POLITECNICO
DI TORINO
e-Lite

Recursion

Tecniche di Programmazione – A.A. 2015/2016

Summary

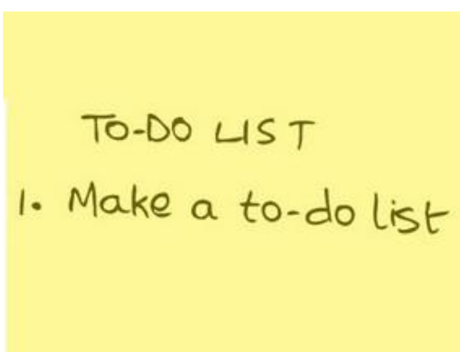
1. Definition and divide-and-conquer strategies
2. Simple recursive algorithms
   1. Fibonacci numbers
   2. Dicothomic search
   3. X-Expansion
   4. Proposed exercises
3. Recursive vs Iterative strategies
4. More complex examples of recursive algorithms
   1. Knight's Tour
   2. Proposed exercises

Definition and divide-and-conquer strategies

Recursion

Definition

- A method (or a procedure or a function) is defined as recursive when:
  - Inside its definition, we have a call to the same method (procedure, function)
  - Or, inside its definition, there is a call to another method that, directly or indirectly, calls the method itself
- An algorithm is said to be recursive when it is based on recursive methods (procedures, functions)



Example: Factorial

$$\begin{cases} 0! \stackrel{\text{def}}{=} 1 \\ \forall N \geq 1: \\ N! \stackrel{\text{def}}{=} N \times (N-1)! \end{cases}$$

```
public long recursiveFactorial(long N)
{
  long result = 1 ;

  if ( N == 0 )
    return 1 ;
  else {
    result = recursiveFactorial(N-1) ;
    result = N * result ;
    return result ;
  }
}
```

Motivation

- **Many problems lend themselves, naturally, to a recursive description:**
  - We define a method to solve sub-problems similar to the initial one, but smaller
  - We define a method to combine the partial solutions into the overall solution of the original problem

Divide et impera

Gaius Julius Caesar

Divide et Impera – Divide and Conquer

- Solution = Solve (Problem) ;

- Solve (Problem) {
  - List<SubProblem> subProblems = **Divide** (Problem) ;
  - For (each subP[i] in subProblems) {
    - SubSolution[i] = **Solve** (subP[i]) ;
  - }
  - Solution = **Combine** (SubSolution[1..N]) ;
  - return Solution ;
- }

Divide et Impera – Divide and Conquer

- Solution = Solve (Problem) ;

- Solve (Problem) {
  - List<SubProblem> subProblems = **Divide** (Problem) ;
  - For (each subP[i] in subProblems) {
    - SubSolution[i] = **Solve** (subP[i]) ;
  - }
  - Solution = **Combine** (SubSolution[1..N]) ;
  - return Solution ;
- }

"a" sub-problems, each "b" times smaller than the initial problem

recursive call

How to stop recursion?

- Recursion **must not** be infinite
  - Any algorithm must always terminate!
- After a sufficient nesting level, sub-problems become so small (and so easy) to be solved:
  - Trivially (ex: sets of just one element)
  - Or, with methods different from recursion

Warnings

- Always remember the "termination condition"
- Ensure that all sub-problems are strictly "smaller" than the initial problem

Divide et Impera – Divide and Conquer

- Solve (Problem) {
  - if(problem is trivial)
    - Solution = **Solve_trivial** (Problem) ;
  - else {
    - List<SubProblem> subProblems = **Divide** (Problem) ;
    - For (each subP[i] in subProblems) {
      - SubSolution[i] = **Solve** (subP[i]) ;
    - }
    - Solution = **Combine** (SubSolution[1..N]) ;
  - }
  - return Solution ;
- }

do recursion

What about complexity?

