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: printing integers from n_1 to n_2, where $n_1 <= n_2$

Iterative solution:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Recursion

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

Example: printing integers from n_1 to n_2, where $n_1 <= n_2$

Recursive solution:

```
public static void printSeries(int n1, int n2) {
    if (n1==n2){
       System.out.println(n2);  }
    else {
       System.out.print(n1 + ", ");
       printSeries(n1+1, n2); }
}
```

Tracing a recursive method

```
public static void printSeries(int n1, int n2) {
    if (n1==n2) {
       System.out.println(n2);  }
     else {
        System.out.print(n1 + ", ");
        printSeries(n1+1, n2);   }
}
```

What happens when execute: printSeries(5,7)

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 static void printSeries(int n1, int n2) {
    if (n1==n2) {
        System.out.println(n2);  }  //base case
    else {
        System.out.print(n1 + ", ");
        printSeries(n1+1, n2);   }
}
```

- 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 static void printSeries(int n1, int n2) {
    if (n1==n2) {
          System.out.println(n2);  }
     else {
        System.out.print(n1 + ", ");
        printSeries(n1+1, n2);   }
}
```

The recursive case:

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

Factorial

n! = 1×2×3×…×(n-1)×n

Recursive definition:

$$factorial(n) = \begin{cases} 1 & if\ n = 0 \\ n \times factorial(n-1) & else \end{cases}$$

As a Java method:

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );    // argument must be nonnegative
  else if (n == 0)
    return 1;                                 // base case
  else
    return n * factorial(n−1);                // recursive case
}
```

Exercise: trace execution (show method calls) for n=5

Broken recursive factorial

```
public static int brokenFactorial(int n){
   int x = brokenFactorial(n-1);
   if (n == 1)
        return 1;
   else
        return n * x;
}
```

What's wrong here?

Trace calls "by hand"

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

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

Structure of a recursive method

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

     // possibly do something her
    }
}
```

- There can be multiple base cases and recursive cases
- When we make the recursive call, we typically use parameters that bring us closer to a base case

Rules for recursive algorithms

Base case - must have a way to end the recursion

Recursive call - must ***change*** at least one of the parameters ***and*** make progress towards the base case

Power function, power (y, n) = y^n

$$power(y,n) = \begin{cases} 1 & if\ n = 0 \\ y \times power(y, n-1) & else \end{cases}$$

base case

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

Linear recursion

Algorithm: linearSum(A, n)

Input: Array A, of integers

Integer n such that $0 \le n \le |A|$

Output: Sum of the first n integers in A

Recursive definition:

$$linearSum(A,n) = \begin{cases} 0 & if\ n = 0 \\ A[n-1] + linearSum(A, n-1) & else \end{cases}$$

Exercise: trace execution (show method calls) for linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]

Tail recursion

Tail recursion occurs when a linearly recursive method makes its recursive call as its last step

Algorithm: reverseArray(A, i, j)

Input: Array A of integers
nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

```
if i <  j then
      Swap A[i] and A[ j]
      reverseArray(A, i + 1,  j – 1)
return
```

Defining arguments for recursion

In creating recursive methods, it is important to define the methods in ways that facilitate recursion

This sometimes requires to define additional parameters that are passed to the method

```
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                          // if at least two elements in subarray
    int temp = data[low];                    // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1);   // recur on the rest
  }
}
```

Binary recursion

Binary recursion occurs whenever there are **two** recursive calls for each non-base case

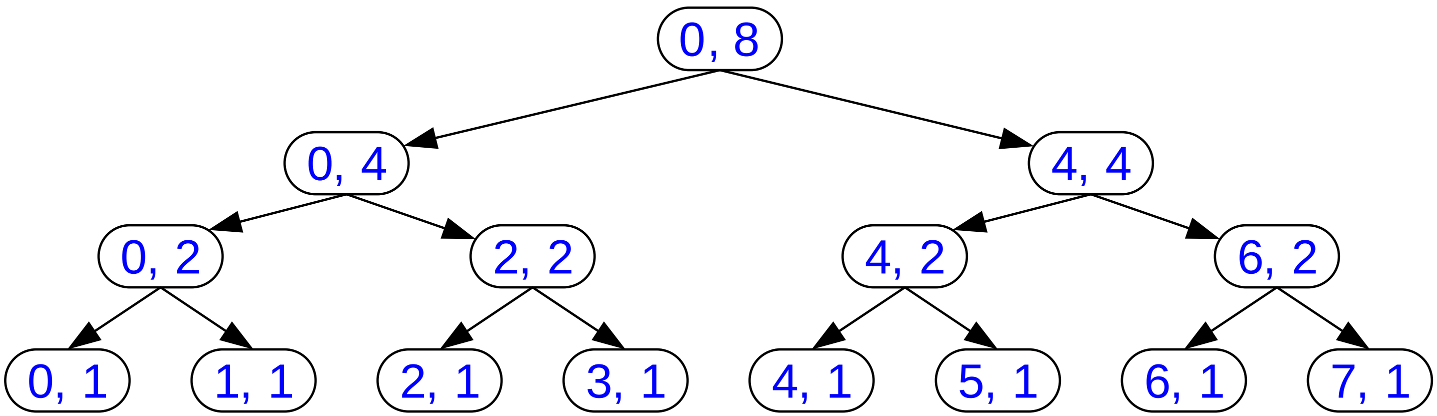
Problem: add all the numbers in an integer array A:

