POLITECNICO DI TORINO

e-Lite

Recursion

Tecniche di Programmazione – A.A. 2018/2019

Summary

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

Definition and divide-and-conquer strategies

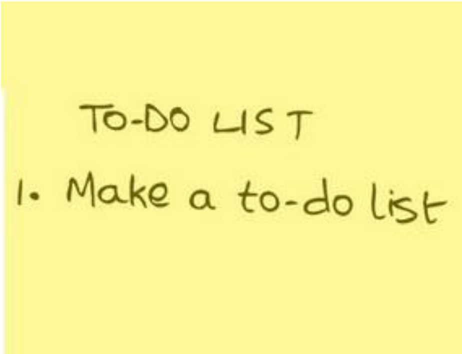
Recursion

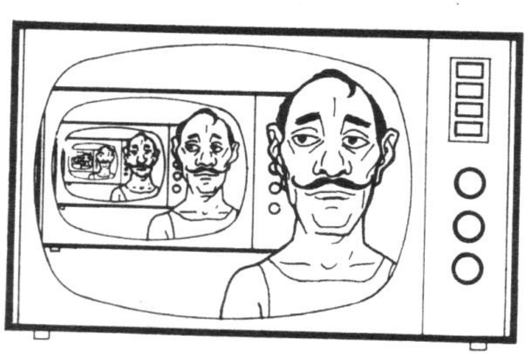
Why recursion?

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

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)






Per capire la
ricorsione prima
devi capire la
ricorsione
@vitadainformatici

Example: Factorial

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

```java
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

Gaius Julius Caesar

Recursion

- **Divide et Impera**

- Split a problem $\mathcal{P}$ into $\{\mathcal{Q}_i\}$ where $\mathcal{Q}_i$ are still complex, yet *simpler* instances of the same problem.
- Solve $\{\mathcal{Q}_i\}$, then merge the solutions
- Merge & split must be "simple"
- A.k.a., *Divide 'n Conquer*

- **Exploration**

- Systematic procedure to enumerate all possible solutions
- Solutions (built stepwise)
  - Paths
  - Permutations
  - Combinations
- Divide et Impera, by "dividing" the possible solutions

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) ; ← Θ(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 **$T(n) = \Theta(n \log n)$**.

Exploration

- **Explore (S) {**
  - List<Step> steps = **PossibleSteps** (Problem, **S**) ;
  - for (each **p** in steps) {
    - **S**.**Do** (**p**)
    - **Explore** (**S**) ;
    - **S**.**Undo** (**p**) ;
  - }
- }

Exploration

- Explore (**S**) {
  - List<Step> steps = **PossibleSteps** (Problem, **S**) ;
  - for (each **p** in steps) {
    - **S**.**Do** (**p**)
    - **Explore** (**S**) ;
    - **S**.**Undo** (**p**) ;
  - }
- }

The "status" of the problem

Local variable

"Try" the step

Recursion

Backtrack!

Design tips

Recursion

Goal

1. Analysis of a problem to be solved with recursive techniques
2. Identification of the main design choices
3. Identification of the main implementation strategies

Analizzare il problema

- Come imposto in generale la ricorsione?
- Che cosa mi rappresenta il "livello"?
- Com'è fatta una soluzione parziale?
- Com'è fatta una soluzione totale?

Generale le possibili soluzioni

- Qual è la regola per generare tutte le soluzioni del livello+1 a partire da una soluzione parziale del livello corrente?
- Come faccio a riconoscere se una soluzione parziale è anche completa? (terminazione con successo)
- Come viene avviata la ricorsione (livello 0)?

Identificare le soluzioni valide

- Data una soluzione **parziale**, come faccio a
  - sapere se è valida (e quindi continuare)?
  - sapere se non è valida (e quindi terminare la ricorsione)?
  - nb. magari non posso…
- Data una soluzione **completa**, come faccio a
  - sapere se è valida?
  - sapere se non è valida?
- Cosa devo fare con le soluzioni complete valide?
  - Fermarmi alla prima?
  - Generarle e memorizzarle tutte?
  - Contarle?

Progettare le strutture dati

- Qual è la struttura dati per memorizzare una soluzione (parziale o completa)?
- Qual è la struttura dati per memorizzare lo stato della ricerca (della ricorsione)?

Scheletro del codice

```
// Struttura di un algoritmo ricorsivo generico

void recursive (... , level) {

  // E -- sequenza di istruzioni che vengono eseguite sempre
  // Da usare solo in casi rari (es. Ruzzle)
  doAlways();

  // A
  if (condizione di terminazione) {
    doSomething;
    return;
  }

  // Potrebbe essere anche un while ()
  for () {

    // B
    generaNuovaSoluzioneParziale;


    if (filtro) { // C
      recursive (..., level + 1);
    }

    // D
    backtracking;
  }

}
```

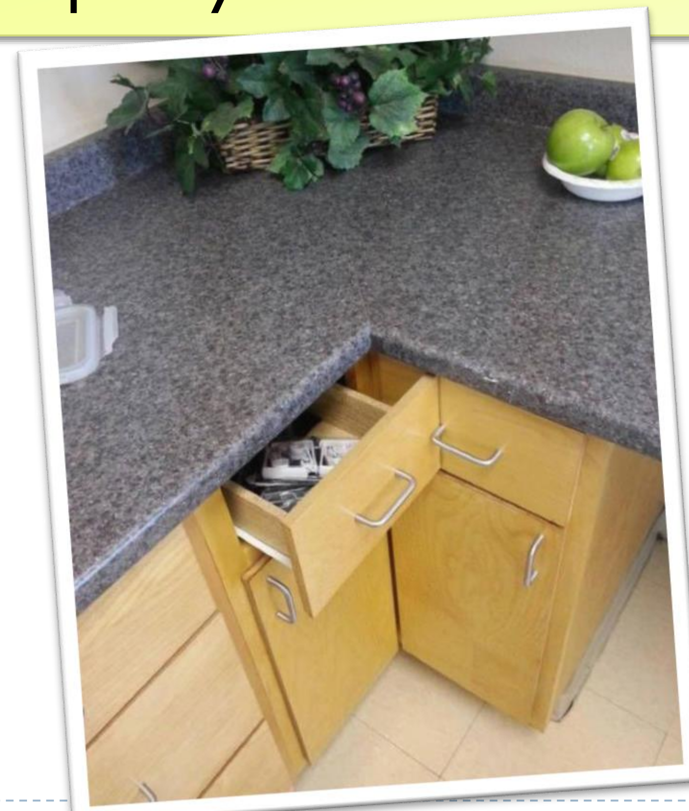
Riempire lo scheletro (del codice)

Blocco	Frammento di codice
A	
B	
C	
D	
E	

```
// Struttura di un algoritmo ricorsivo

void recursive (... , level) {

  // E -- sequenza di istruzioni che ve
  // Da usare solo in casi rari (es. Ru
  doAlways();

  // A
  if (condizione di terminazione) {
    doSomething;
    return;
  }

  // Potrebbe essere anche un while ()
  for () {

    // B
    generaNuovaSoluzioneParziale;


