✍🏾

Practico 3.4 - Programación Dinámica

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

$$
cambio(i,j) = \begin{cases} 0 & \text{si } j = 0 \\ \infty & \text{si } j > 0 \land i = 0 \\ min q \in (0,1,..j/di)(q + cambio(i-1, j - q * di) & \text{si } j > 0 \land i > 0 \end{cases}
$$

```
fun cambio(D: Array[1..n] of nat, j : nat) ret res Nat

 {- Cada celda de la matriz dp[i][k] representa la minima cantidad de
        monedas necesarias para alcanzar la cantidad "k" usando las
        primeras "i" monedas del arreglo "D" -}

    var dp[0..n, 0..j] of nat {- Mi matriz que representa cada estado -}

    var temp : nat

    {- Llenamos la tabla con los casos bases -}
    for i:= 0 to n do  {- Primeer caso base si j = 0 -}
        dp[i][0] = 0
    od

    for k:= 0 to j do  {- Segundo caso base si i = 0 -}
        dp[0][j] = INF
    od

    {- Llenamos la matriz dp -}
    for i:= 0 to n do {- Recorremos las monedas disponibles -}
        for k:= 0 to j do  {- Recorre la cantidad de dinero desde 1 hasta j -}
            temp := dp[i][k]
                for q:= 0 to j/D[i] do  {- Todas las posibles cantidad de monedas -}
                    temp = min(temp, temp + dp[i-1, j-q * D[i]])
                od
                dp[i][k] = temp
        od
    od

    {- Representa el minimo numero de monedas necesarias utilizando
         todas las monedas disponibles "n" para alcanzar la cantidad "j" -}
    res := dp[n, j]

end fun
```

> 2. Para el ejercicio anterior, ¿es posible completar la tabla de valores “de abajo hacia arriba”? ¿Y “de derecha a izquierda”?
>
> En caso afirmativo, reescribir el programa. En caso negativo, justificar.

Viendo que en el algoritmo anterior llenamos la tabla de dp de izquierda a derecha y de abajo hacia arriba, analicemos si lo podemos hacer al revés.

Para hacer de arriba hacia abajo es posible,

```
fun cambio(D: array [1...n]of Nat, j : Nat) ret res:Nat
    var dp : array[0...n,0...j] of Nat
    var temp : Nat

    {- Llenamos la tabla con los casos bases -}
    for i := 0 to n do
        dp[i,j] := 0
    od

    for k:=0 to j-1 do
        dp[n,k] := ∞
    od

    {- Llenamos la tabla de dp de arriba hacia abajo -}
    for i:= n-1 downto 0 do
        for k:=j-1 downto 0 do
            temp := ∞
            for q:=0 to j / D[i] do
            temp := min(temp, q + dp[i+1, j-q * D[i]])
            od
            dp[i,k] := temp
        od
od
    res := [0,0]
end fun
```

> 3. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de cada una de las siguientes definiciones recursivas (backtracking):

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ 1 + \min_{i' \in \{1,2,\dots,i \mid d_{i'} \leq j\}}(cambio(i', j - d_{i'})) & j > 0 \end{cases}$$

```
fun cambio(D : Array[1..n] of nat, k : nat) ret res : nat
    var dp : Array[1..n, 1..k]
    var min : nat
    {- Llenamos la tabla de dp con los casos bases -}
    for i:= 1 to n do
        dp[i, 0] = 0
    od
    for i:= 0 to to n do
        for j:= 1 to k do
            min := ∞
            for i':= 1 to n do
                if( D[i'] <= j && min > (dp[i′, j-D[i′] ])) then
                    min := (dp[i',j-D[i′]])
                else
                    skip
                fi
```

```
            dp[i,j] := min +1
            od
        od
    od
    res := dp[n, k]
end fun
```