- a = number of sub-problems for a problem
- b = how smaller sub-problems are than the original one
- n = size of the original problem
- T(n) = complexity of **Solve**
  - …our unknown complexity function
- **Θ(1)** = complexity of **Solve_trivial**
  - …otherwise it wouldn't be trivial
- D(n) = complexity of **Divide**
- C(n) = complexity of **Combine**

Divide et Impera – Divide and Conquer

- Solve (Problem) { ← T(n)
  - if(problem is trivial)
    - Solution = **Solve_trivial** (Problem) ; ← $\Theta(1)$
  - else {
    - List<SubProblem> subProblems = **Divide** (Problem) ; ← D(n)
    - For (each subP[i] in subProblems) { ← a times
      - SubSolution[i] = **Solve** (subP[i]) ; ← T(n/b)
    - }
    - Solution = **Combine** (SubSolution[1..a]) ; ← C(n)
  - }
  - return Solution ;
- }

Complexity computation

- $T(n) =$
  - $\Theta(1)$ for $n \leq c$
  - $D(n) + a\,T(n/b) + C(n)$ for $n > c$

- Recurrence Equation not easy to solve in the general case
- Special case:
  - If $D(n)+C(n)=\Theta(n)$
  - We obtain **$T(n) = \Theta(n \log n)$**.

Simple recursive algorithms

Recursion

Fibonacci Numbers

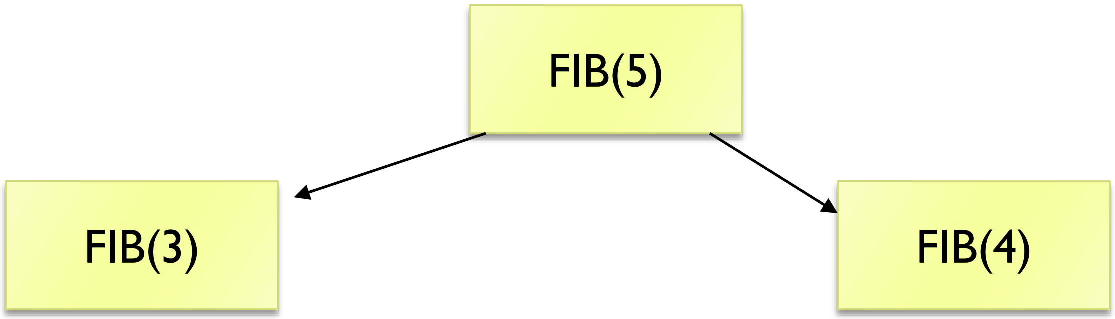
- Problem:
  - Compute the N-th Fibonacci Number
- Definition:
  - $FIB_{N+1} = FIB_N + FIB_{N-1}$ for $N>0$
  - $FIB_1 = 1$
  - $FIB_0 = 0$