    if (filtro) { // C
      recursive (..., level + 1);
    }

    // D
    backtracking;
  }

}
```

Recursion myths

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

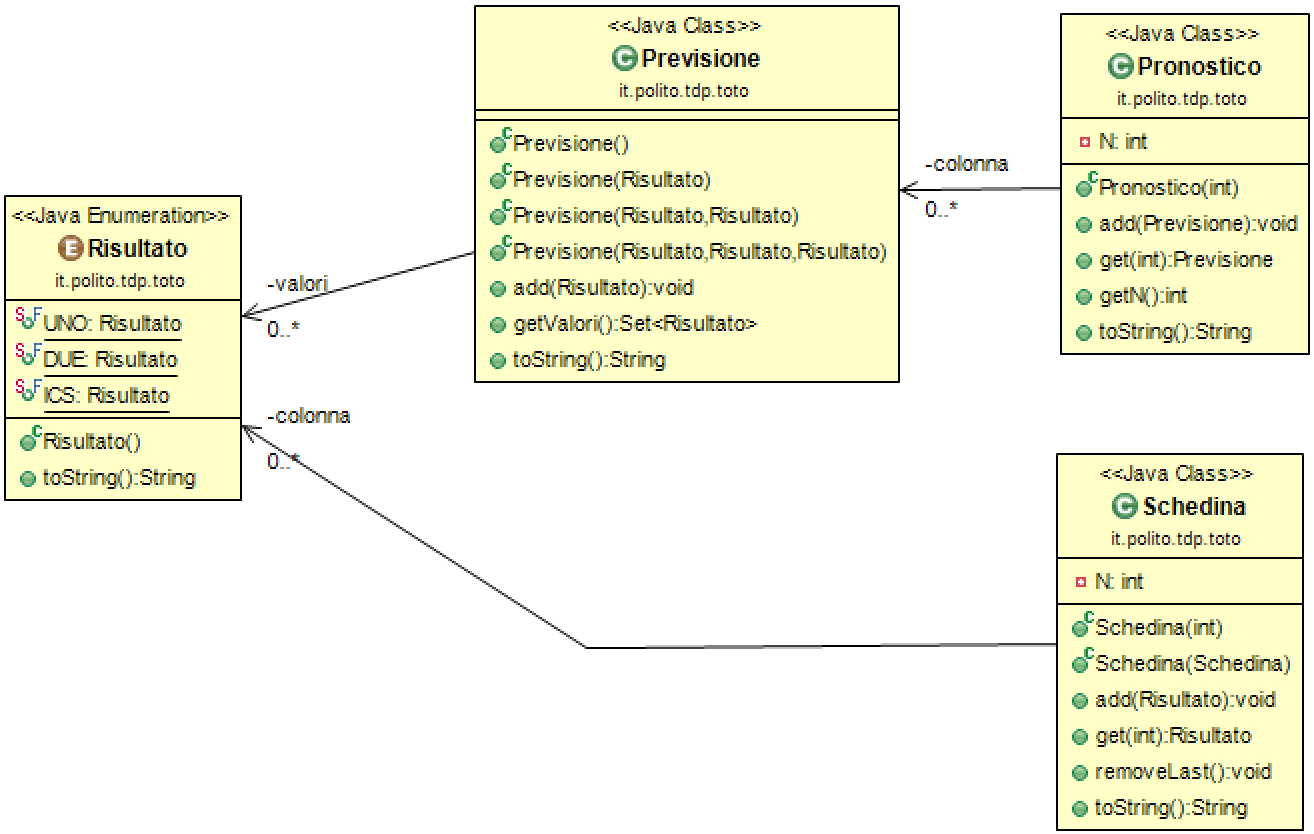


Simple recursive algorithms

Recursion

Schedina Totocalcio

Classi



<<Java Class>>
Previsione
it.polito.tdp.toto
Previsione()
Previsione(Risultato)
Previsione(Risultato,Risultato)
Previsione(Risultato,Risultato,Risultato)
add(Risultato):void
getValori():Set<Risultato>
toString():String
<<Java Class>>
Pronostico
it.polito.tdp.toto
N: int
Pronostico(int)
add(Previsione):void
get(int):Previsione
getN():int
toString():String
-colonna
0..*
<<Java Enumeration>>
Risultato
it.polito.tdp.toto
UNO: Risultato
DUE: Risultato
ICS: Risultato
Risultato()
toString():String
-valori
0..*
-colonna
0..*
<<Java Class>>
Schedina
it.polito.tdp.toto
N: int
Schedina(int)
Schedina(Schedina)
add(Risultato):void
get(int):Risultato
removeLast():void
toString():String

Fibonacci Numbers

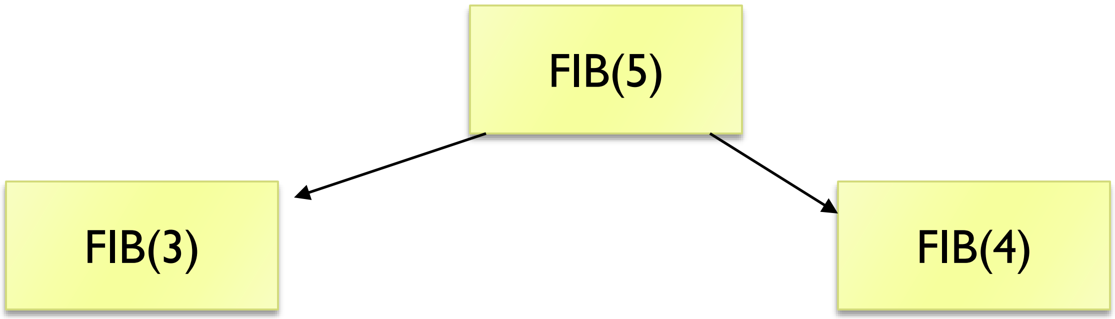
- Problem:
  - Compute the N-th Fibonacci Number
- Definition:
  - $FIB_{N+1} = FIB_N + FIB_{N-1}$ for $N>0$
  - $FIB_1 = 1$
  - $FIB_0 = 0$

Recursive solution

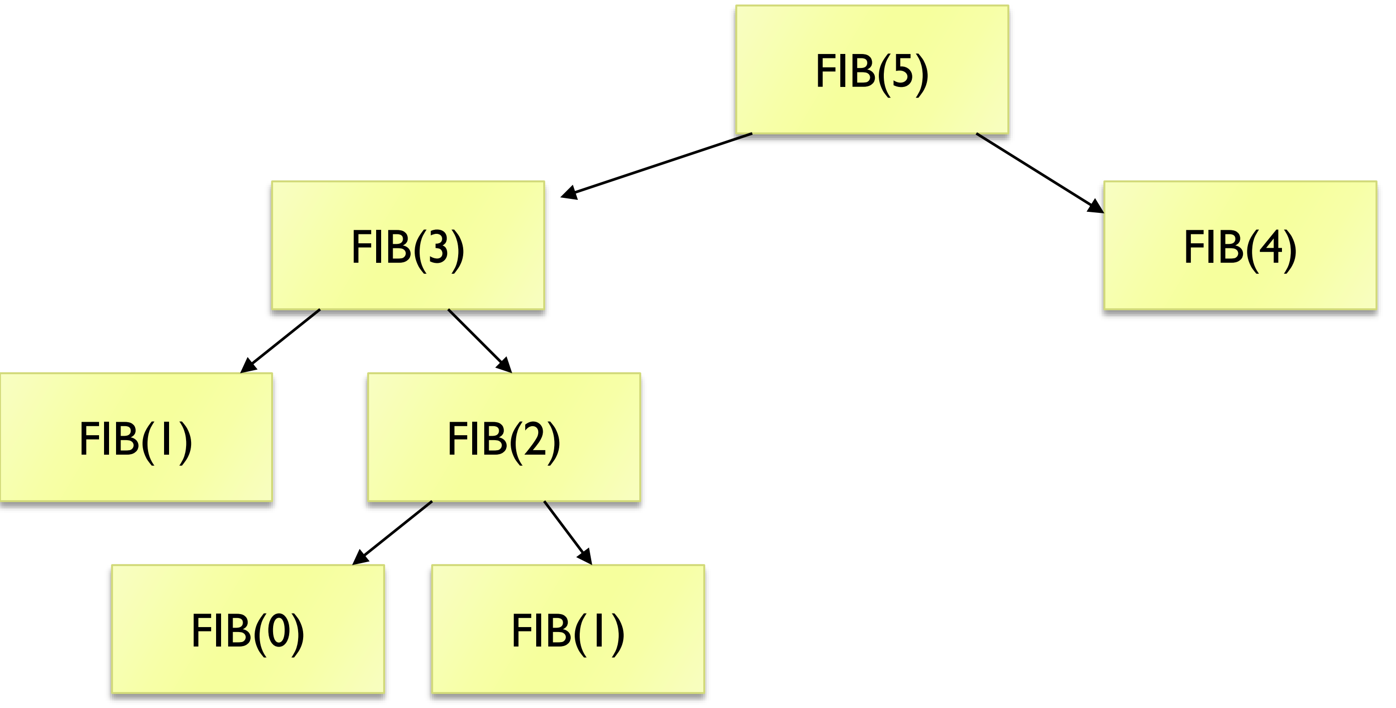
```
public long recursiveFibonacci(long N) {
    if(N==0)
        return 0 ;
    if(N==1)
        return 1 ;

