Ej1)

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

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

- Llamada principal:
  - Denominaciones d_1…. d_n
  - Dar cambio por un monto total k
  - cambio(n+1,k+1) (le pones el 0 más adelante)

- En esta versión, “cambio” es una sola recursión que considera todas las posibles monedas de una denominación fija (incluido cantidad cero)

  Ejemplo d_1 = 25, d_2 = 50
  cambio(2,200) = min(q en 0, 1, 2, 3, 4 = j / d_2)

- Tabla a llenar:
  Es una tabla n * k.
  Esta tabla se llena de arriba hacia abajo si o si, pq usamos el valor de i-1 siempre.
  Se puede llenar de izquierda a derecha o viceversa mientras sea de arriba a abajo.

Las filas corresponden a las monedas y su denominación
Las columnas corresponden a el valor del cambio a dar

```
fun cambio( d: array[1..n] of nat, k: Nat ) ret r: Nat
      {- variables -}
      var cam: array[0..n,0..k]
      var aux_min: nat
      {- casos base -}
      for i := 0 to n do cam[i,0] := 0 od
      for j := 0 to n do cam[0,j] := inf od
      {- casos recursivos -}
      for i := 1 to n do
            for j := 1 to k do
                  var aux_min := inf
                  for q := 0 to j / d[i] do
                        aux_min =   aux_min ‘min’ (q + cam[ i-1, j - q *
                  d[i] ] )
                  od
                  cam[i,j] := aux_min
            od
      od
      {- resultado: llamada principal -}
      r := cam[n,k]
```

```
end fun
```

Versión con cálculo de la solución:

```
fun cambio( d: array[1..n] of nat, k: Nat ) ret r: List of nat
      {- variables -}
      var cam: array[0..n,0..k] of nat
      var solucion:  array[0..n,0..k] of (List of nat)
      var aux_min: nat
      {- casos base -}
      for i := 0 to n do cam[i,0] := 0, solucion[i,0] := empty_list() od
      for j := 0 to n do cam[0,j] := inf, {- no hay solución -} od
      {- casos recursivos -}
      for i := 1 to n do
            for j := 1 to k do
                  var aux_min := inf
                  for q := 0 to j / d[i] do
                        if q + cam[i-1, j-q*d[i]] < aux_min then
                              aux_min := q + cam[i-1, j-q*d[i]]
                              q_min := q
                        fi
            od
                  cam[i,j] := aux_min
                  {- quiero poner q veces d[i] para el q elegido y además
                  todos los elementos de la solución[i-1,j-q*d[i]] -}
                  if aux_min < infinito then
                        solución[i,j] := armar_solucion(q_min, d[i],
                  solución[i-1, j-q*d[i]])
                  fi
            od
      od
      {- resultado: llamada principal cam[i,j] ≡ cambio[i,j] para todo
i,j-}
      {- y solución[i,j] tiene la solución correspondiente-}
      r := copy_list(solución[n,k]) {- si no hay solución, no hay nada -}

      for i := 0 to n do
            for j := 0 to k do
                  if tabla[i,j] < infinito then
                        destroy_list(solución)
                  fi
            od
      od
end fun
```

```
fun armar_solucion(q: nat, d: nat, sol_ant: List of nat)
        ret sol: List of nat
    {- copia la solución anterior y agregamos q monedas de denominación d (podria ser
q = 0) -}
    sol := copy_list(sol_ant)
    for i:= 1 to n do
        addr(sol, d)
    od
end fun
```

Ej2)

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.

No es posible completar los valores de abajo hacia arriba, para completar las posiciones del arreglo siempre estamos mirando la fila anterior, comenzando desde el caso base. Por lo tanto, como no tenemos las filas intermedias inicializadas esa forma no funcionaria. Lo que sí podemos hacer ir completando la tabla desde derecha a izquierda o viceversa, incluso con valores intermedios o raros de calcular.

De derecha a izquierda:

```
fun cambio( d: array[1..n] of nat, k: Nat ) ret r: Nat
    {- variables -}
    var cam: array[0..n,0..k]
    var aux_min: nat
    {- casos base -}
    for i := 0 to n do cam[i,0] := 0 od
    for j := 0 to n do cam[0,j] := inf od
    {- casos recursivos -}
    for i := 1 to n do
        for j := k to 1 do
            var aux_min := inf
            for q := 0 to j / d[i] do
                aux_min =   aux_min ‘min’ (q + cam[ i-1, j - q *
            d[i] ] )
            od
            cam[i,j] := aux_min
        od
    od
    {- resultado: llamada principal -}
    r := cam[n,k]
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):

(a)

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

(b)

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

Ej3a)

- Llamada principal:
  - Denominaciones d_1…. d_n
  - Dar cambio por un monto total k
  - cambio(n+1,k+1) (le pones el 0 más adelante)

- En esta versión, “cambio” tiene una sola recursión
  - si j > 0, entonces para todo i’ desde 1,2,....,i me quedo con la mínima denominación anterior a i tq se cumpla $d_{i'} \leq j$, es decir siempre que haya algo para dar cambio, vamos a utilizar las monedas con menor valor posible menor igual a j. No restamos la cantidad de monedas, sabemos que i’ es la moneda con menor denominación que podemos pagar, así que continuamos pagando con esa.
  - Ejemplo d_1 = 25, d_2 = 50 cambio(2,200) = min(i en 0, 1, 2) nos da 8 (usamos 8 veces la moneda de 25)
- Tabla a llenar:
  - Es una tabla n * k.
  - Esta tabla se llena de izquierda a derecha, ya que nuestro caso base es la columna 0
  - Se puede llenar desde abajo o desde arriba, no estamos mirando la anterior fila, si no que estamos viendo la fila con el i tq d[i] sea el mínimo que cumpla d[i] ≤ j

