POLITECNICO DI TORINO

e-Lite

Recursion (cont'ed)

Tecniche di Programmazione – A.A. 2014/2015

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

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 ↔ Paths
- Similar to D+I with { s, $\mathcal{P}'$ }

Complessità

Divide et Impera

- 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

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**

What about complexity?

- **Θ(1)** = complexity of **Solve_trivial**
  - …otherwise it wouldn't be trivial
- D(n) = complexity of **Divide**
- C(n) = complexity of **Combine**

- **T(n) = Θ(n log n)**

Exploration

- Explore ( ) {
  - List<Step> steps = **PossibleSteps** (Problem) ;
  - For (each **s** in steps) {
    - **Do** (**s**)
    - **Explore** ( ) ;
    - **Undo** (**s**)
  - }
- }

Exploration

- Explore ( ) {
  - List<Step> steps = PossibleSteps (Problem) ;
  - For (each **s** in steps) {
    - **Do** (**s**)
    - **Explore** ( ) ;
    - **Undo** (**s**)
  - }
- }

Local variable

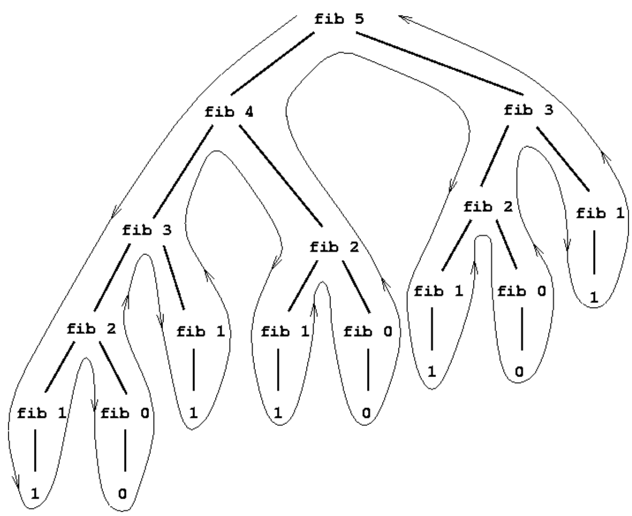
Update "global" status

Backtrack!

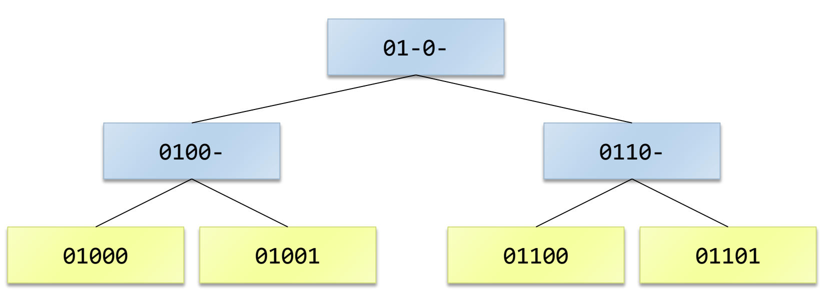
What about complexity?

- (Almost always)
- **$T(n) = \Theta(e^n)$**

Recursion Tree (exploration)



Don't care expansion



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

More complex examples of recursive algorithms

Recursion

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

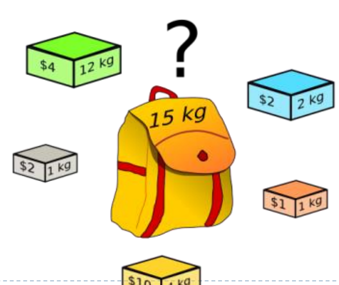
Output: Selection for knapsack: $\{x_1, x_2, \ldots x_n\}$ where $x_i \in \{0,1\}$.

Sample input:

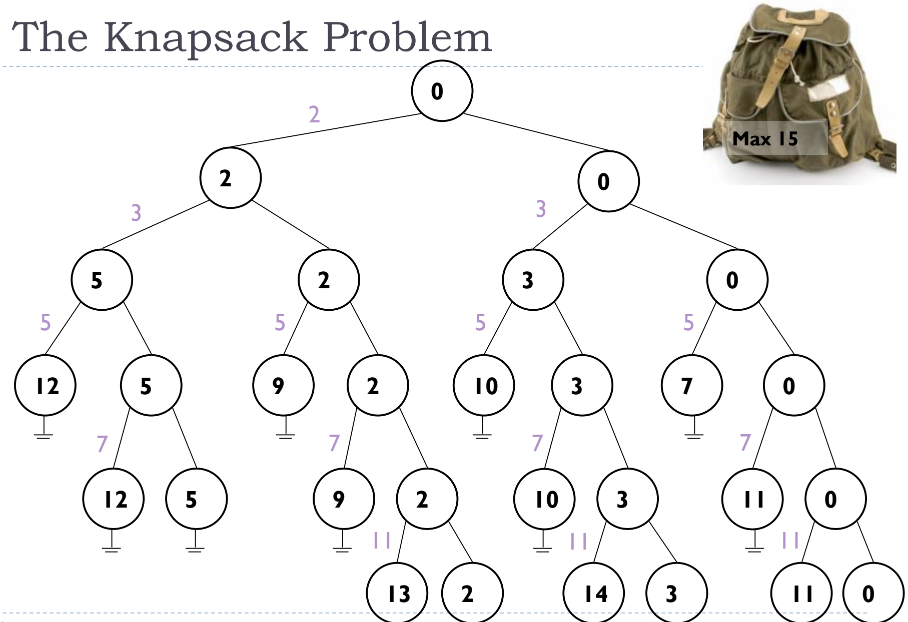
$w_i = \{1,1,2,4,12\}$

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

S=15



The Knapsack Problem



Licenza d'uso