    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
```

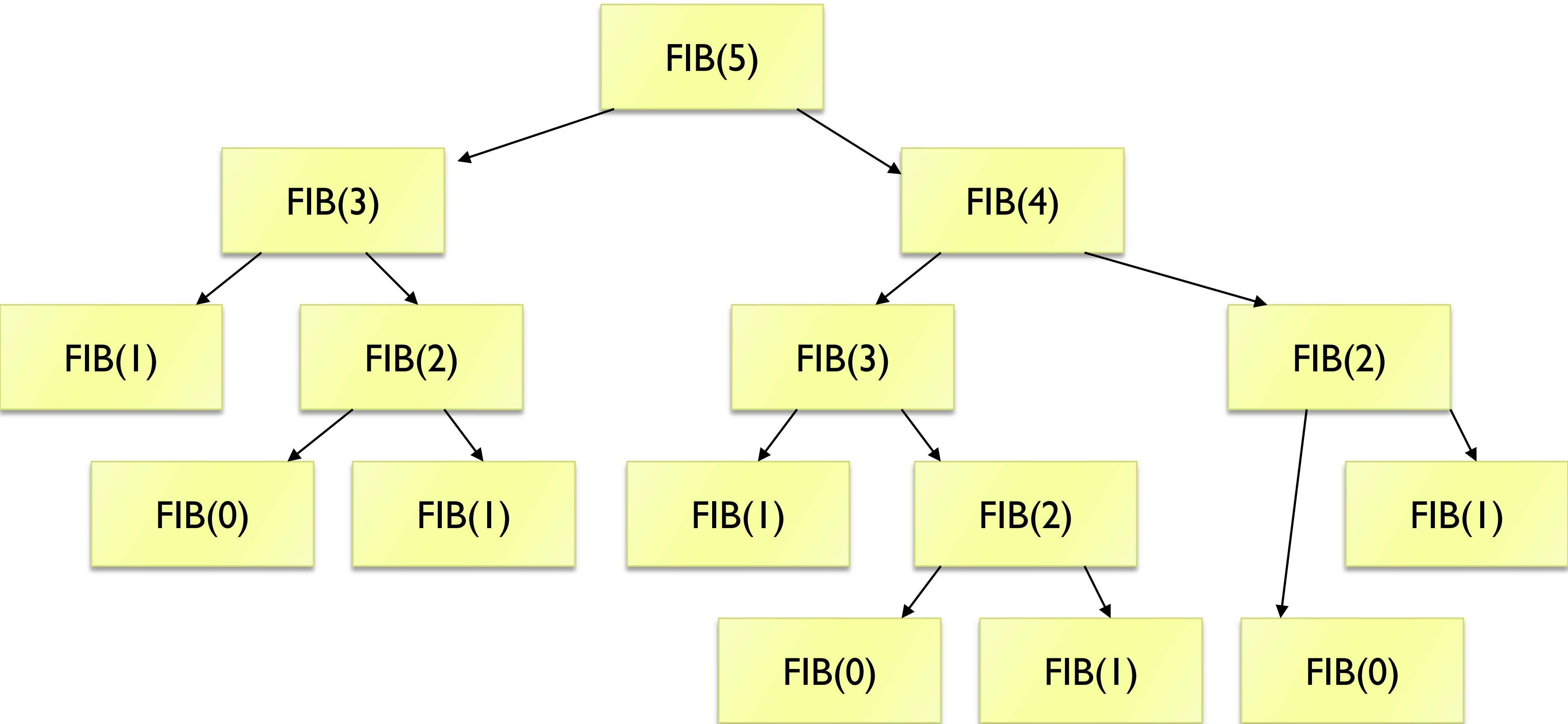
Analysis



Analysis

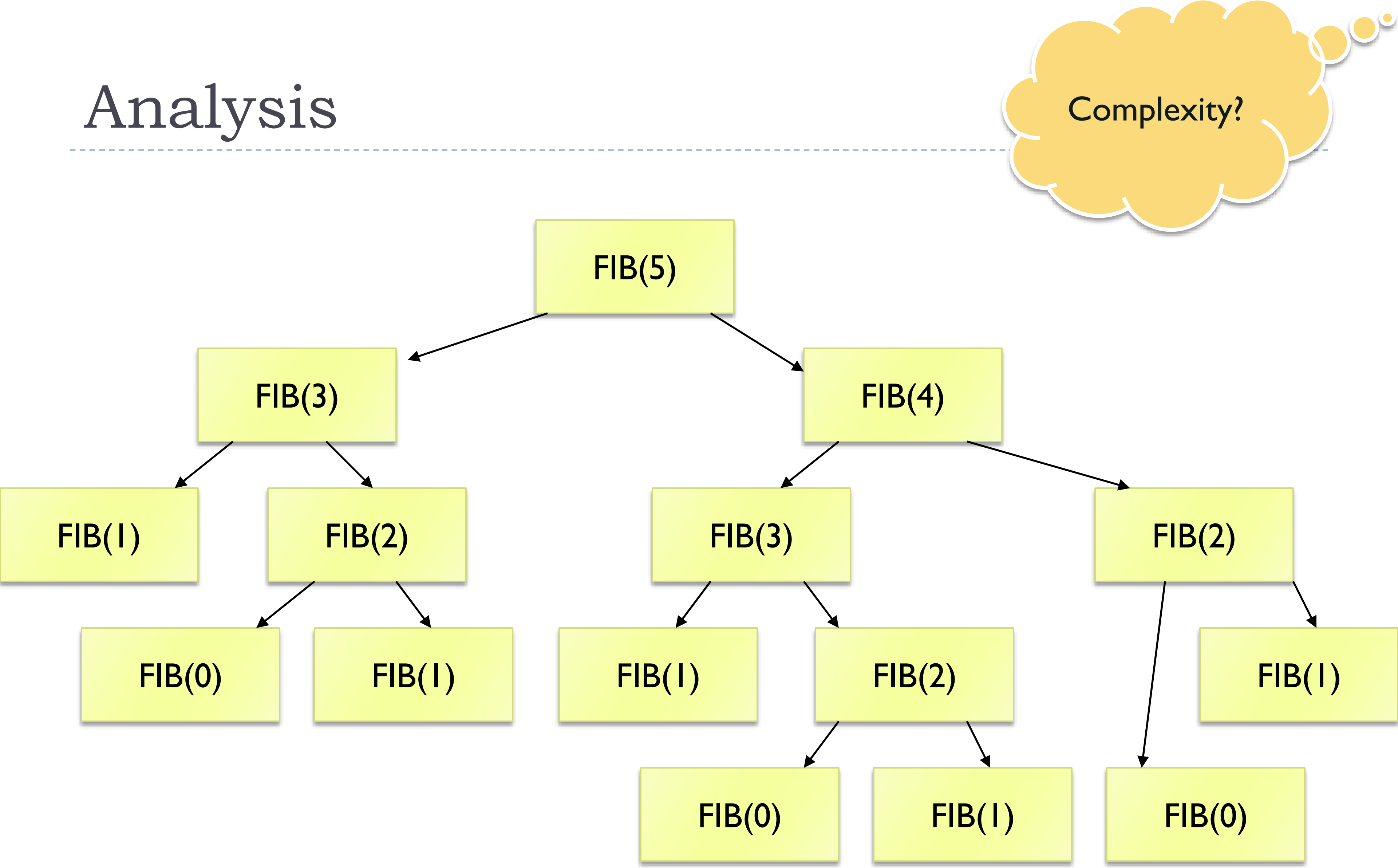


Analysis



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)

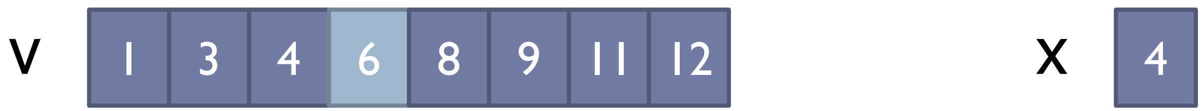
Analysis


Complexity?
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)

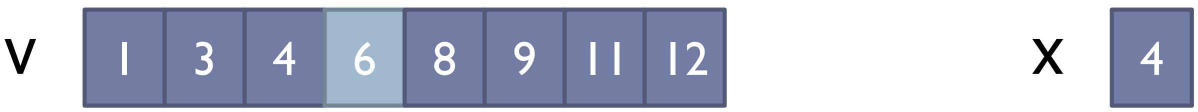
Example: dichotomic search

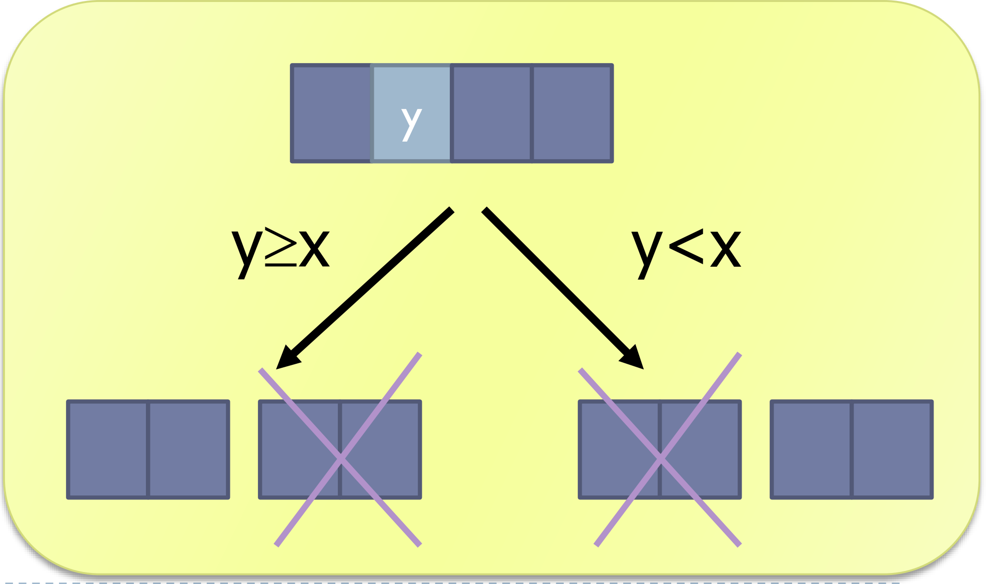
- Problem
  - Determine whether an element x is **present** inside an ordered **vector** v[N]
- Approach
  - Divide the vector in two halves
  - Compare the middle element with x
  - Reapply the problem over one of the two halves (left or right, depending on the comparison result)
  - The other half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

Example

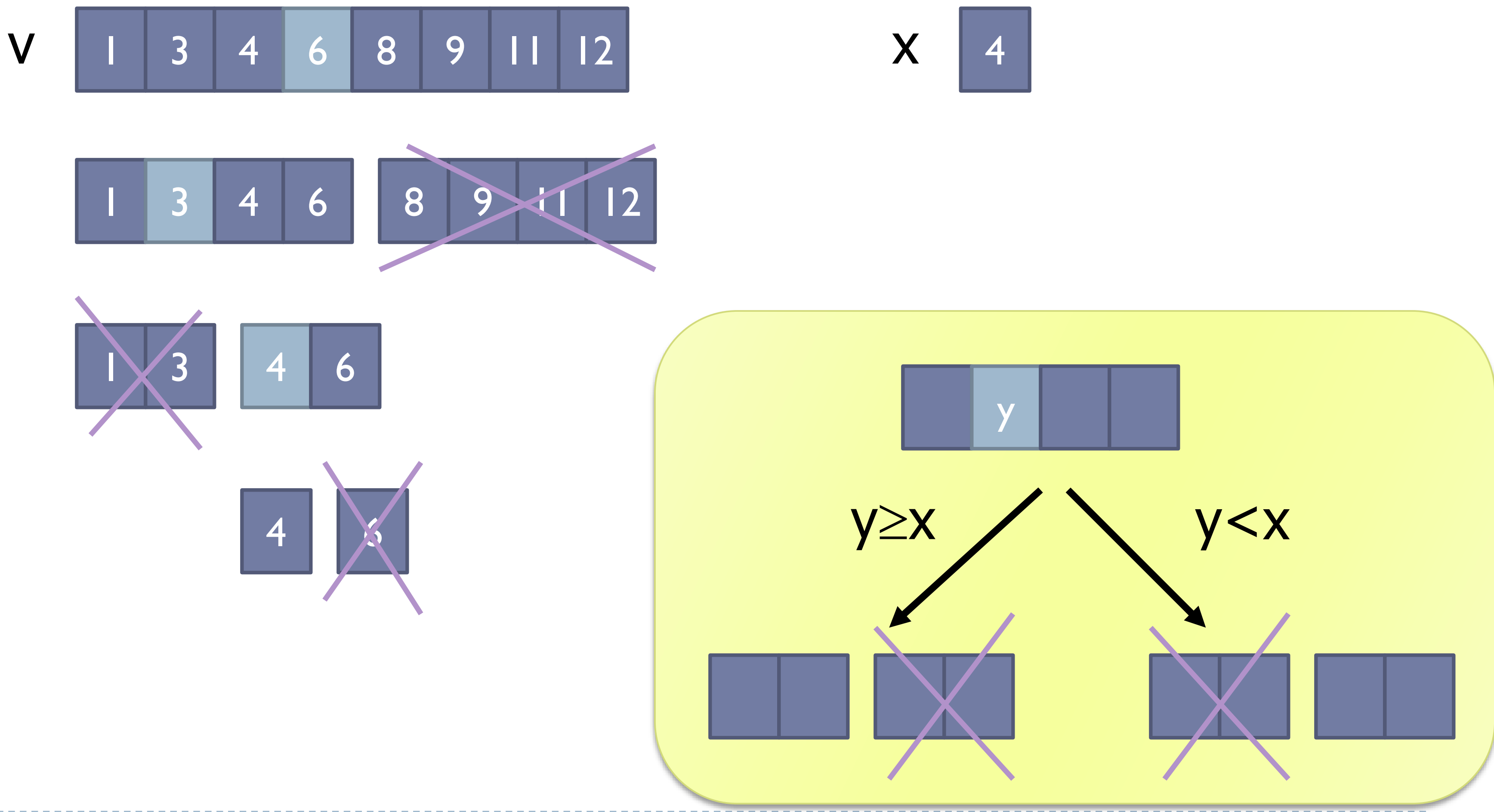


Example





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

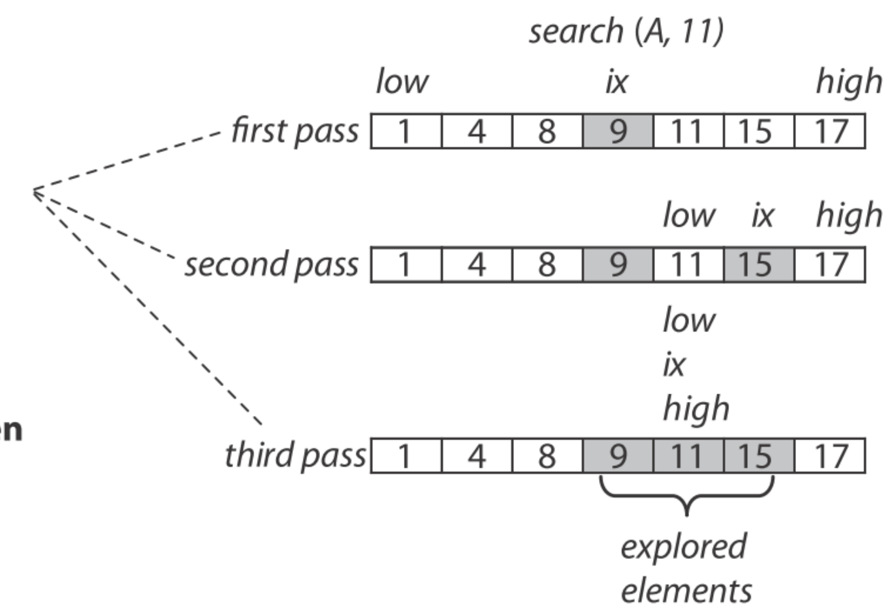
BINARY SEARCH

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`.