Recursive solution

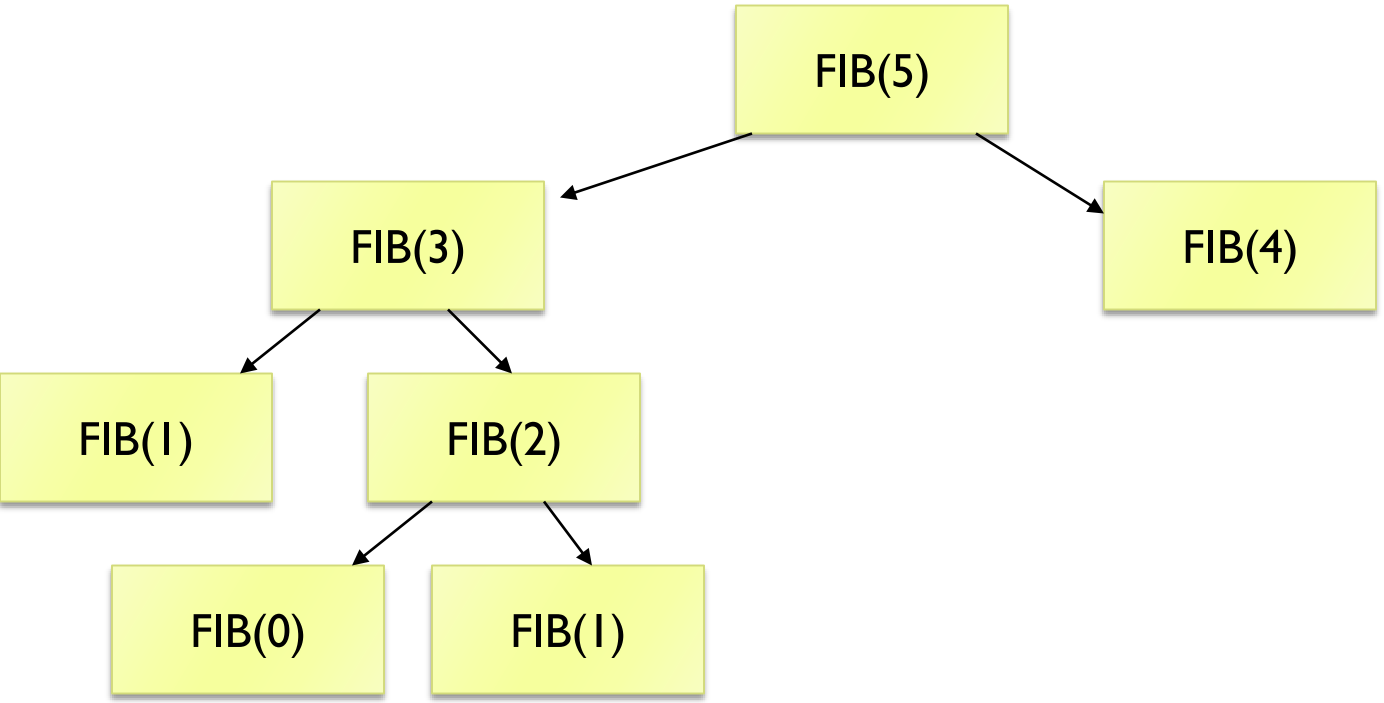
```
public long recursiveFibonacci(long N) {
    if(N==0)
        return 0 ;
    if(N==1)
        return 1 ;

    long left = recursiveFibonacci(N-1) ;
    long right = recursiveFibonacci(N-2) ;

    return left + right ;
}
```

```
Fib(0)  = 0
Fib(1)  = 1
Fib(2)  = 1
Fib(3)  = 2
Fib(4)  = 3
Fib(5)  = 5
```

