4. Para cada una de las soluciones que propuso a los ejercicios del 3 al 9 del práctico 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 sólo calcule el valor óptimo sino que devuelva la solución que tiene dicho valor (por ejemplo, en el caso del ejercicio 3, cuáles serían los pedidos que debería atenderse para alcanzar el máximo valor).

3. Una panadería recibe n pedidos por importes $m_1, \ldots, m_n$, pero sólo queda en depósito una cantidad H de harina en buen estado. Sabiendo que los pedidos requieren una cantidad $h_1, \ldots, h_n$ de harina (respectivamente), determinar el máximo importe que es posible obtener con la harina disponible.

La función recursiva obtenida con backtracking es:

```
panaderia(p,f) = ( si p = 0 ∨ f = 0  ⟶ 0
                 | si p > 0 ∧ f < hₚ ⟶ panaderia(p-1,f)
                 | si p > 0 ∧ f ≥ hₚ ⟶ panaderia(p-1,f) `max` mₚ + panaderia(p-1,f-hₚ)
                 )
```

donde *p* es el conjunto de pedidos que se hicieron a la panadería y *f* el total de harina disponible.

Siendo su definición en programación dinámica la siguiente:

```
fun panaderia(h: array[0..n] of nat, m: array[0..n] of nat, H: nat) ret solucion: nat
    var tabla: array[0..n,0..H] of nat   {- tabla[i,j] = panaderia(i,j) -}

    {- Caso 1 -}
    for i := 0 to n do
        tabla[i,0] := 0
    od

    for j := 0 to H do
        tabla[0,j] := 0
    od


   for i := 1 to n do
        for j := 1 to H do
            if h[i] > j then   {- Caso 2 -}
                tabla[i,j] := tabla[i-1,j]
            else    {- Caso 3 -}
                tabla[i,j] := tabla[i-1,j] `max` m[i] + tabla[i-1,j-h[i]]
            fi
        od
    od

    solucion := tabla[n,H]
end fun
```

¿Qué forma tiene la tabla? Es un arreglo de ***dos dimensiones***: *[0..n,0..H]*.
¿En qué orden se llena la tabla? Para llenar cada celda debo tener en cuenta:

`tabla[i-1,j-h[i]]`

- **i**: necesito ver la fila anterior (`i-1`) ⇒ ***n*** ⟶ ***0***
- **j**: necesito ver la columna anterior ⇒ ***H*** ⟶ ***0***

Otra versión que no solo calcula la ganancia máxima sino también cuáles son los pedidos más convenientes a atender es:

```
type Pedidos = tuple
                 id: nat
                 monto: nat
                 harina: nat
               end tuple

fun panaderia(h: array[0..n] of nat, m: array[0..n] of nat, H: nat)
              ret res: List of Pedidos

    var tabla: array[0..n,0..H] of nat   {- tabla[i,j] = panaderia(i,j) -}
    var solucion: array[0..n,0..H] of (List of Pedidos)
    var ganancia: nat
    var maximo: nat

    {- Caso 1 -}
    for i := 0 to n do
        tabla[i,0] := 0
        solucion[i,0] := empty_list()
    od

    for j := 0 to H do
        tabla[0,j] := 0
        solucion[0,j] := empty_list()
    od

    {- Caso 2 -}
    for i := 1 to n do
        for j := 1 to H do
            if h[i] > j then
                tabla[i,j] := tabla[i-1,j]
                solucion[i,j] := copy_list(solucion[i-1,j])
            else    {- Caso 3 -}
                maximo := tabla[i-1,j] `max` m[i] + tabla[i-1,j-h[i]]
                if maximo = tabla[i-1,j] then
                    solucion[i,j] := copy_list(solucion[i-1,j])
                else if maximo = m[i] + tabla[i-1,j-h[i]] then
                    solucion[i,j] := copy_list(solucion[i-1,j])
                    addr(solucion[i,j] , (i,m[i],h[i]))
                fi
            fi
        od
    od

    ganancia := tabla[n,H]
    res := solucion[n,H]
end fun
```