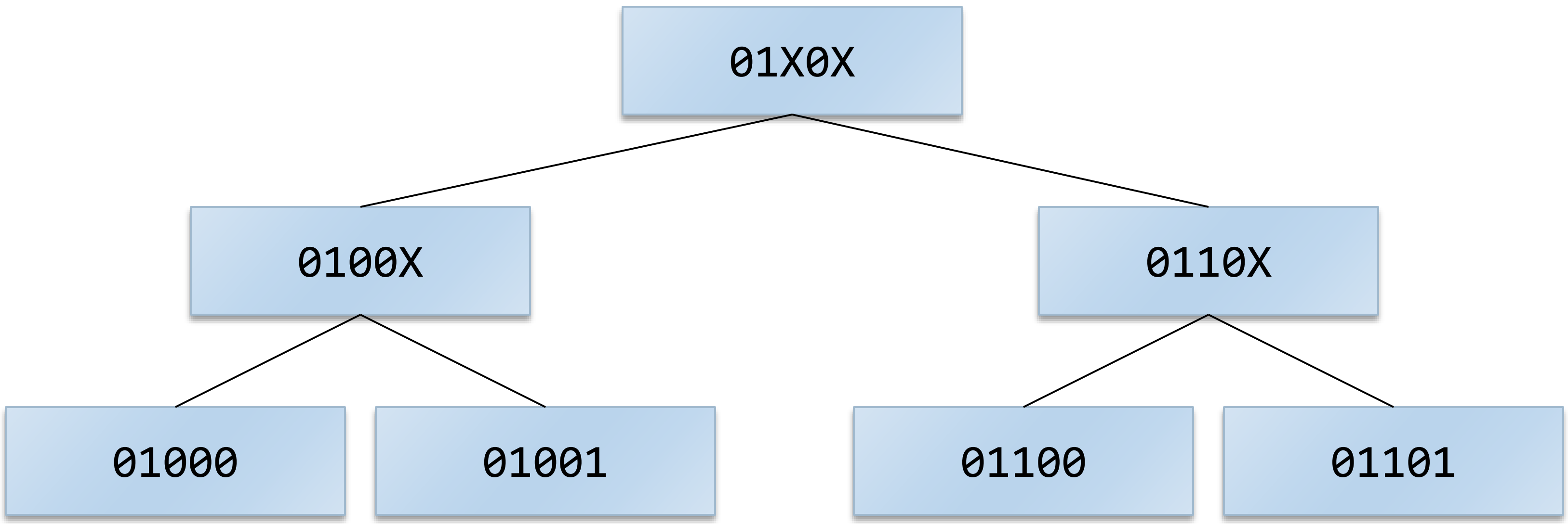
X-Expansion

- 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

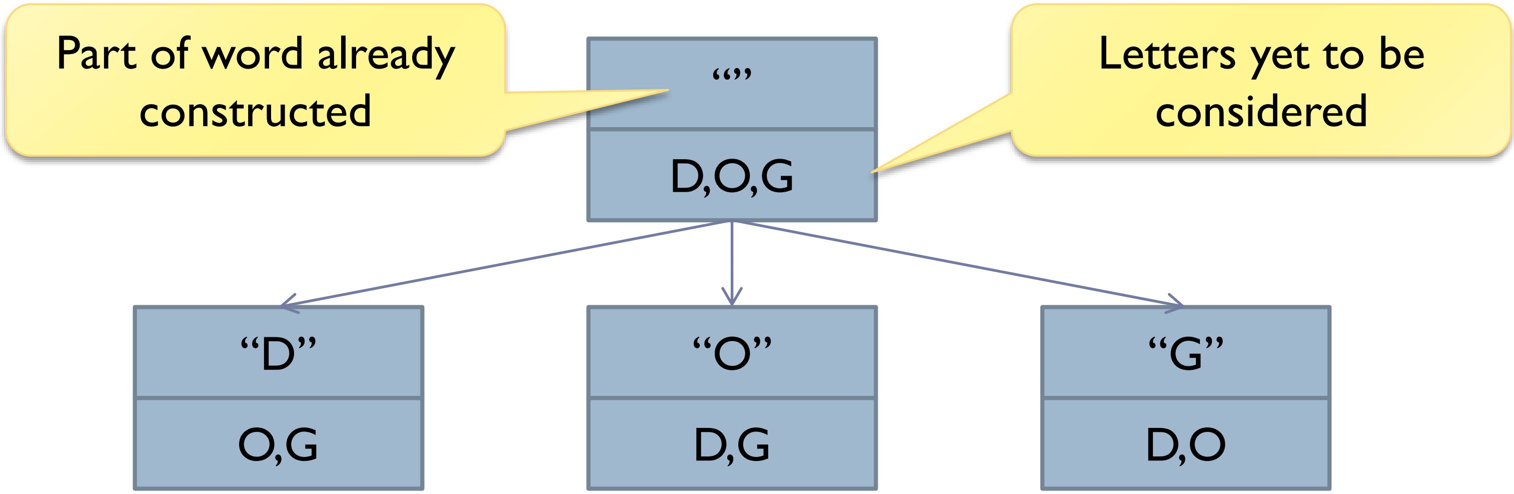


Exercise: Anagram

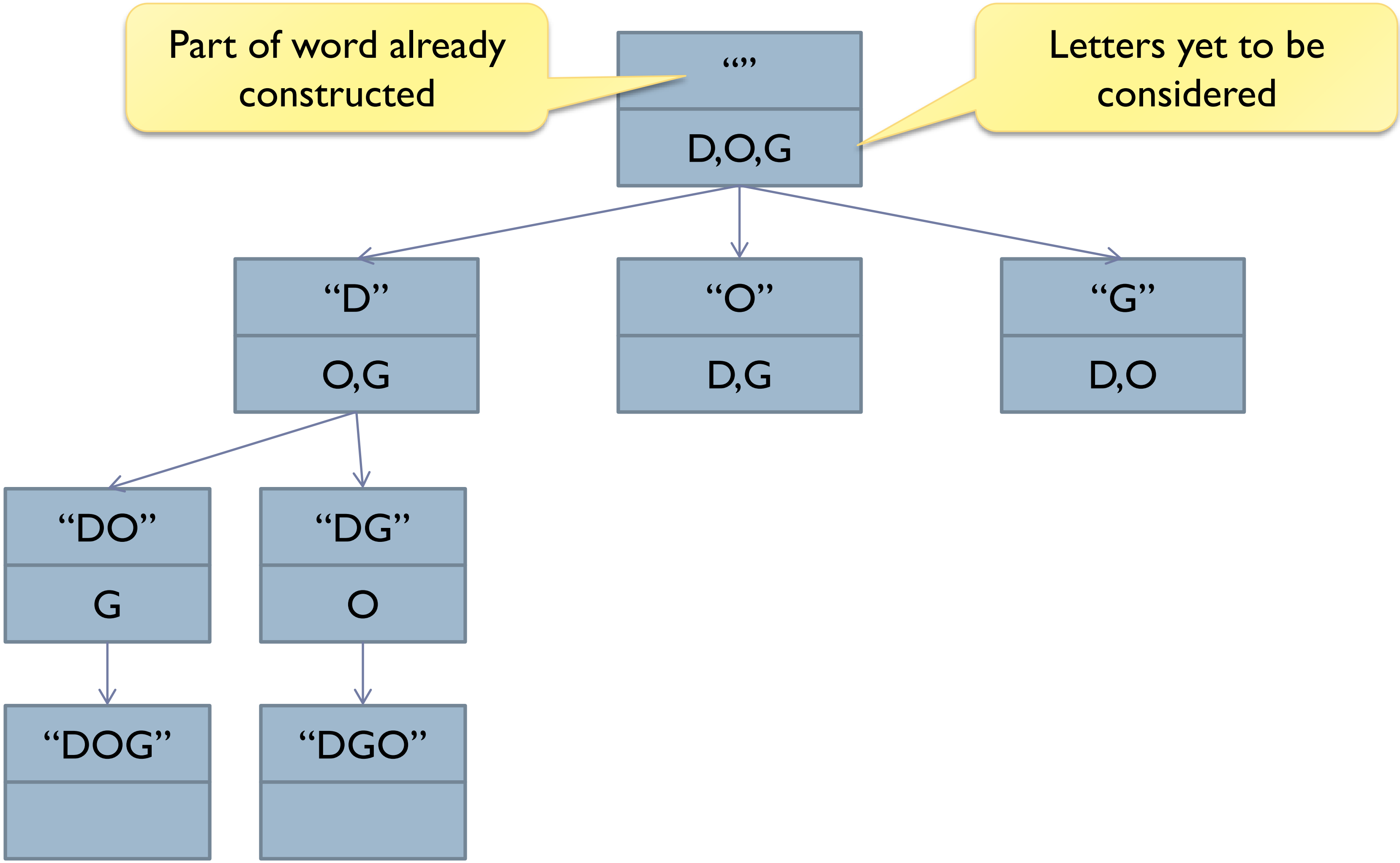
- Given a word, find all possible anagrams of that word
  - Find all permutations of the elements in a set
  - Permutations are N!
- E.g.: «Dog» → dog, dgo, god, gdo, odg, ogd

Anagrams: recursion tree

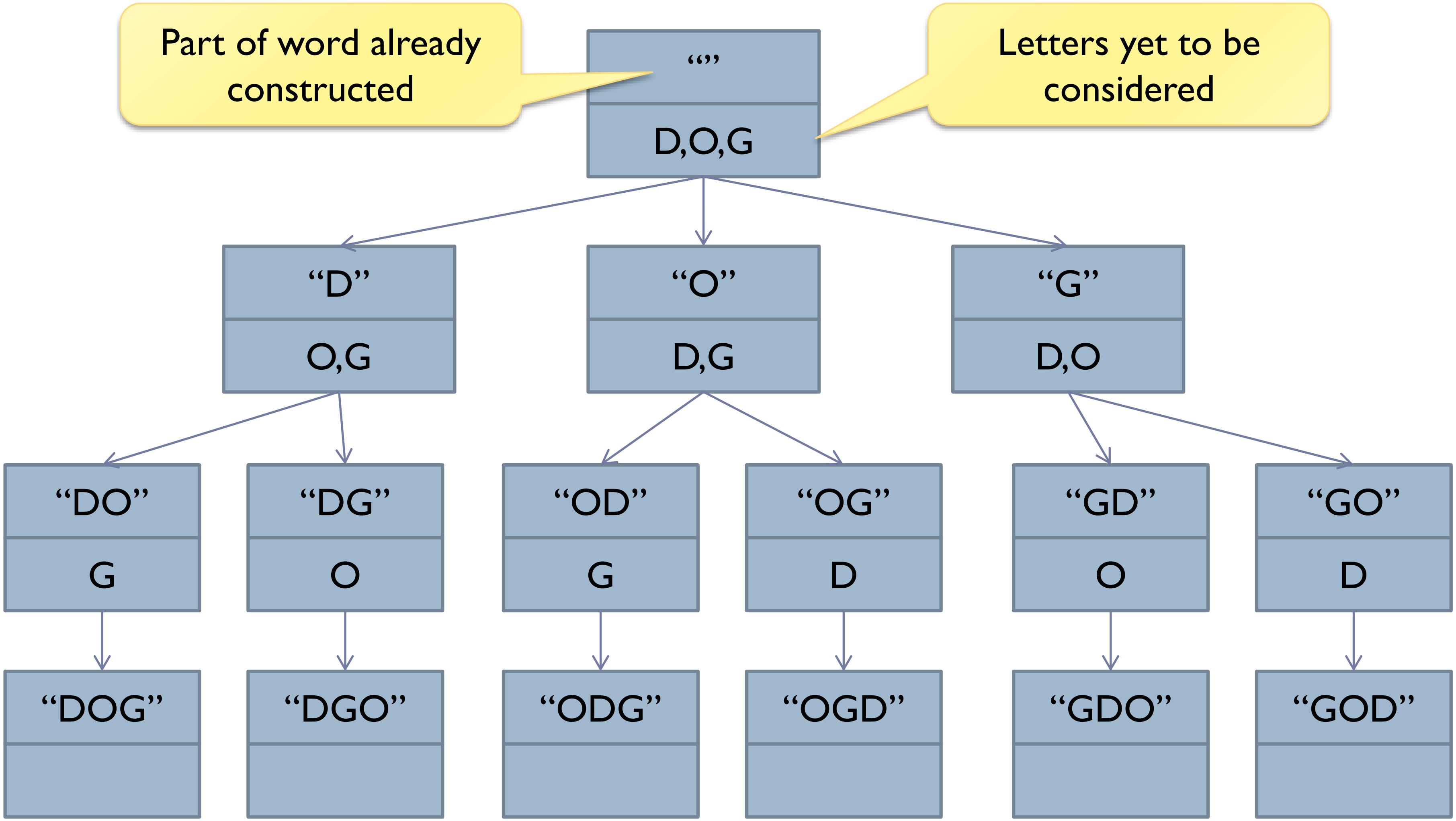
Anagrams: recursion tree



Anagrams: recursion tree



Anagrams: recursion tree

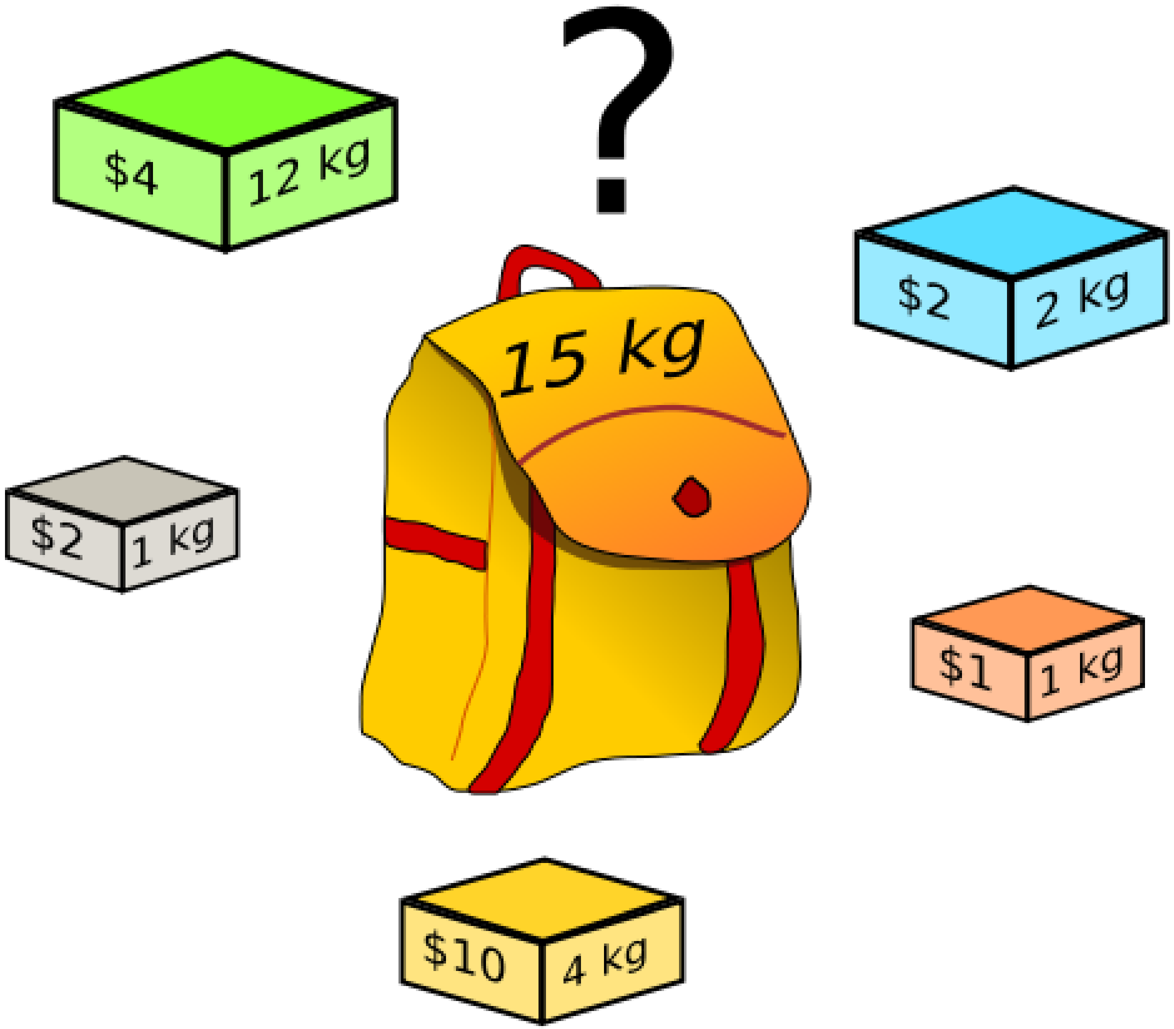


Anagrams: problem variants

- Generate only anagrams that are "valid" words
  - At the end of recursion, check the dictionary
  - During recursion, check whether the current prefix exists in the dictionary
- Handle words with multiple letters: avoid duplicate anagrams
  - E.g., "seas" → seas and seas are the same word
  - Generate all and, at the end or recursion, check if repeated
  - Constrain, during recursion, duplicate letters to always appear in the same order (e.g, s alwaws before s)