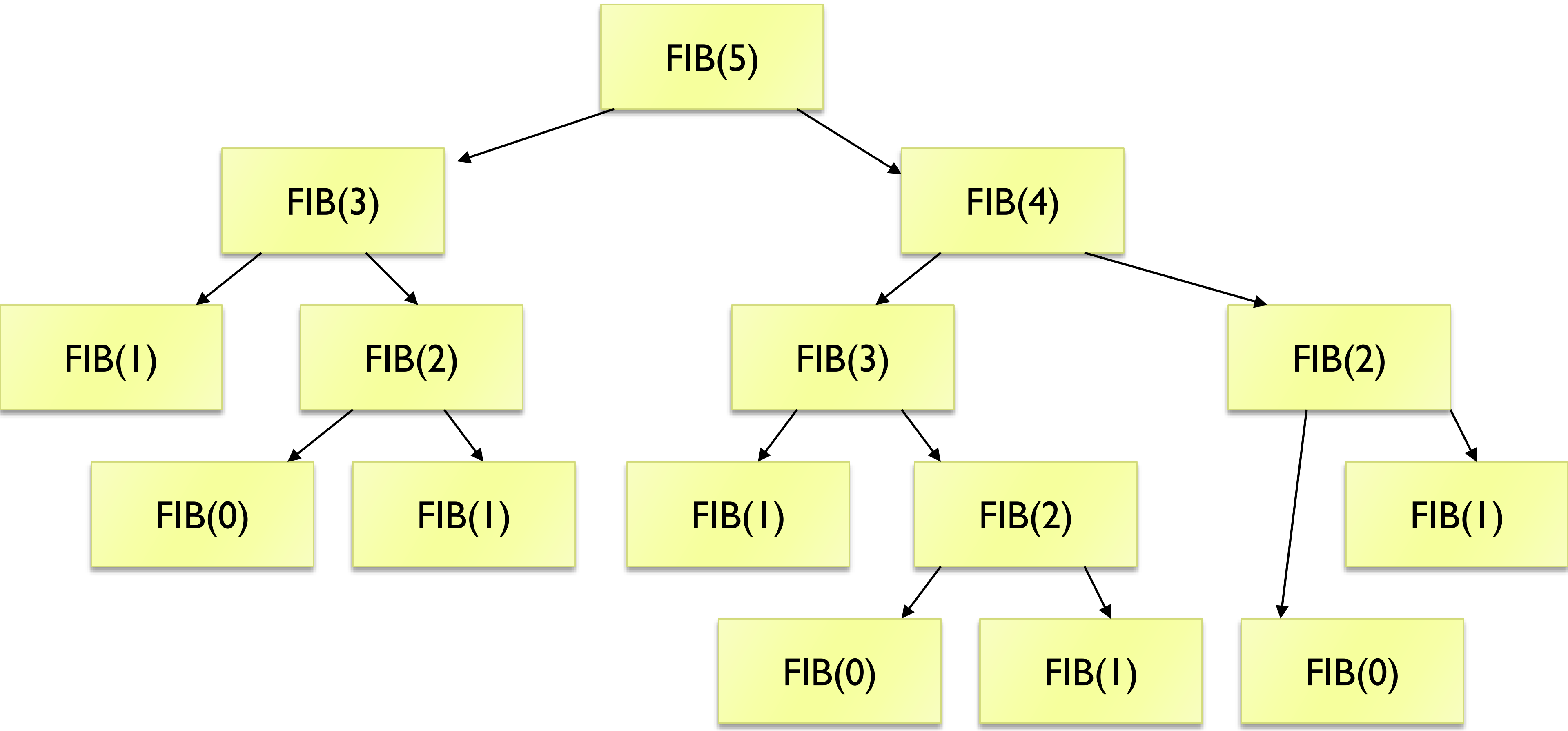
Analysis



Analysis



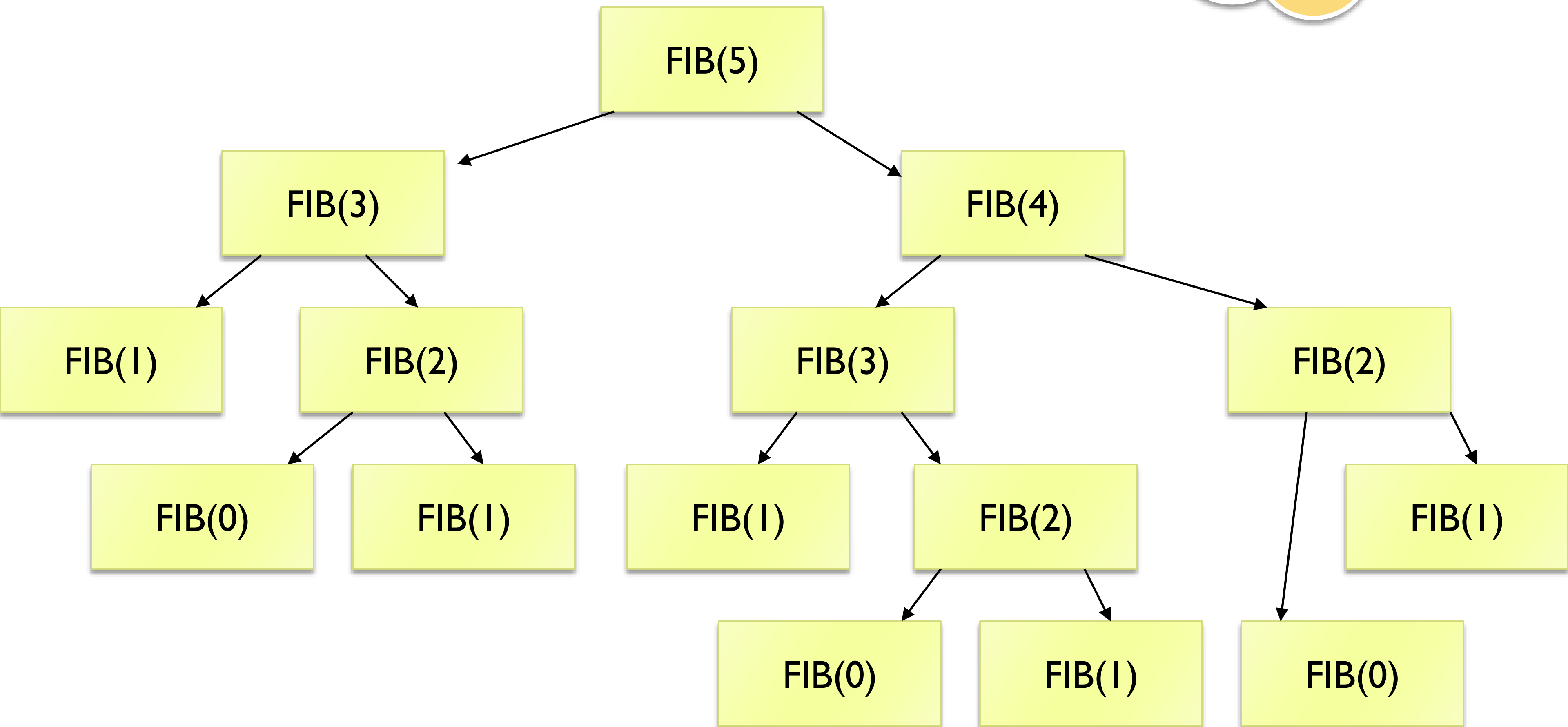
Analysis



FIB(5)
FIB(3)
FIB(4)
FIB(1)
FIB(2)
FIB(3)
FIB(2)
FIB(0)
FIB(1)
FIB(1)
FIB(2)
FIB(1)
FIB(0)
FIB(1)
FIB(0)

Analysis

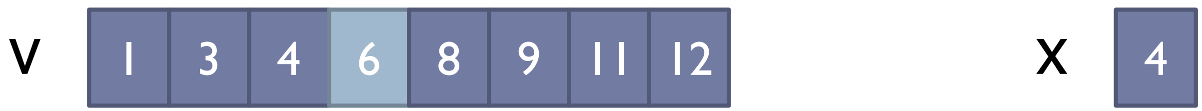
Complexity?



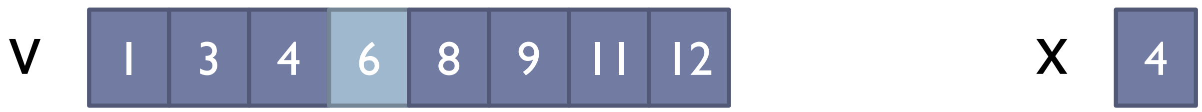
Example: dichotomic search