```
fun cambio( d: array[1..n] of nat, k: Nat ) ret r: Nat
      {- variables -}
      var cam: array[0..n,0..k]
      var aux_min: nat
      {- casos base -}
      for i := 1 to n do cam[i,0] := 0 od
      {- casos recursivos -}
      for i := 1 to n do
            for j := 1 to k do
                  var aux_min := inf
                  for i' := 1 to i do
                        if d[i] ≤ j then
                            aux_min =   aux_min 'min' 1 + cam[i', j - d_i']
                        fi
                  od
                  cam[i,j] := aux_min
            od
      od
      {- resultado: llamada principal -}
      r := cam[n,k]
end fun
```

i j->	0	1	2	3	4	5	6
1	0	inf	1	inf	2	inf	3
2	0						
3	0						

EL PRIMER CASO NO ES MUY DESCRIPTIVO DEL ALGORITMO, PQ I' SOLO PUEDE SER 1 PERO EN LOS DEMÁS, SIEMPRE VA LA MENOR CANTIDAD DE MONEDAS DE MENOR DENOMINACIÓN PARA EL VALOR J

tengo que pagar 6 pesos con 3 denominaciones d_1 = 2, d_2 = 3, d_3 = 6

para i = 1, j = 1 tenemos:
aux_min = inf
i' = 1
d[1] ≤ 1, no pues d_1 = 2
sale del bucle con aux_min = inf
`cam[1,1] = inf`

para i = 1, j = 2 tenemos:

aux_min = inf
i’ = 1
d[1] ≤ 2, si pues d_1 = 2
aux_min := inf ‘min’ 1 + cam[1,0]
aux_min := 1 + 0 = 1
sale del bucle con i’ = 2 ya que i = 1
cam[1,2] = 1

para i = 1, j = 3 tenemos:
aux_min = inf
i’ = 1
d[1] ≤ 3, si pues d_1 = 2
aux_min := inf ‘min’ 1 + cam[1,1]
aux_min := inf ‘min’ 1 + inf = 1
i’ = 2
cam[1,3] = inf

para i = 1, j = 4 tenemos:
aux_min = inf
i’ = 1
d[1] ≤ 4, si pues d_1 = 2
aux_min := inf ‘min’ 1 + cam[1,2]
aux_min := inf ‘min’ 1 + 1 = 2
i’ = 2
cam[1,4] = 2

para i = 1, j = 5 tenemos:
aux_min = inf
i’ = 1
d[1] ≤ 5, si pues d_1 = 2
aux_min := inf ‘min’ 1 + cam[1,3]
aux_min := inf ‘min’ 1 + inf = inf
i’ = 2
cam[1,5] = inf

para i = 1, j = 6 tenemos:
aux_min = inf
i’ = 1
d[1] ≤ 6, si pues d_1 = 2
aux_min := inf ‘min’ 1 + cam[1,4]
aux_min := inf ‘min’ 1 + 2 = 3
i’ = 2
cam[1,6] = 3

Ej3b)

(b)

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

- El algoritmo se basa en la idea general del algoritmo original solo que enves de recorrer la la “lista” de denominaciones de derecha a izquierda 0, …, n lo hace al revés, es decir, desde n, n-1, n-1,...,0
- Esto nos dice que la tabla se completa de derecha a izquierda y desde abajo hacia arriba, puesto que los casos bases están en i = n y j = 0

	0	1	2	3
0	0			
1	0			
2	0	inf	inf	inf

- La llamada principal es cambio(n,k) y su solución se encuentra en la tabla en la posición cam[0,k]
- La llamada recursiva es cambio(i,j)