http://wordsmith.org/anagram/index.html

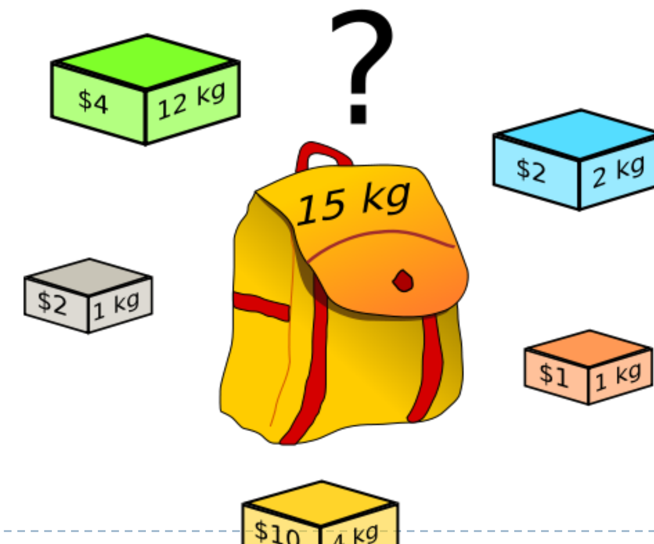
The Knapsack Problem



The Knapsack Problem

Input: Weight of N items $\{w_1, w_2, \ldots, w_n\}$
Cost of N items $\{c_1, c_2, \ldots, c_n\}$
Knapsack limit S

Output: Selection for knapsack: $\{x_1,x_2,\ldots x_n\}$
where $x_i \in \{0,1\}$.



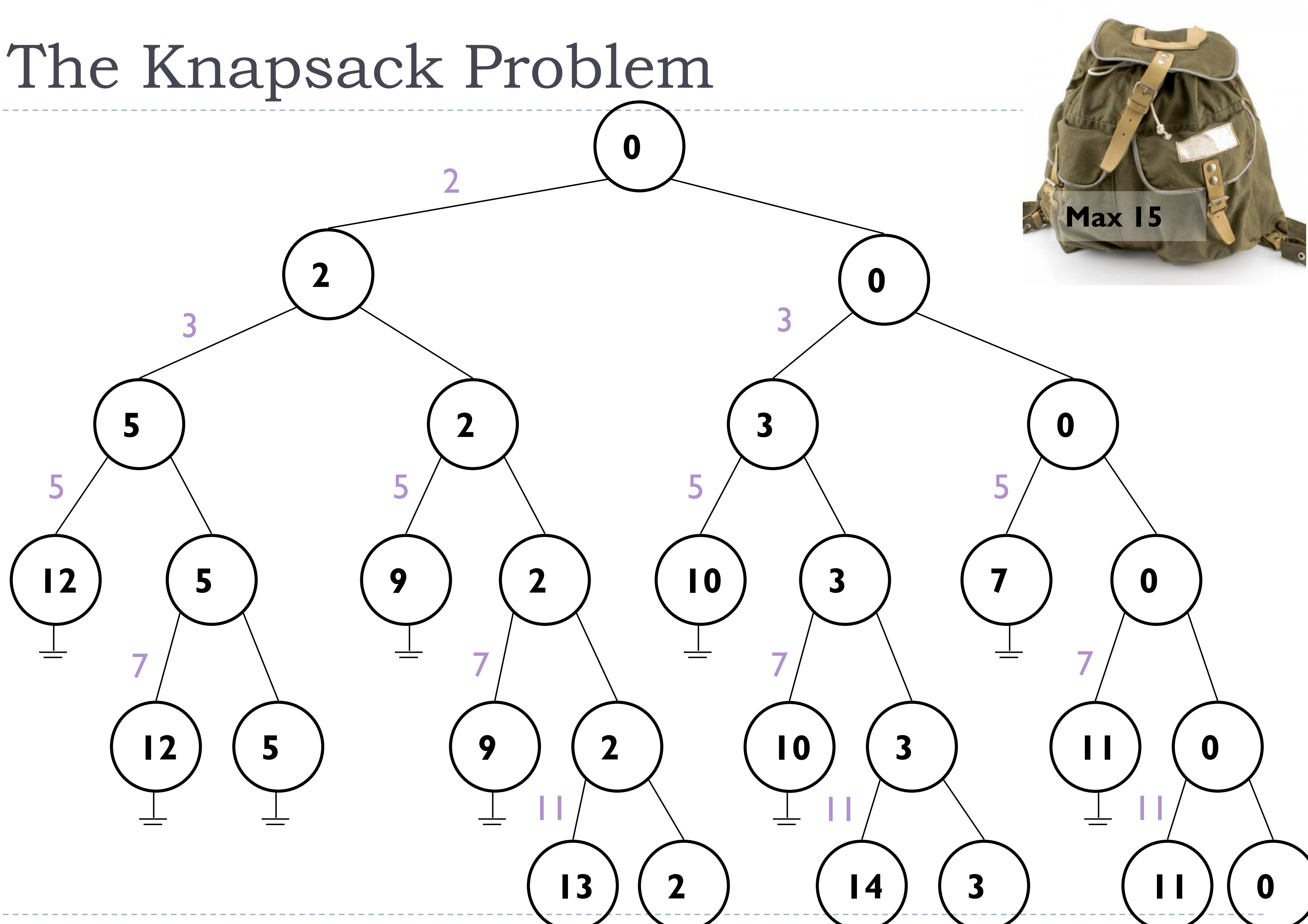
Sample input:

$w_i=\{1,1,2,4,12\}$

$c_i=\{1,2,2,10,4\}$

$S=15$

The Knapsack Problem



8	2	5	5	6	7	3	9
1	2	4	1	9	2	3	1
2	2	5	2	42	7	9	7
8	2	5	6	6	6	3	9
1	2	4	1	9	2	3	1
2	7	1	1	4	7	8	9
2	3	5	3	1	8	9	9
8	2	3	1	6	7	3	9

4	2	5	5	3	7	3	9
1	2	4	1	9	2	3	1
2	2	5	2	42	7	1	3
8	2	5	6	1	1	1	9
1	2	4	1	9	2	3	1
2	7	1	1	4	7	8	2
2	3	5	3	1	8	9	9
8	2	3	1	6	7	3	9

Exercise: Binomial Coefficient

- Compute the Binomial Coefficient (n m) exploiting the recurrence relations (derived from Tartaglia's triangle):

$$\begin{cases} \binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \\ \binom{n}{n} = \binom{n}{0} = 1 \\ 0 \le n, \quad 0 \le m \le n \end{cases}$$

Exercise: Determinant

- Compute the determinant of a square matrix
- Remind that:
  - $\det(M_{1x1}) = m_{1,1}$
  - $\det(M_{NxN})$ = sum of the products of all elements of a row (or column), times the determinants of the (N-1)x(N-1) sub-matrices obtained by deleting the row and column containing the multiplying element, with alternating signs $(-1)^{(i+j)}$.

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} M_{i,j} = \sum_{i=1}^{n} (-1)^{i+j} a_{i,j} M_{i,j}.$$

Laplace's Formula, at
http://en.wikipedia.org/wiki/Determinant

Recursive vs Iterative strategies

Recursion

Recursion and iteration

- Every **recursive** program can **always** be implemented in an **iterative** manner
- The best solution, in terms of efficiency and code clarity, depends on the problem

Example: Factorial (iterative)

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

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

    for (long i=2; i<=N; i++)
        result = result * i ;

    return result ;
}
```