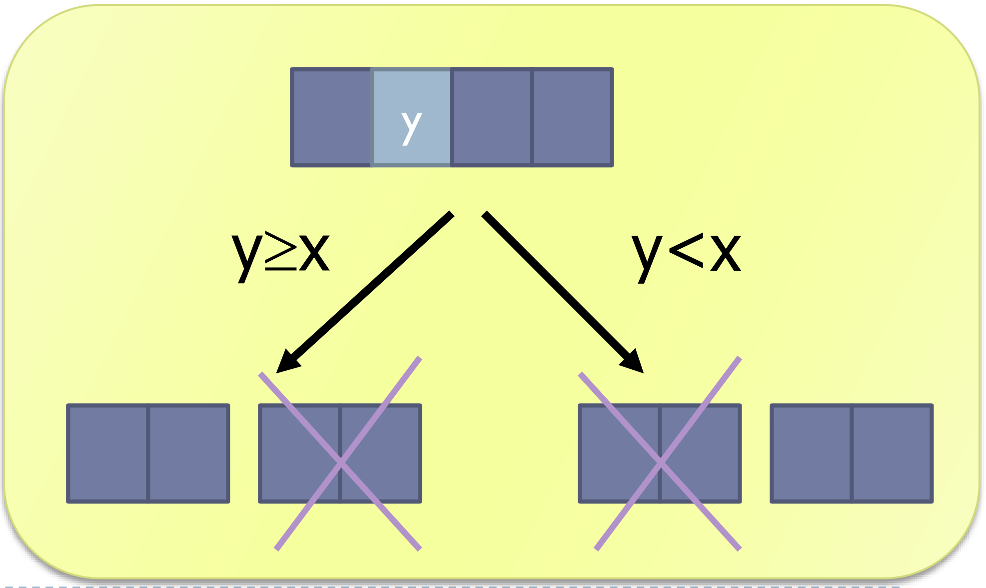
- Problem
  - Determine whether an element x is **present** inside an ordered **vector** v[N]
- Approach
  - Divide the vector in two halves
  - Compare the middle element with x
  - Reapply the problem over one of the two halves (left or right, depending on the comparison result)
  - The other half may be ignored, since the vector is ordered

