**SUBJECT**: DESIGN AND ANALYSIS OF ALGORITHMS

**CODE**: 503040

Duration: 150 minutes

Allowed to use materials.

**LAB 06: Dynamic Programming (Part 1)**

# Objectives

Understand the properties of Dynamic Programming algorithm design technique

Be able to design, implement, and analyze Dynamic Programming algorithms solving common problems.

# Idea

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances

- solve smaller instances once

- record solutions in a table

- extract solution to the initial instance from that table

## An example of a dynamic programming algorithm

Implement and analyze a dynamic programming algorithm to compute n-th Fibonacci number

The implementation in Python is presented as follows

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **def** **fibonacci**(n):  """  calculates n-th Fibonacci numbers  input:  n(int) - the ordinal number >= 0  output:  F (int) - n-th Fibonacci number  """  F = [**0** **for** \_ **in** range(n+**1**)]  F[**0**] = **0**  F[**1**] = **1**  **for** i **in** range(**2**,n+**1**):  F[i] = F[i-**1**] + F[i-**2**]  **return** F[n]  **print**(fibonacci(**4**)) | |

Analysis:

1/ Basic operation: addition on line 13

2/ Worst case: as average case

3/Counting the number of basic operations in the worst case:

…

**Time efficiency**

***T*(*n*) = *n*-1 ∈ Θ(*n*)**

# Exercises

For each of the problems in this section, implement (in Python) and analyze a dynamic programming (DP) algorithm to solve the problem

**Warm up**

1. Computing a binomial coefficient

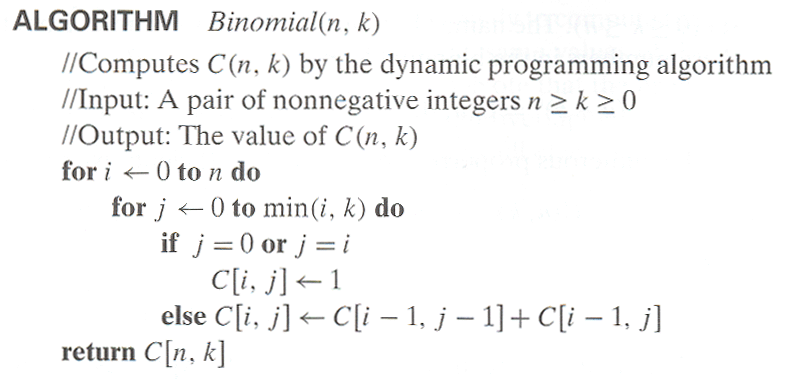
Hint:

Recurrence:

*C*(*n*,*k*) = *C*(*n-*1,*k*) + *C*(*n*-1,*k*-1) for *n > k* > 0

*C*(*n*,0) = 1, *C*(*n*,*n*) = 1 for *n* ≥ 0

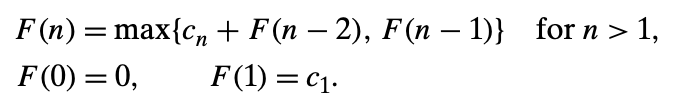
Pseudocode:

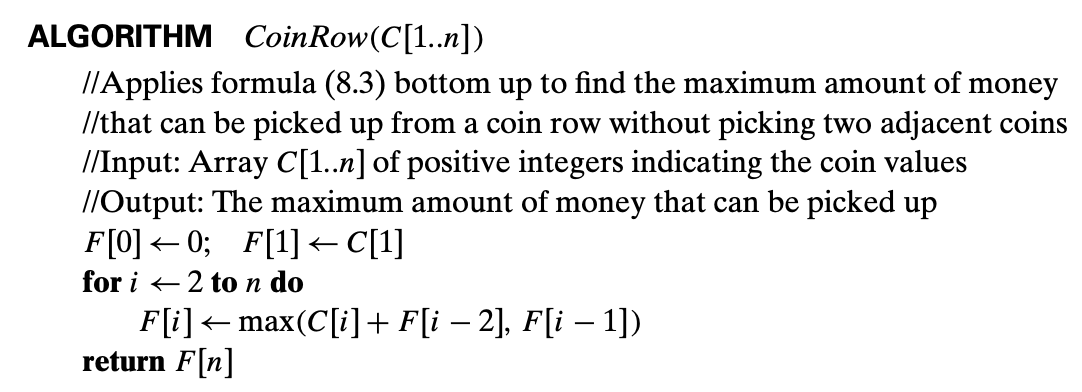


1. Coin-row problem

There is a row of n coins whose values are some positive integers c1, c2, . . . , cn, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

