PROGRAMACIÓN DINÁMICA

- Método para transformar una definición recursiva en iterativa (más eficiencia) a través de la confección de una tabla de valores.
- Objetivo: evitar la reiteración de cómputos que usando una definición recursiva, aparecen varias veces.
- Observación: no todo problema de backtracking se puede pasar a programación dinámica.

La tabla comienza desde los casos bases.
Cuando hay dos parámetros, en lugar de hacer una tabla se hace una matriz.
En el caso de matrices, se debe prestar atención al orden en que se van llenando las filas y/o columnas.

Ejercicio 1:

1. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de la siguiente definición recursiva (backtracking):

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ \min_{q \in \{0,1,\ldots,j \div d_i\}} (q + cambio(i-1, j - q * d_i)) & j > 0 \wedge i > 0 \end{cases}$$

Notar que en la llamada recursiva, siempre se pasa a la siguiente denominación (i-1).
En esta versión, el algoritmo considera en una sola recursión todas las cantidades posibles de monedas de una denominación fija (incluso cantidad cero).
Llamada principal: cambio(n, k)

Ejemplo:
d_1 = 25
d_2 = 50
j = 200

cambio(2, 200) = min(q en 0,1,2,3,4)...

¿Qué forma tiene la tabla que voy a llenar?
Es una tabla de (n+1) filas x (k+1) columnas (se cuenta el 0, fijarse en los casos base).

¿En qué orden hay que llenar la tabla?
Las filas de arriba hacia abajo, las columnas no importa el orden.

¿Por qué? Para llenar cambio(i,j) debo mirar elementos en posiciones cambio(**i-1**, j- q*d_i), o sea siempre **en la fila arriba a la actual** (i-1).

Tipo (e implementación) de la función:

```
fun cambio(d: array[1..n] of nat, k: nat) ret r: nat
    var tabla: array[0..n, 0..k] of nat
```

```
     var mimin: nat

     {- primero, llenamos en la tabla los casos base -}
     for i:=0 to n do {- caso base j = 0, j está representado en las columnas
-}
          tabla[i, 0] := 0
     od
     for j:=1 to k do {- caso base j > 0, i = 0 -}
          tabla[0, j] := infinito
     od

     {- ahora, vamos llenando los casos recursivos teniendo en cuenta el
     orden en el que debemos llenarla -}
     for i:=1 to n do {- filas de arriba hacia abajo -}
          for j:=1 to k do
               {- Cálculo del mínimo con varios q posibles -}
               mimin := infinito
               for q:=0 to (j/d[i]) do
                    mimin := mimin ‘min’ (q + tabla[i-1, j- q*d[i]])
               od
               tabla[i, j] := mimin
          od
     od

     {- llamada principal (ahora, una vez completada la tabla, sabemos que
     tabla[i, j] = cambio(i, j) para todo i, j)-}
     r := tabla[n, k]
end fun
```

PARTE 2: Calculamos la solución (cuáles son las monedas que usamos) además de la cantidad mínima de monedas

IDEA: vamos a necesitar otra tabla paralela para ir guardando las soluciones para todos los i, j posibles. Tendremos:

tabla[i,j] = cambio(i,j)

solucion[i,j] = “lista de monedas correspondiente al problema que minimiza el cambio(i,j)”

```
fun cambio(d: array[1..n] of nat, k: nat) ret r: List of nat
     var tabla: array[0..n, 0..k] of nat
     var solucion: array[0..n, 0..k] of (List of nat)
     var mimin: nat

     {- primero, llenamos en la tabla los casos base -}
```

```
     for i:=0 to n do {- caso base j = 0, j está representado en las columnas
-}
          tabla[i, 0] := 0
          solucion[i, 0] := empty_list()
     od
     for j:=1 to k do {- caso base j > 0, i = 0 -}
          tabla[0, j] := infinito  {- no hay solución -}
     od

     {- ahora, vamos llenando los casos recursivos teniendo en cuenta el
     orden en el que debemos llenarla -}
     for i:=1 to n do {- filas de arriba hacia abajo -}
          for j:=1 to k do
               {- Cálculo del mínimo con varios q posibles -}
               mimin := infinito
               for q:=0 to (j/d[i]) do
                    if q + tabla[i-1, j-q*d[i]] < mimin then:
                         mimin := mimin ‘min’ (q + tabla[i-1, j-q*d[i]])
                         q_min := q
               od
               tabla[i, j] := mimin
               {- quiero poner q veces d[i] para el q elegido y además todos
               los elementos de solucion[i-1, k- q*d[i]] -}
               if mimin < inifinito then
                    solucion[i, j] := armar_solucion(q_min, d[i],
                                                    solucion[i-1,j-q_min*d[i]])
               fi
          od
     od
     if tabla[n, k] < infinito then:
          r := copy_list(solucion[n, k])
     fi
     {- LIBERO MEMORIA -}
     for i := 0 to n do
          for j := 0 to k do
               if tabla[i, j] < infinito then
                    destroy_list(solucion)
               fi
          od
     od
end fun

fun armar_solucion(q: nat, d: nat, sol_ant: List of nat) ret sol: List of nat
```

```
     {- copiamos la solución anterior y agregamos q monedas de denominación d
-}
     sol := copy_list(sol_nat)
     for i := 1 to q do
          addr(sol, d)
     od
end fun
```