Example

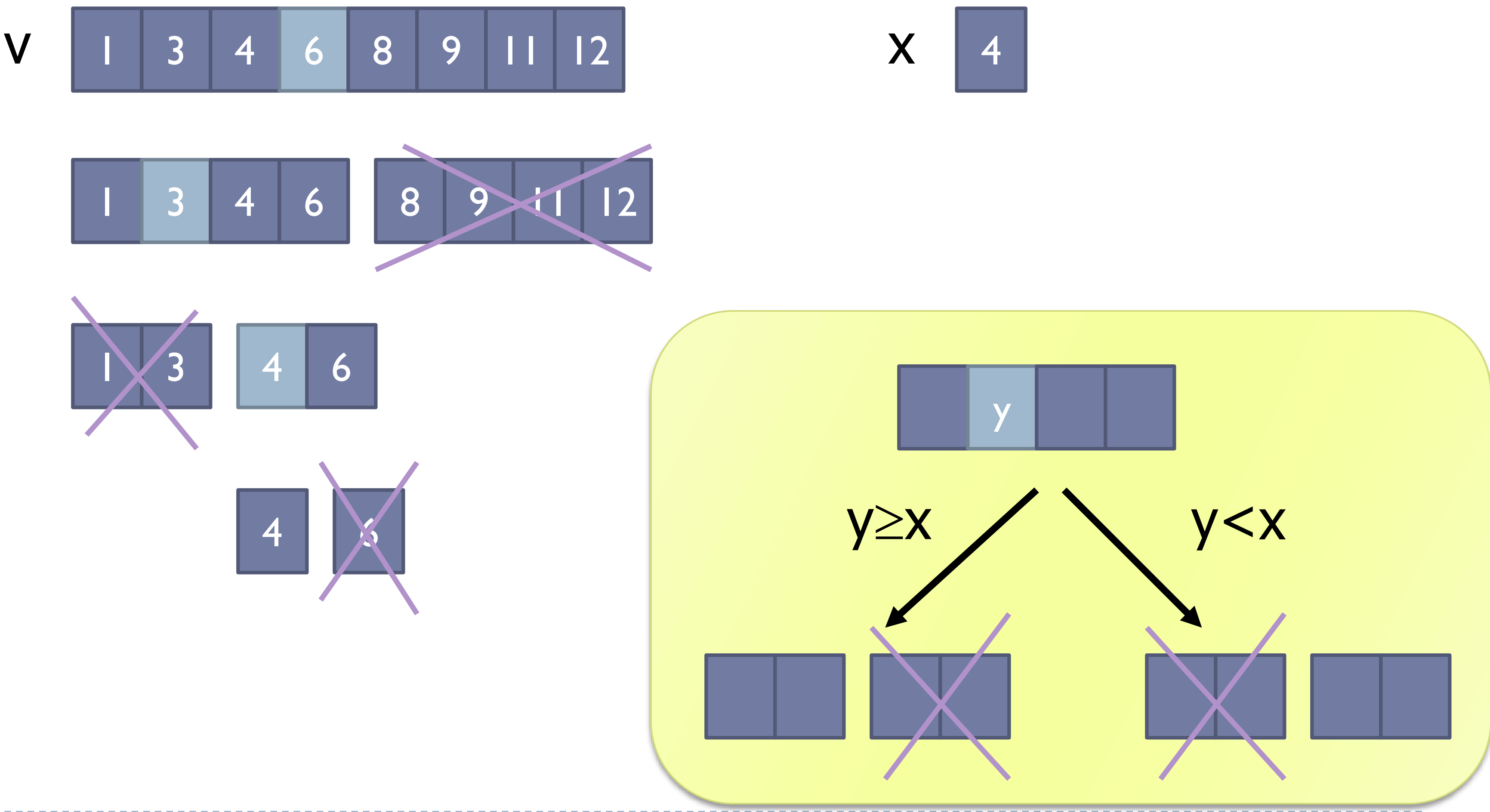


Example





Example



Solution

```
public int find(int[] v, int a, int b, int x)
{
	if(b-a == 0) { // trivial case
		if(v[a]==x) return a ; // found
		else return -1 ;	// not found
	}

	int c = (a+b) / 2 ; // splitting point
	if(v[c] >= x)
		return find(v, a, c, x) ;
	else return find(v, c+1, b, x) ;
}
```

Solution

```
public int fir
{
        if(b-a

        }

        int c = (a+b) / 2         itting point
        if(v[c] >= x)
                return find(v, a, c, x) ;
        else return find(v, c+1, b, x) ;
}
```

Beware of integer-arithmetic approximations!