Algorithm: BinarySum(A, i, n)
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i

```
if n = 1 then
    return A[i]
return BinarySum(A, i, n/2) + BinarySum(A, i + n/2, n/2)
```

Binary Recursion

Example trace: BinarySum(A, i, n)



Fibonacci Sequence

Fibonacci numbers are defined recursively:

$$F_0 = 0$$

$$F_1 = 1$$

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

Algorithm: BinaryFib(k):

Input: Nonnegative integer k

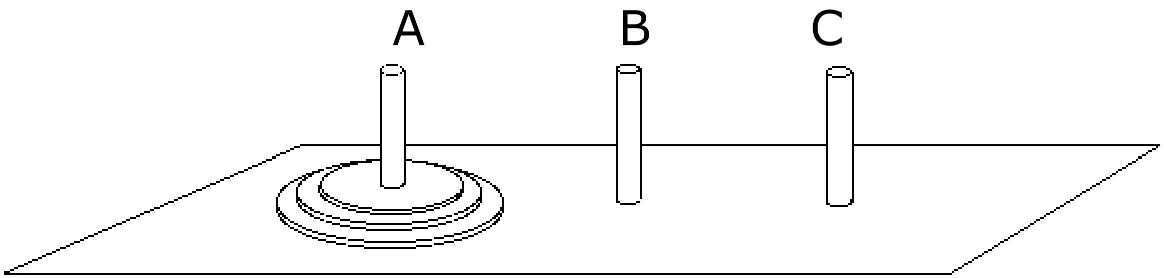
Output: The kth Fibonacci number F_k

```
if k < 1 then
    return k
else
    return BinaryFib(k - 1) + BinaryFib(k - 2)
```

Multiple recursion

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

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

```
Hanoi-Tower (N, TowerA, TowerB, TowerC)
 if  (n = 1)
    Move disk TowerA → TowerB
 else
         Hanoi-Tower (n-1, TowerA, TowerC, TowerB)
         Hanoi-Tower (1, TowerA, TowerB, TowerC)
         Hanoi-Tower (n-1, TowerC, TowerB, TowerA)
```

Consult: http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/13/hanoi.html

Backtracking

In recursive problems that involve ***backtraking***, 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 backtraking, assuming that the number of potential candidates in each step is finite, are:

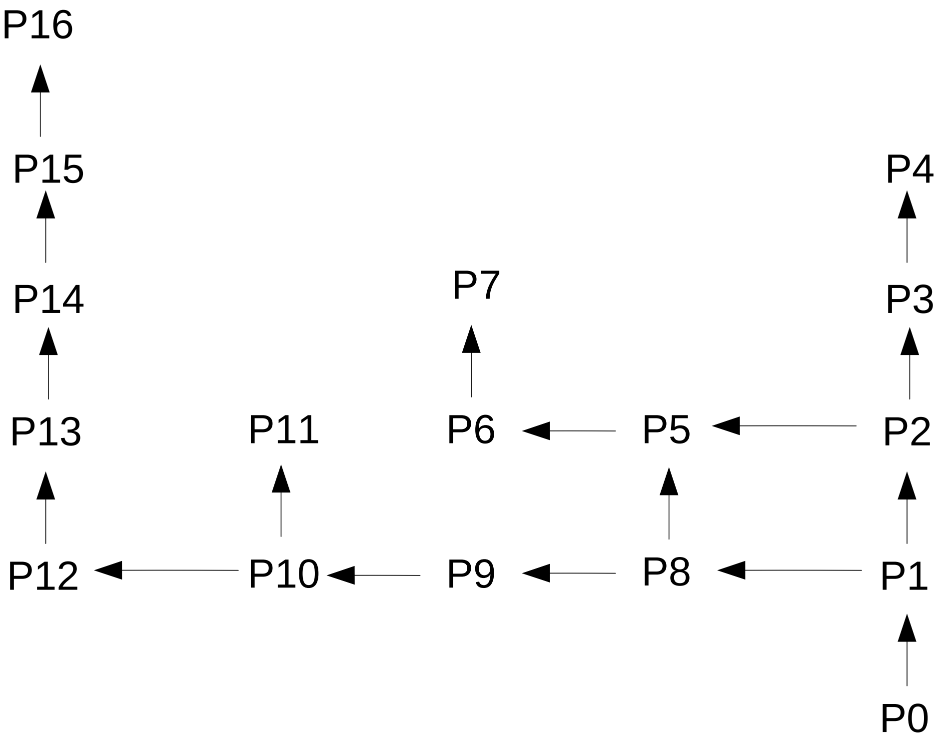
> Initialize candidates
> ***repeat***
>   select next
>   If acceptable save
>   If incomplete solution try next step
>   If fails cancel
> ***until*** success or no longer exists candidates

Backtracking

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

Movement directions: North, West

Without repeat Positions



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
- If the data structure is itself recursive, example: trees, graphs, recursion are the natural way to handle them
- If the cost of recursion is too high use iteration, example: Fibonacci sequence