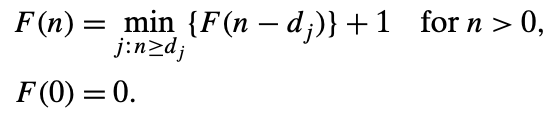
Let F (n) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F (n), we partition all the allowed coin selections into two groups: those that include the last coin and those without it. The largest amount we can get from the first group is equal to cn + F (n − 2)—the value of the nth coin plus the maximum amount we can pick up from the first n − 2 coins. The maximum amount we can get from the second group is equal to F (n − 1) by the definition of F (n). Thus, we have the following recurrence subject to the obvious initial conditions:

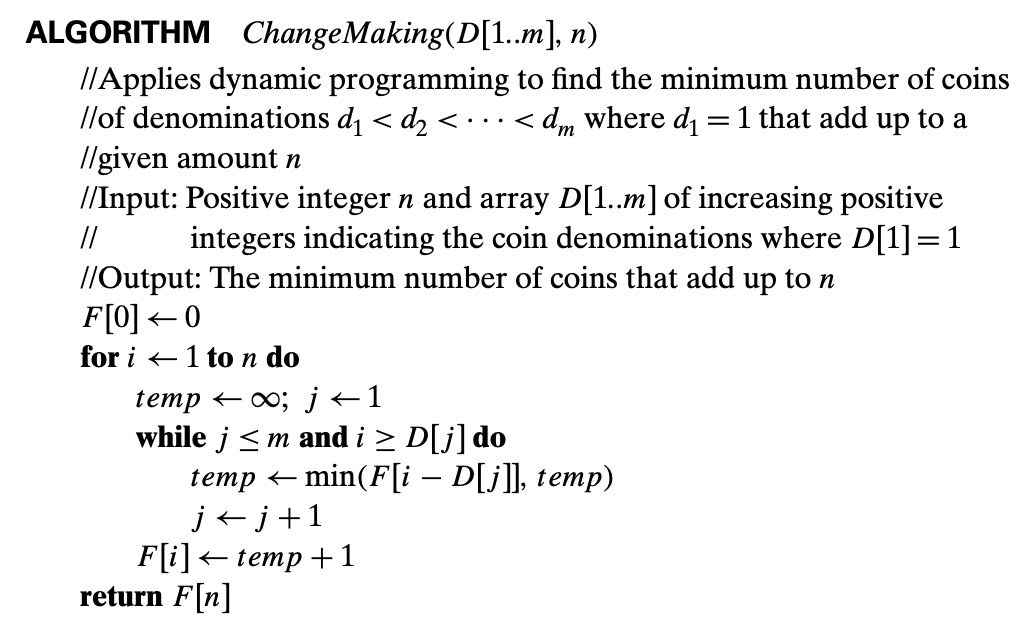




1. Change-making problem

Consider the general instance of the following well-known problem. Give change for amount n using the minimum number of coins of denominations d1 < d2 < . . . < dm. For the coin denominations used in the United States, as for those used in most if not all other countries, there is a very simple and efficient algorithm discussed in the next chapter. Here, we consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the m denominations d1 < d2 < . . . < dm where d1 = 1. Let F (n) be the minimum number of coins whose values add up to n; it is convenient to define F (0) = 0. The amount n can only be obtained by adding one coin of denomination dj to the amount n−dj for j =1,2,...,m such that n≥dj. Therefore, we can consider all such denominations and select the one minimizing F(n − dj) + 1. Since 1 is a constant, we can, of course, find the smallest F(n − dj) first and then add 1 to it. Hence, we have the following recurrence for F (n):





**Intermediate exercise**

1. Knapsack Problem

Given n items of known weights w1, . . . , wn and values v1, . . . , vn and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack. We assume here that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

Let F(i,j) be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

