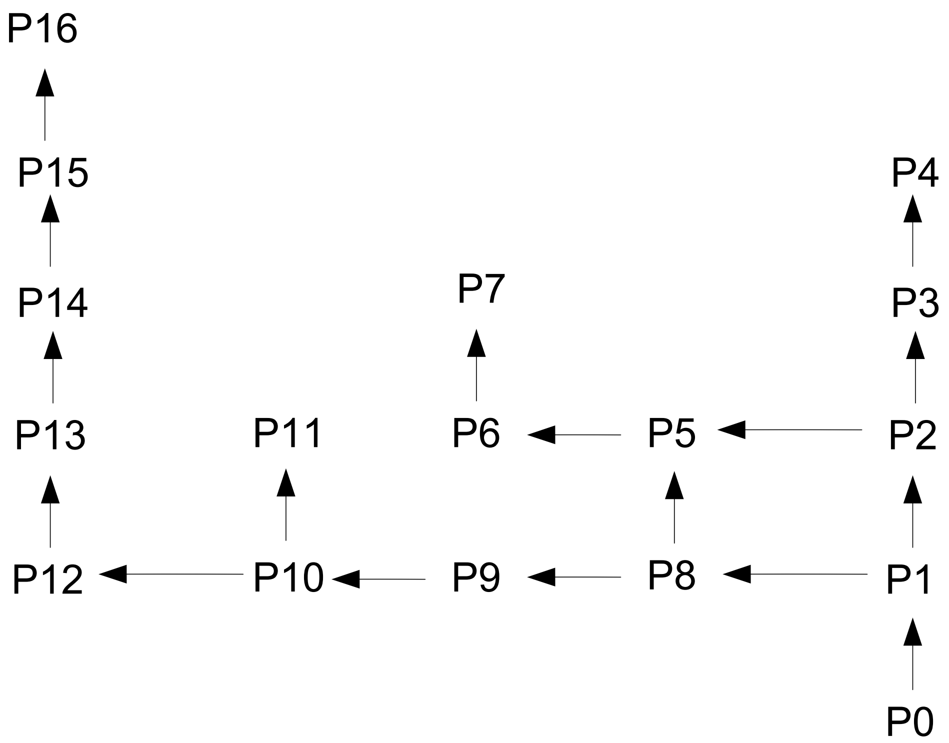
Initialize candidates
repeat
  select next
  If acceptable save
  If incomplete solution try next step
  If fails cancel
until success or no longer exists candidates

Backtracking – the power of recursion

Find a path from the position $\mathbf{P_0} \rightarrow \mathbf{P_{16}}$

Movement directions: North, West

Without repeat Positions



Fibonacci Sequence - binary recursion

Fibonacci numbers are defined recursively: 0, 1, 1, 2, 3, 5, 8, ...

$F_0 = 0,$

$F_1 = 1,$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$

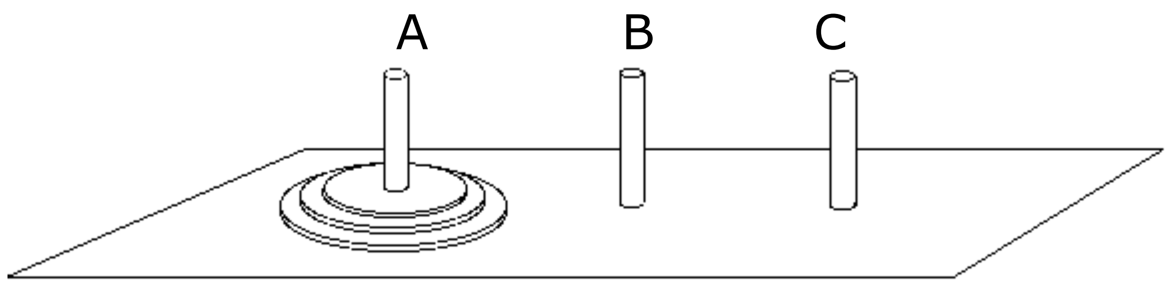
```
public int Fibonacci (int n){
  if (n <= 1)
     return n;

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Tower of Hanoi puzzle

The **Tower of Hanoi** is a game or puzzle that consists of three rods, and N disks of different sizes which can slide onto any rod

The puzzle starts with the N disks in ascending order of size on one rod, the smallest at the top, thus making a conical shape



The objective of the puzzle is to move the entire stack of disks to another rod, obeying the following simple rules:

- Only one disk can be moved at a time
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack
- No disk may be placed on top of a smaller disk

Consult: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Multiple recursion

- The solution to this problem is trivial if the number of discs is 1
- If we have N disks in Tower A the solution is to reduce the complexity of the problem to the situation where we have only one disk and for which we know the solution

```
public void towersHanoi(int n, char A, char B, char C) {
    if (n == 1)
      System.out.println("Disk 1 from "+ A + " to " + C);
    else{
        towersHanoi(n-1, A, C, B);
        System.out.println("Disk " + n + " from " + A + " to " + C);
        towersHanoi(n-1, B, A, C);
    }
}
```

Iterative Solution:

https://marcin-chwedczuk.github.io/iterative-solution-to-towers-of-hanoi-problem

Recursion vs. Iteration

- Any recursive algorithm can be re-written as an iterative algorithm (loops). This is especially true for methods in which:
  - there is only one recursive call
  - it comes at the end of the method – tail recursion

- **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

Recursion vs. iteration

Rule of thumb:

- If it's easier to formulate a solution recursively, use recursion, example: Hanoi Tower puzzle
- If the cost of recursion is too high use iteration, example Fibonacci sequence
- If the data structure is itself recursive, example: trees, graphs - recursion is the natural way to handle them

Exercise: Tracing a Recursive Method

```
public static void mystery(int i) {
       if (i <= 0) {                    // base case
          return;
       }
       System.out.println(i);      //recursive case
       mystery(i-1);
       System.out.println(i);
}
```

Exercise: trace execution for mystery(2)

```
mystery(2)
       System.out.println(2)                > 2
       mystery(1)

             System.out.println(1)          > 1
             mystery(0)

                   System.out.println(1)    > 1
                   System.out.println(2)    > 2
```

mistery(1)
mistery(2)