Fibonacci (iterative)

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

  // now we know that N >= 2
  long i = 2 ;
  long fib1 = 1 ; // fib(N-1)
  long fib2 = 0 ; // fib(N-1)

  while( i<=N ) {
    long fib = fib1 + fib2 ;
    fib2 = fib1 ;
    fib1 = fib ;
    i++ ;
  }

  return fib1 ;
}
```

Dichotomic search (iterative)

```
public int findIterative(int[] v, int x) {
  int a = 0 ;
  int b = v.length-1 ;

  while( a != b ) {
    int c = (a + b) / 2; // middle point
    if (v[c] >= x) {
      // v[c] is too large -> search left
      b = c ;
    } else {
      // v[c] is too small -> search right
      a = c+1 ;
    }
  }
  if (v[a] == x)
    return a;
  else
    return -1;
}
```

Exercises

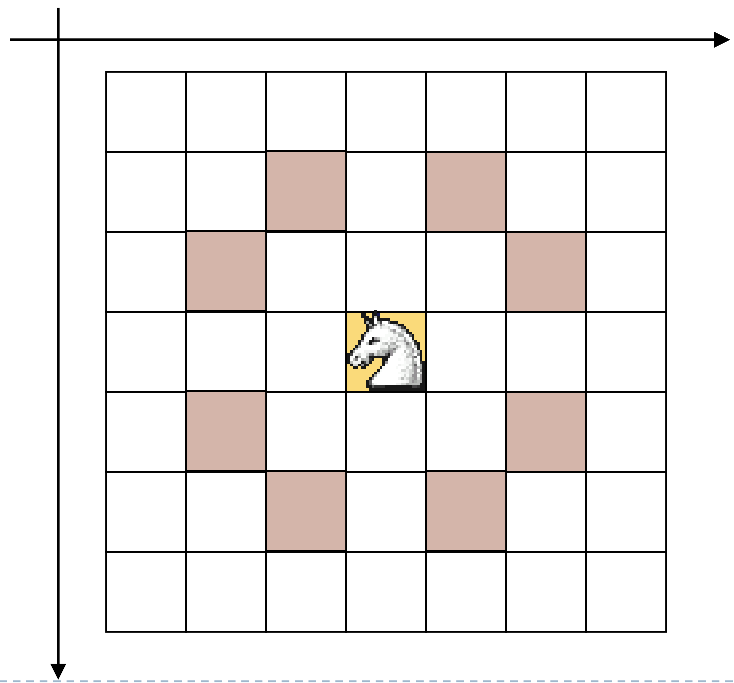
- Create an iterative version for the computation of the binomial coefficient (n m).
- Analyze a possible iterative version for computing the determinant of a matrix. What are the difficulties?
- Can you find a simple iterative solution for the X-Expansion problem? And for the Anagram problem?

More complex examples of recursive algorithms

Recursion

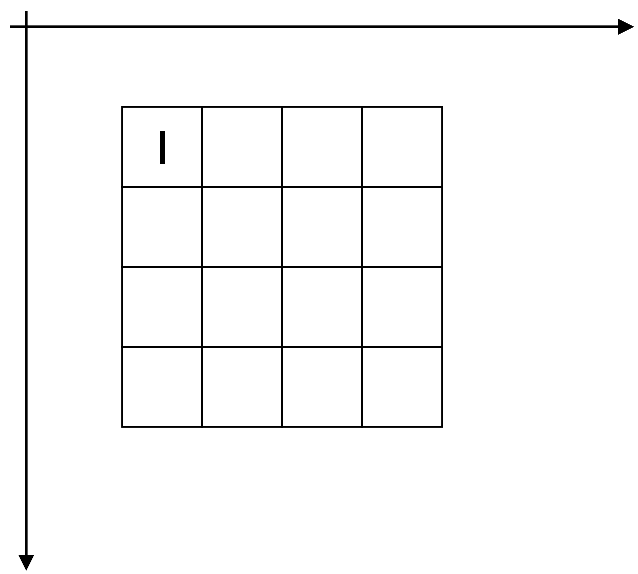
Knight's tour

- Consider a NxN chessboard, with the Knight moving according to Chess rules
  - The Knight may move in 8 different cells
- We want to find a **sequence** of moves for the Knight where
  - **All** cells in the chessboard are visited
  - Each cell is touched exactly **once**
- The starting point is arbitrary

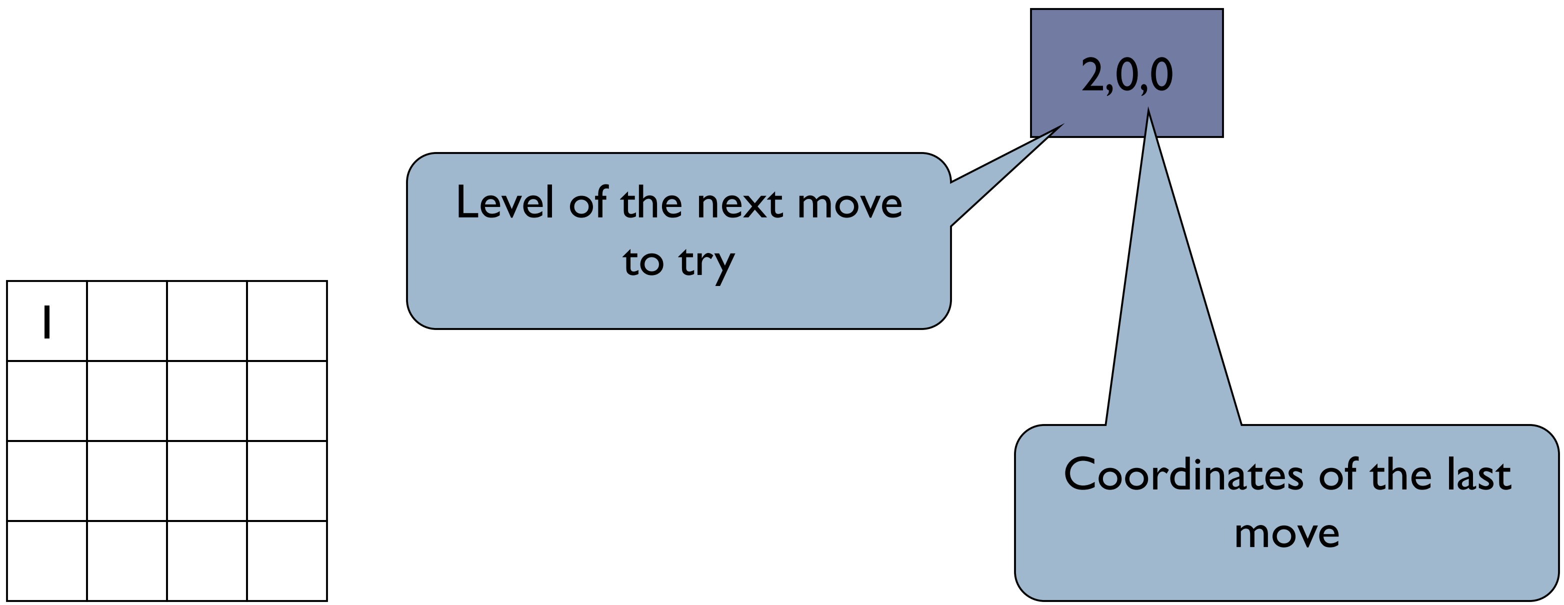


