UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIIC

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

# Data Structures and Algorithms

Lab Practice 5: Dynamic Programming

2019-2020 course

# Contents

# 1 Introduction

This practice aims to achieve the following objectives:

- Apply the dynamic programming technique to solve optimization problems.

# 2 The coin change problem

In this practice we will address the coin change problem. This problem, studied in the theory sessions, consists in determining the minimum amount of coins necessary to be able to change or pay a certain amount of money, and how much of each type of currency must be used. For example, to change a 5€ banknote into coins, we could use, among many other options, 500 coins of 1 cent, 5 coins of 1€, or 2 coins of 2€ + 1 coin of 1€. In this context, we will prefer the solution that uses less coins, that is, the third of them.

Formally, given a set of $N$ coins, each one with different monetary values $v = \{v_1, \ldots, v_N\}$, we have to find the number of coins of each type $x = \{x_1, \ldots, x_N\}$ that minimises the total number of coins used:

$$\hat{x} = \arg\min_{x} \sum_{i=1}^{N} x_i \tag{1}$$

with the restriction that the optimum number of coins of each type $\hat{x}$ satisfies the amount of money $A$ (*amount*) that we must change or pay:

$$\sum_{i=1}^{N} v_i x_i = A \tag{2}$$

For instance, in a problem consisting of $N = 3$ different coins, with monetary values $v = \{1, 4, 6\}$ m.u. (monetary units), and a total amount to change or pay of 8 m.u., the optimum solution would be $\hat{x} = \{0, 2, 0\}$ (2 coins). In comparison, a greedy algorithm would deliver a sub-optimal solution: $x = \{2, 0, 1\}$ (3 coins).

# 3 Resolution by dynamic programming

The dynamic programming technique allows us to efficiently solve optimization problems, such as the coin change problem, or the discrete backpack. The strategy is to divide the original problem into simpler (dependent) overlapping subproblems, solve them, and store their solutions in memory.

Thus, each time the same subproblem is addressed again, instead of solving it, the stored solution is used. In this way, we sacrifice (increase) the spatial cost in benefit (reduce) the temporal cost.

As seen in the theory sessions, the pseudocode for the solution to the problem of coin change through dynamic programming is as follows:

```
function coinChange(v[1..N], A:int):int
  C[1..N,0..A]
  for i:=1 to N do C[i,0]:=0 done
  for i:=1 to N do
    for j:=1 to A do
      C[i,j] := if i=1 then
                    if v[1]>j then infinity
                    else 1 + C[1,j-v[1]]
                else
                    if v[i]>j then C[i-1,j]
                    else min(C[i-1,j], 1 + C[i,j-v[i]])
                fi
    done
  done
  return C[N,A]
```

where `C` is the matrix that stores the solutions to all subproblems `C[i,j]`. This term `C[i,j]` can be read as the minimum number of coins needed to pay or change an amount of money $j$ (where $0 \leq j \leq A$) using only coins of type 1 to $i$ (where $1 \leq i \leq N$). The solution to the problem, therefore, will be determined by `C[N,A]`, that is, considering all ($N$) coin types and the original amount ($A$) we want to pay or change. In Appendix A you can check this same pseudocode but enriched with line-by-line comments, to facilitate its understanding.

In this pseudocode there are two important details that should be noted. First, it assumes that the vector of values of the different coin types `v` is sorted in ascending order. Second, at the end of the algorithm, we obtain and return the minimum number of coins in total (`C[N,A]`), but we do not remember the number of coins used of each type. Therefore, we must introduce the necessary changes to the algorithm to allow us to record these decisions, to finally determine the values of the solution $x = \{x_1, \ldots, x_N\}$. In practice, it will suffice to keep, for each sub-problem, the number of coins used of each type, taking into account the information provided by the sub-problems on which it depends.

# 4 Implementation details

In this practice we will implement a solution to the coin change problem through dynamic programming in Java, based on the pseudo-code that we have seen in the previous section. To do this,

we will create a *CoinChange* class, which will contain a static method called *coinChange*. This method, given an ordered array with the values of the considered coin types (`int [] v`), and an amount of money to change or pay (`int A`), returns the minimum amount of coins required, as well as how many coins of each type are needed. This would be the schema of the *CoinChange* class:

```
package P5;
public class CoinChange {
    public static int[] coinChange(int[] v, int A){...}
}
```

**IMPORTANT NOTE**: In order to jointly provide the solution to the problem (vector $\hat{x} = \{\hat{x_1}, \ldots, \hat{x_N}\}$) and the minimum value of that solution (`C[N,A]`), an array of `N+1` elements will be returned, storing, in position 0, the value of `C[N,A]`, and, from position 1 to $N$, values $\hat{x_1}, \ldots, \hat{x_N}$. Evaluation tests will assume this organization of the returned data.

# 5 Deliverables

As part of the assessment of the lab practice 5, the following exercise must be done:

1. Implementation of class *CoinChange*.

This exercise will be delivered as a single file named *CoinChange.java*. This class must be encapsulated into a package called `P5` (`package P5;`).

# A Pseudocode of algorithm *coinChange* with comments

```
function coinChange(v[1..N], A:int):int
  # v -> array of coin type values, sorted from lower to higher value
  # A -> amount of money to change/pay
  C[1..N,0..A] # sub-problems solutions matrix C[i,j]
  for i:=1 to N do C[i,0]:=0 done # solve cases where no money to be changed
  for i:=1 to N do # for an incremental set of coin types (1...i)
    for j:=1 to A do # for an amount of money j to pay/change
      C[i,j] := if i=1 then # sub-problem considering only the lowest coin
                    if v[1]>j then infinity # coin value > money: not feasible
                    else 1 + C[1,j-v[1]] # add 1 coin & substact coin value
                else # sub-problems considering 2 or more coin types
                    if v[i]>j then C[i-1,j] # coin value > money: discard coin
                    else min(C[i-1,j], 1 + C[i,j-v[i]])  # choose best option:
   discard this coin or use it.
                fi
    done
  done
  return C[N,A] # Return solution to the initial problem
```