Hoja 1: Ejercicio 2

TEMA

Diego Rodríguez Cubero

UCM

4 de febrero de 2026

Contenidos

Enunciado del problema

El rapero con sueltecillo

Un rapero lleva excesivo dinero suelto en el bolsillo de su pantalón, lo que contribuye a que lleve los pantalones demasiado caídos. Como consecuencia, cada vez que paga una cantidad C, su deseo es emplear el mayor número posible de monedas. Suponiendo que para todo $i, 1 \leq i \leq n$, tiene n_i monedas del tipo m_i, diseñar un algoritmo que le diga cuáles monedas debe emplear

Idea Principal

La idea principal del algoritmo es similar a la del problema de las monedas, pero en este caso, en lugar de minimizar el número de monedas, se busca maximizarlo.

Algoritmo propuesto

Llamemos monedas(i, j) = número máximo de monedas que se pueden usar para pagar j euros usando los primeros i tipos de monedas. Entonces, la recurrencia es:

$$\text{monedas}(i,j) = \max_{0 \leq k \leq n_i, k \cdot m_i \leq j} \{k + \text{monedas}(i-1, j - k \cdot m_i)\}$$

Y los casos base son:

$$\text{monedas}(0,j) = -\infty \quad \text{para todo } j \geq 0$$

y

$$\text{monedas}(i,0) = 0 \quad \text{para todo } i \geq 0$$

Pseudocódigo del algoritmo

Listing 1: Pseudocódigo del algoritmo propuesto

```
function max_monedas(n, m, N, C):
// n (tipos de monedas), m (valores de las monedas), N (número de
    monedas de cada tipo), C (cantidad a pagar)
crear matriz monedas[n+1][C+1]
para j desde 0 hasta C:
    monedas[0][j] = -infinity
para i desde 0 hasta n:
    monedas[i][0] = 0
para i desde 1 hasta n:
    para j desde 1 hasta C:
        max_monedas = -infinity
        para k desde 0 hasta N[i-1]:
            si k * m[i-1] <= j:
                max_monedas = max(max_monedas, k + monedas[i-1][j -
                    k * m[i-1]])
        monedas[i][j] = max_monedas
return monedas[n][C]
```

Reconstruccion de la solución

Listing 2: Reconstrucción de la solución

```
function reconstruir_solucion(monedas, n, m, N, C):
i = n
j = C
resultado = []
mientras i > 0 y j > 0:
    para k desde 0 hasta N[i-1] {
        si k * m[i-1] <= j // No nos pasamos de j
        // Lo optimo es usar k monedas del tipo i-1
        && monedas[i][j] == k + monedas[i-1][j - k * m[i-1]]:
                resultado.agregar((m[i-1], k))
                j = j - k * m[i-1]
                romper
    }
    i = i - 1
    si k == N[i-1] + 1: // No se usó ninguna moneda de este tipo
        resultado.agregar((m[i-1], 0))
return resultado
```

Complejidad del algoritmo

La complejidad temporal del algoritmo es $O(n \cdot C \cdot \text{máx}(N_i))$. Esto se debe a que para crear la solucion recorremos una matriz de tamaño $n \cdot C$ y para cada celda, en el peor caso, tendremos que iterar hasta N_i veces para encontrar el máximo número de monedas que se pueden usar. La reconstruccion de la solución tiene una complejidad de $O(n \cdot \text{máx}(N_i))$ en el peor caso, ya que en el peor caso, podríamos tener que revisar todas las posibles cantidades de monedas para cada tipo. La complejidad en espacio es $O(n \cdot C)$ debido a la matriz utilizada para almacenar los resultados intermedios.

Conclusión

Hemos diseñado un algoritmo para resolver el problema, basandonos al inicio en la idea del problema de las monedas, pero adaptándolo para maximizar el número de monedas utilizadas. La solución se basa en una programación dinámica que nos permite calcular el número máximo de monedas que se pueden usar para pagar una cantidad dada, teniendo en cuenta las restricciones de cantidad y tipos de monedas disponibles.