Analysis

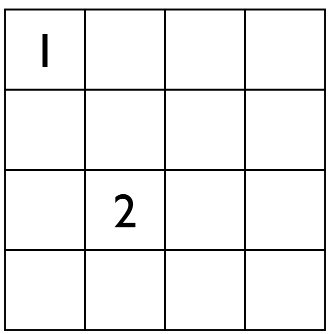
- Assume N=4

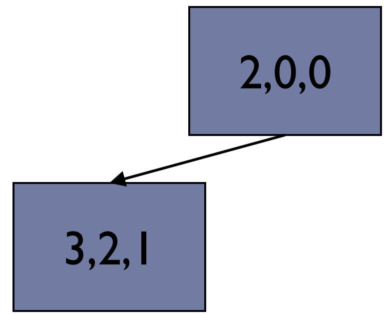


Move 1


1
2,0,0
Level of the next move to try
Coordinates of the last move

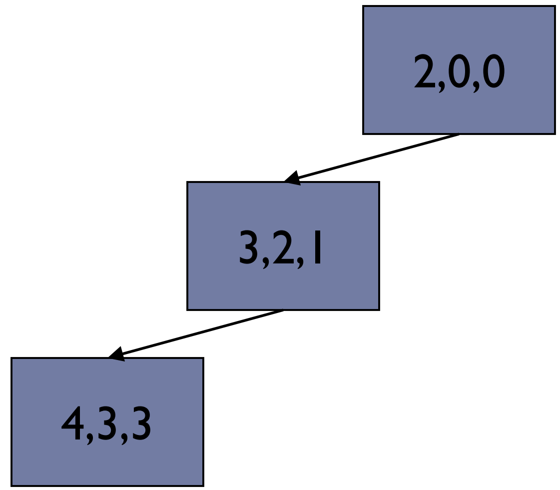
Move 2





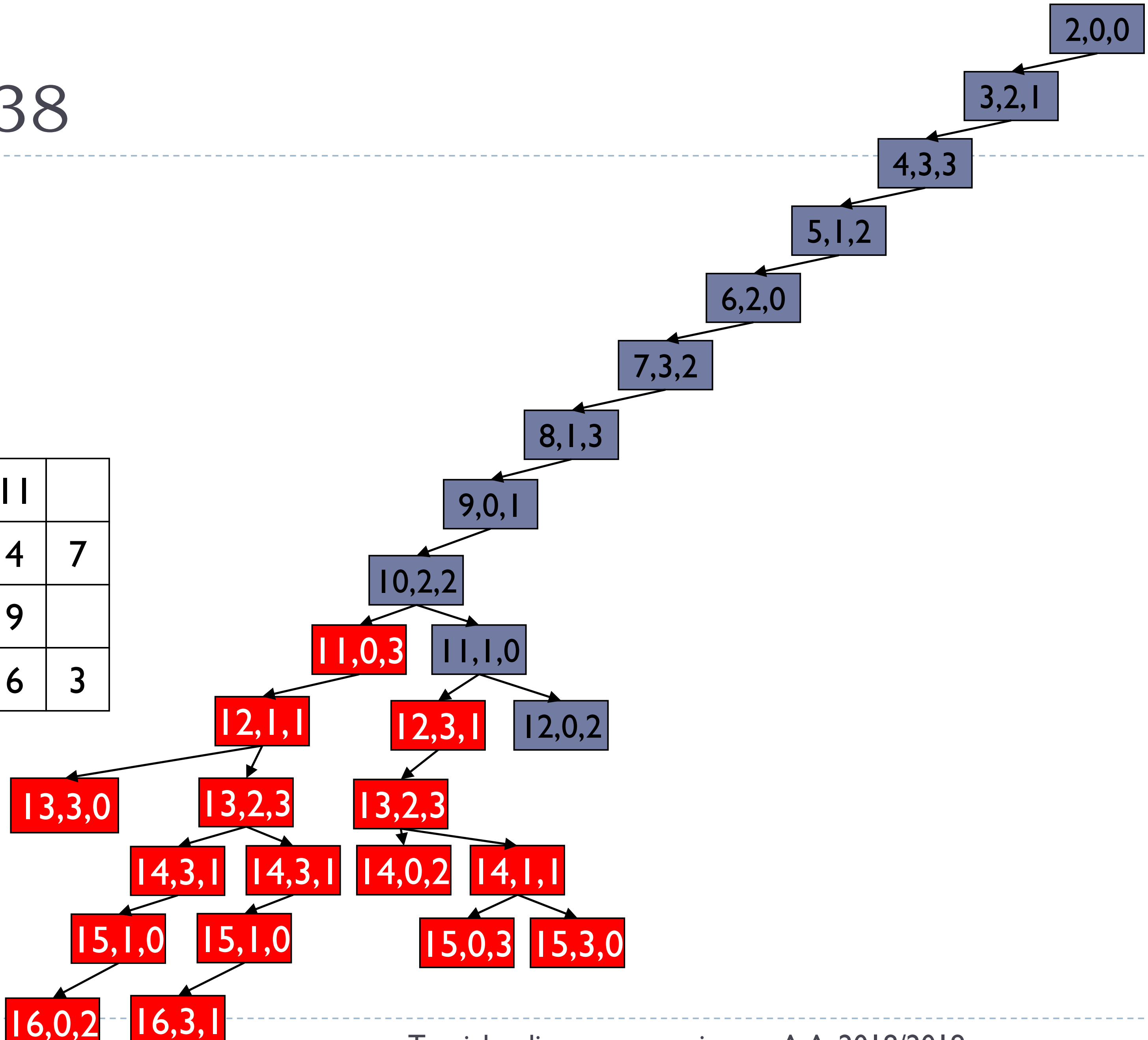
Move 3

1			
	2		
			3



Move 38

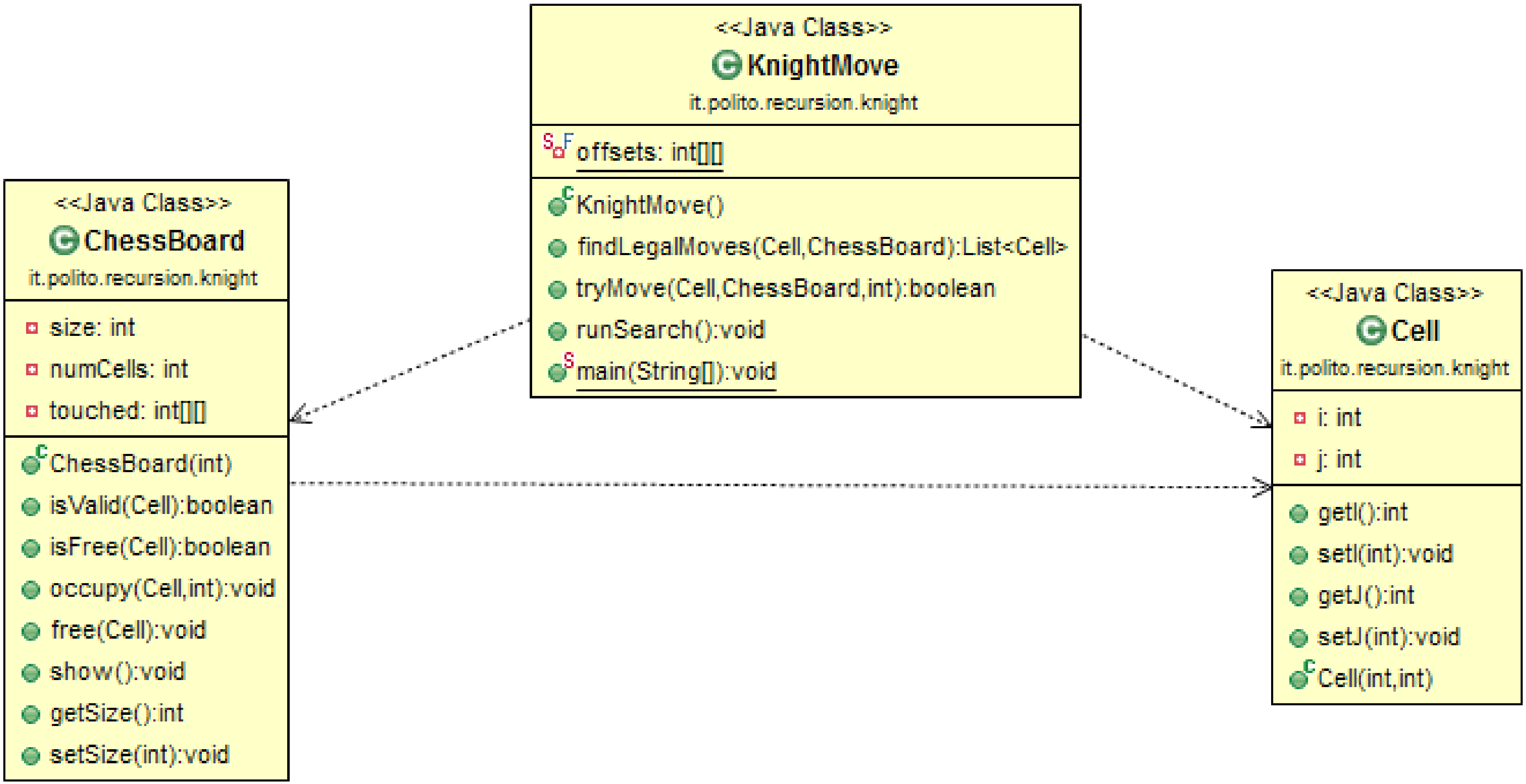
1	8	11	
10		4	7
5	2	9	
		6	3



Complexity

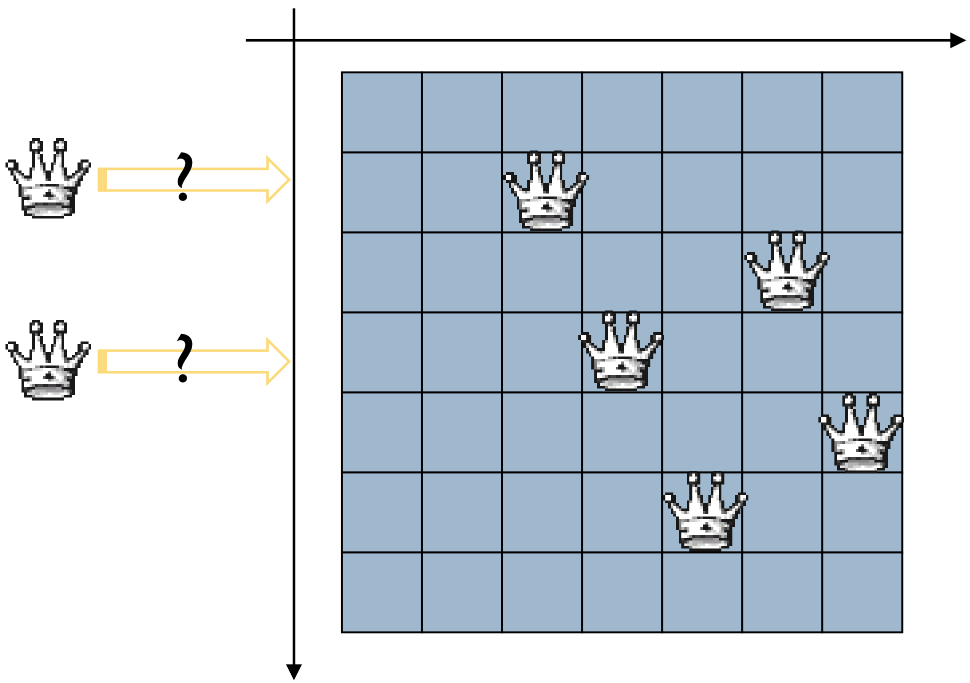
- The number of possible moves, at each step, is at most 8.
- The number of steps is N^2.
- The solution tree has a number of nodes $\leq 8^{N^2}$.
- In the worst case
  - The solution is in the right-most leave of the solution tree
  - The tree is complete
- The number of recursive calls, in the worst case, is therefore $\Theta(8^{N^2})$.

Implementation



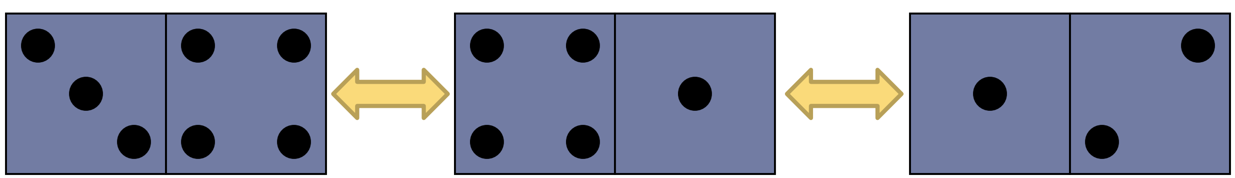
The N Queens

- Consider a NxN chessboard, and N Queens that may act according to the chess rules
- Find a position for the N queens, such that no Queen is able to attack any other Queen



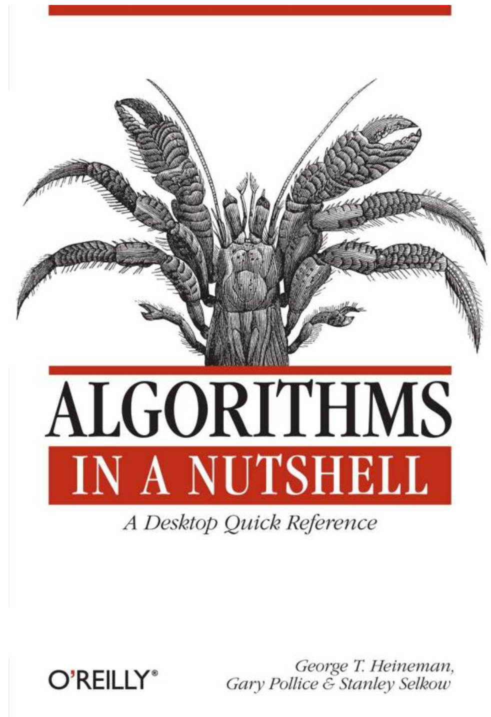
Domino game

- Consider the game of Domino, composed of two-sided pieces: each side is labeled with a number from 0 to 6. All combinations of number pairs are represented exactly once.
- Find the longest possible sequence of pieces, such that consecutive pieces have the same value on the adjacent sides.



Resources

- Algorithms in a Nutshell, By George T. Heineman, Gary Pollice, Stanley Selkow, O'Reilly Media



Licenza d’uso