Quick reference

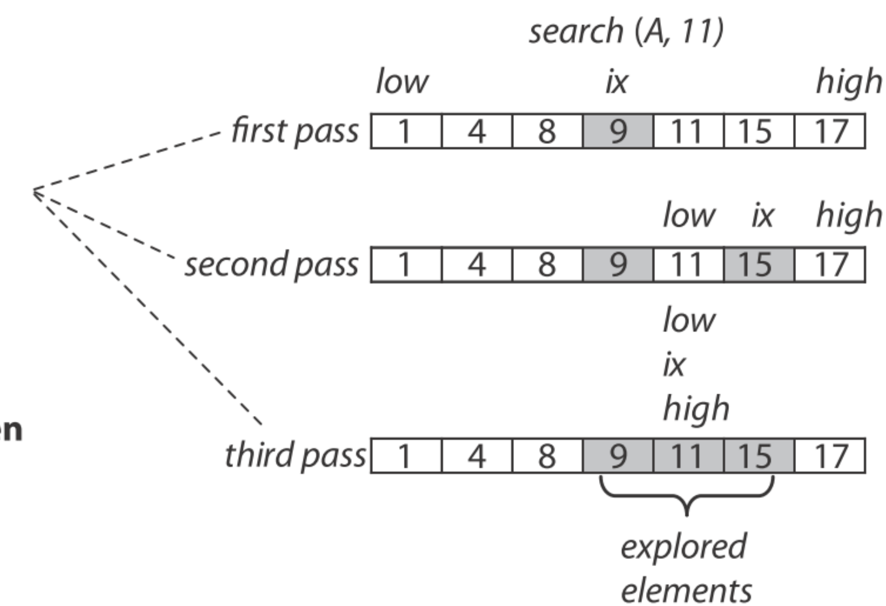
BINARY SEARCH

Best	Average	Worst
O (1)	O (log n)	O (log n)

Array

Divide and Conquer

```
search (A, t)
    low = 0
    high = n −1
    while (low ≤ high) do
      ix = (low + high)/2
      if (t = A[ix]) then
          return true
      else if (t < A[ix]) then
          high = ix − 1
      else low = ix + 1
    return false
end
```



Exercise: Value X

- When working with Boolean functions, we often use the symbol X, meaning that a given variable may have indifferently the value `0` or `1`.
- Example: in the OR function, the result is `1` when the inputs are `01`, `10` or `11`. More compactly, if the inputs are `X1` or `1X`.

