CentraleSupélec

Algorithmics and complexity

TD 7/7 – Resolution of NP-hard Problems

The goal of this TD is to work on solving NP-hard problems.

Exercise 1 : Knapsack problem

Let us consider three different practical problems.

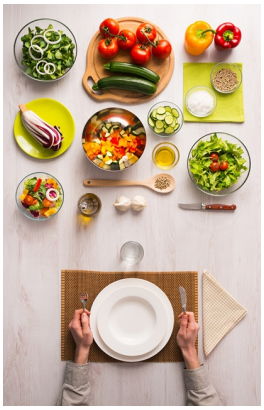
Cabbin Luggage

You live a nomad life, always traveling around the world. You have a set of items which you value differently, for exemple your laptop is something you always want with you hence has a high value. On the contrary, you like your SF t-shirt, but you can always buy a t-shirt at destination, so your SF t-shirt has a small value. You only carry cabin baggage. Different airlines put different limits on the weight of cabin baggage, for example 7kg or 10kg. How do you maximize the value of items you can bring with you, while remaining below the maxium weight allowed ?

Robbery

A thief manages to break into a jewelry. There are many items he can rob : watches, rings, necklaces... All these items have different values and different weights. The thief can only carry a limited weight of goods. Assume items can be split into categories, each category having a weight, a value and a number of items occuring in this category. How should the thief chose what to take ?

Balanced Lunch



You are given a list of nutritional values for a portion of each item served in the daily menu of your favorite university restaurant. For each item, you get the number of calories (or weight) and the protein value of a portion of the item. What do you chose to eat to maximize the proteins you get and without overtaking a certain amount of calories ?

Complexity

Question 1

All three problems can be formalized using the same framework. Give a formal description of this problem : what are the inputs, what is the decision question, what is the optimization question ?

Question 2

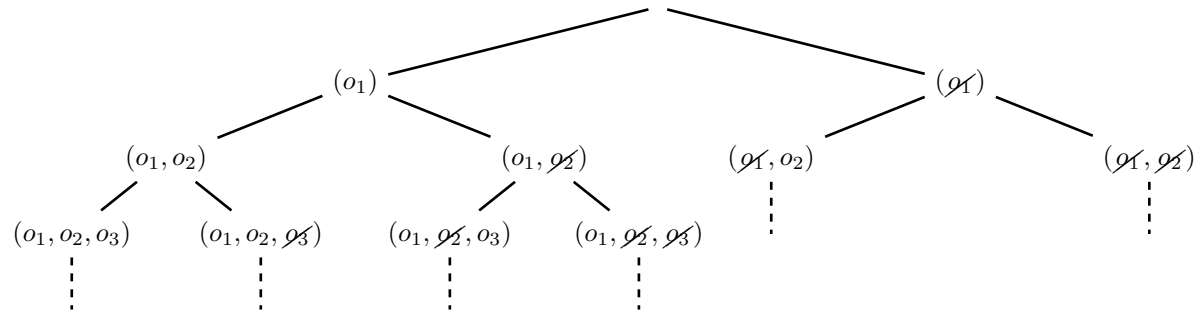
Show that this problem, known as **KNAPSACK problem**, is NP-complete. You can use the SUBSET-SUM problem which is known to be NP-complete :

SUBSET SUM
Inputs :
- a set A of non-negative integers $a_1, \dots, a_n$
- a value $t \in \mathbb{N}$

Question : Is there a subset $S \subseteq [1, n]$ with a total sum $\sum_{i \in S} a_i = t$?

Backtracking

We Consider a **backtracking** algorithm that decides the presence or absence of an item in the solution (under construction) at each step. We obtain the following branching binary tree of height n :



The backtracking algorithm will enumerate all possible solutions to find the one having the best score (the greatest value here).

The backtracking algorithm will avoid to explore a branch if it already exceed the weight limit.

Question 3

Write the Python code of this algorithm. To help you, go to the practice page of the TD :

https://wdi.centralesupelec.fr/1CC2000/TD7ProgEn

Greedy

A **greedy** algorithm constructs a solution step by step without going back on decisions already taken. There is no guaranty of optimality for the solution.

Question 4

Propose a **greedy** algorithm as efficient as possible for the knapsack problem.

Question 5

What is the time complexity of this algorithm ?

Question 6

What would be the worst instance for this algorithm ?

Question 7

Go back to the practice page of the TD. Complete the code concerning the greedy algorithm.

A *benchmark* is given in order to compare the execution time of *backtracking* and *greedy* on random instances. An other *benchmark* is given to compare the quality of the solutions obtained by those algorithms.

Dynamic Programming

We want to propose a dynamic programming algorithm to solve the knapsack problem. We note $V(i, j)$ the maximum total value that can be inserted into the knapsack of capacity j picking up objects among $o_1, \ldots, o_i$.

Question 8

What is the recursion formula that computes $V(i, j)$ for all i and j in $\mathbb{N}$?

Question 9

What would be the time and the space complexity of a dynamic programming algorithm implementing the formula above ?

Question 10

Does this means that $P = NP$?

Question 11

Back to the practice page of the TD, write and test this dynamic programming algorithm.

A *benchmark* is given to compare the execution time between all the algorithms.