$$cambio(i,j) = \begin{cases} 0 & \text{si } j = 0 \\ \infty & \text{si } j > 0 \wedge (i = n) \\ cambio(i+1, j) & \text{si } di > j > 0 \wedge i < n \\ min(cambio(i+1, j), 1 + cambio(i, j - di)) & \text{si } j \geq di > 0 \wedge i < n \end{cases}$$

```
fun cambio (D : Array[1..n] of nat, k : nat) ret res : nat
    var dp : array[1..n, 1..j] of nat

    {- Llenamos los casos bases -}
    {- j = 0 -}
    for i:= 1 to n do
        dp[i, 0] = 0
    od

    {- j > 0 && i = n -}
    for j:= 1 to n do
        dp[n, j] = ∞
    od

    {- Llenamos la tabla de dp -}
    for i:= 1 to n-1 do  {- Hacemos hasta n-1 ya que sino lo hacemos hasta n
        for j:= 1 to n do     al hacer i+1 nos vamos de rango -}
            if (D[i] > j) then
                dp[i+1, j] := dp[i, j] {- La moneda actual es de una denominacion
            else                        mayor que el total que yo quiero -}
                dp[i+1, j] = min(dp[i+1, j], 1 + dp[i, j- D[i]]) {- min entre usar o no usar
            fi
        od
    od

    res := dp[n, k]
end fun
```

> 5. Para cada una de las soluciones que propuso a los ejercicios del 3 al 9 del practico de backtracking, dar una definición alternativa que utilice la técnica de programación dinámica. En los casos de los ejercicios 3, 5 y 7 modificar luego el algoritmo para que no solo calcule el valor optimo sino que devuelva la solución que tiene dicho valor (por ejemplo, en el caso del ejercicio 3, cuales serian los pedidos que debería atenderse para alcanzar el máximo valor).

Ejercicio 3:

Link al ej3

$$maxImport(i,j) = \begin{cases} 0 & \text{si } i = 0 \vee j = 0 \\ maxImport(i-1, j) & \text{si } j < hi \wedge (i > 0 \wedge j > 0) \\ max(maxImport(i-1, j - hi) + mi, maxImport(i-1, j) & \text{si } j \geq hi \wedge (i > 0 \wedge j > 0) \end{cases}$$

```
type Product = tuple
                                    importe : nat
                                    harina : nat
                                end tuple

fun maxImport(Products : Array[1..n] of Product, j:nat) ret pedidos : List of nat
    var dp:Array[1..n, 1..j] of nat
    {- keep[i, k] = true si se anadio el pedido "i" fue anadido a la
         solucion cuando habia k harina. lo usaremos para reconstruir -}
    var keep : Array[1..n, 1..j] of bool
    {- Rellenamos la tabla de dp con el caso base -}
    for i:= 1 to n do
        for k:= 1 to j do
            dp[i, k] := 0
                keep[i, k] := false {-lo inicializmos con "false" ya que no hay pedidos-}
        od
    od

    for i:=1 to n do
        for k:= 1 to j do
            if (k<Products[i].harina) then
                dp[i, k] := dp[i-1, k]
            else
            {-vamos a hacer una leve modificacion para ver si se agrega o no el pedido "i" -}
            {- lo que hacemos es ver el maximo manualmente -}
                if dp[i-1, k-Products[i].harina] + Products[i].importe > dp[i-1, k] then
                    dp[i, k] := dp[i-1, k-Products[i].harina] + Products[i].importe
                    keep[i, k] := true {- si anadimos el pedido "i" porque es optimo
        else                              actualizamos la matriz keep a true -}
            dp[i, k] := dp[i-1, k]
            fi
        od
    od

    {- Reconstruimos y buscamos los pedidos que agregamos para la solucion optima -}
    var i, k : nat
    i:= n
    k := j  {- Cantidad de harina disponible total -}
    while(i>0) do
        if keep[i, k] then {- Si el producto fue incluido para la capacidad k, lo agrego -}
            addi(list, i)
            k := k - Products[i].harina
        if
        i := i - 1
    od

end fun
```