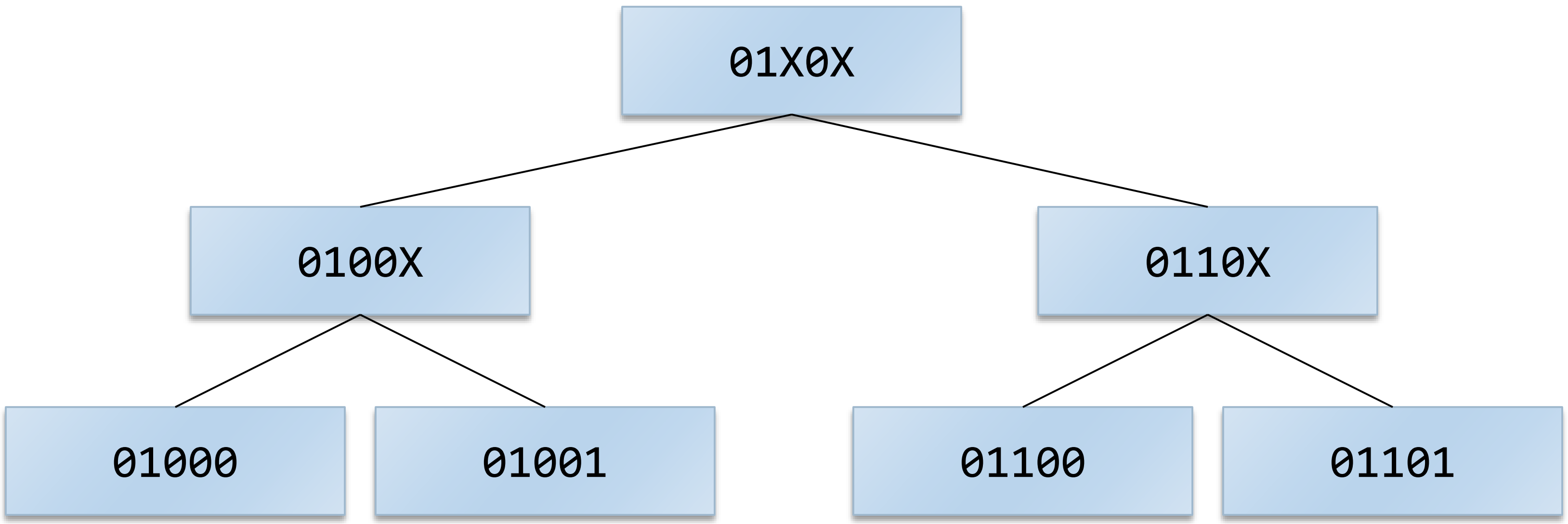
X-Expansion

- We want to devise an algorithm that, given a binary string that includes characters `0`, `1` and X, will compute all the possible combinations implied by the given string.
- Example: given the string `01X0X`, algorithm must compute the following combinations
  - `01000`
  - `01001`
  - `01100`
  - `01101`

Solution

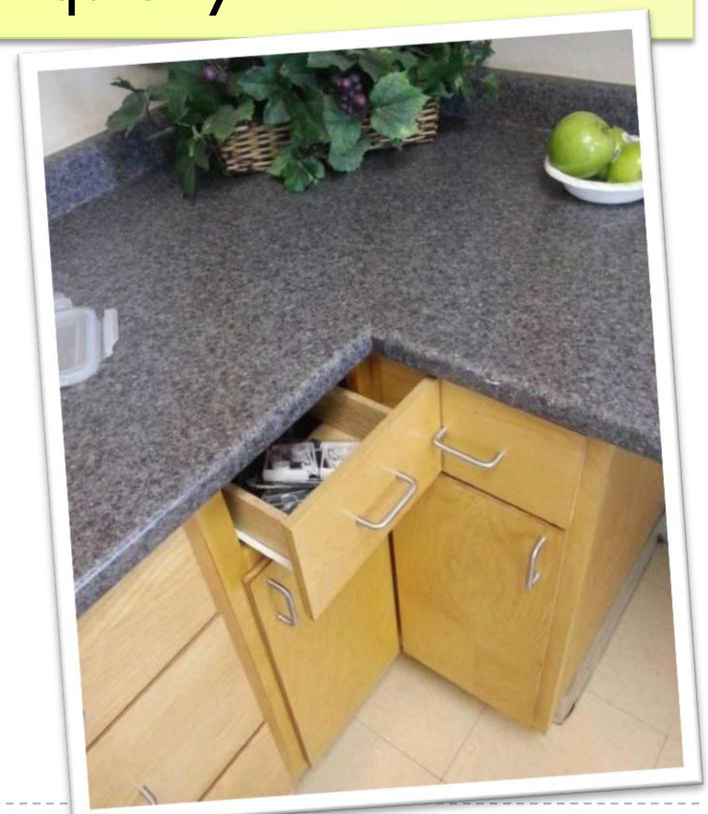
- We may devise a recursive algorithm that explores the complete 'tree' of possible compatible combinations:
  - Transforming each X into a `0`, and then into a `1`
  - For each transformation, we recursively seek other X in the string
- The number of final combinations (leaves of the tree) is equal to 2^N, if N is the number of X.
- The tree height is N+1.

Combinations tree



Recursion myths

- Recursive algorithms are O(n log n)
- Recursive algorithms are better than non-recursive ones
- Recursive algorithms can be coded quickly



Why recursion?

- Divide et impera
- Systematic exploration/enumeration
- Handling recursive data structures

Licenza d'uso