```
fun cambio( d: array[1..n] of nat, k: Nat ) ret r: Nat
      {- variables -}
      var cam: array[0..n,0..k]
      var aux_min: nat
      {- casos base -}
      for i := 0 to n do cam[i,0] := 0 od
      for j := 1 to k do cam[n,j] := inf od
      {- casos recursivos -}
      for i := n-1 downto 0 do
            for j := 1 to k do
                  if d[i] > j then
                        cam[i,j] := cam[i+1,j]
                  else
                        cam[i,j] := min(cam[i+1,j], 1 + cam[i, j - d[i]])
                  fi
            od
      od
      {- resultado: llamada principal -}
      r := cam[0,k]
end fun
```

Ej4)

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 solo calcule el valor óptimo sino que devuelva la solución que tiene dicho valor (por ejemplo, en el caso del ejercicio 3, cuales serıan los pedidos que deberıa atenderse para alcanzar el máximo valor).

Ej3) tp3.3) PD

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

- El enunciado:
  - n pedidos con importes $m_1 , . . . , m_n$ y costos de harina $h_1 , . . . , h_n$
  - H es la harina que tenemos disponible para hacer pedidos
  - Determinar el máximo importe que es posible obtener con la harina disponible
- Parámetros:
  - **IMPORTANTE:** El algoritmo deberá obtener el **MÁXIMO** importe, pero no que pedidos hacemos
  - Definimos:
    - *pedidos(i,j) = “el máximo importe posible que puedo generar con la harina disponible haciendo algunos pedidos entre 1 e i, de manera tal que su costo de harina sea menor o igual a h”*
- Función recursiva:

  pedidos(n,H)

pedidos(i,j)	0	$i = 0 \vee j = 0$
	pedidos(i-1, j)	$h_i > j \wedge (j > 0 \wedge i > 0)$
	max(m_i + pedidos(i-1, j - h_i), pedidos(i-1, j))	$j \geq h_i \wedge (j > 0 \wedge i > 0)$

```
fun panaderia ( n: nat, H: nat, m: array[1..n] of nat, h: array[1..n] of nat ) ret result: nat
        {- variables -}
        var tabla: array[0..n, 0..H] of nat
        {- caso base -}
        for i := 0 to n do
                tabla[i, 0] := 0
        od
        for j := 0 to H do
                tabla[0, j] := 0
        od
        {- caso recursivo -}
        for i := 1 to n do
                for j := 1 to H do
                        if h[i] > j then
                                tablero[i,j] := tablero[i-1,j]
                        else
                                tablero[i,j] := max( m[i] + tablero[i-1, j - h[i]], tablero[i-1,j] )
                od
        od
        result := tablero[n,H]
end fun
```

Devolviendo que pedidos hago para llegar a ese valor:

```
fun panaderia ( n: nat, H: nat, m: array[1..n] of nat, h: array[1..n] of nat ) ret res: List of nat
        {- variables -}
        var tabla: array[0..n, 0..H] of nat
        var solucion: array[0..m, 0..H] of List of nat
        {- caso base -}
        for i := 0 to n do
                tabla[i, 0] := 0
                solución[i,0] := empty_list()
        od
        for j := 0 to H do
                tabla[0, j] := 0
                solución[i,0] := empty_list()
        od
        {- caso recursivo -}
        for i := 1 to n do
                for j := 1 to H do
                        if h[i] > j then
                                tablero[i,j] := tablero[i-1,j]
                                solución[i,j] := solución[i-1,j]
                        else if
                                if m[i] + tablero[i-1, j - h[i]] >  tablero[i-1,j] then
                                        tablero[i,j] := m[i] + tablero[i-1, j - h[i]]
                                        {- agrego por izq el pedido a la lista en solución[i,j] -}
                                        solución[i,j] := copy_list(solución[i-1, j - h[i]])
                                        addr(solución[i,j], i)
                                else
                                        tablero[i,j] :=  tablero[i-1,j]
                                        solución[i,j] := solución[i-1,j]
                                fi
                        fi
                od
        od
        res := solucion[n,H]
end fun
```

Ej4) tp3.3) PD

- El enunciado:
  - n objetos con pesos $p_1,p_2,....,p_n$ y valores $v_1,v_2,...,v_n$
  - P es el peso que debemos perder para que el globo no se caiga.
  - Determinar el MENOR valor total de los objetos que debo tirar para que el globo no se caiga.
- Identifiquemos qué parámetros toma la función recursiva, y qué calcula en función de esos parámetros. IMPORTANTE: el algoritmo deberá obtener el MENOR valor tirado, pero no qué objetos tiramos.

`globo(i,h) = “el menor valor posible del que debo desprenderme tirando algunos de los objetos entre 1 e i, de manera tal que su peso sea mayor o igual a h”`

si entendemos la función que queremos definir, digamos cuál es la “llamada” a esta función que resuelve el problema del ejercicio.

globo(i,h)	0	$h \leq 0$
	+inf	$h > 0 \wedge i = 0$
	`min( v_i + globo(i-1,h-p_i), globo(i-1,h))`	`h > 0 ^ i > 0`

Parámetros:

- La funcion debera tomar dos arreglos, p[1..n] que determinen el peso de los n objetos y v[1..n] que determinen el valor de los n objetos, P: nat que nos indique el peso a perder y por último retornaremos el menor valor posible del que debo desprenderme para que el globo no se caiga
- La tabla se completa desde arriba para abajo y de izquierda a derecha

- Prototipo:
  fun globo(p:array[1..n] **of** nat, v: array[1..n] **of** nat, P: nat) **ret** result: nat

```
fun globo( p:array[1..n] of nat, v: array[1..n] of nat, P: nat ) ret result: nat
        {- variables -}
        var tabla: array[0..n, 0..P] of nat

        {- casos base -}
        for i := 0 to n do
                tabla[i,0] := 0 {- para todos los objetos, si tengo que tirar peso 0 es 0 -}
        od
        for h := 1 to P do
                tabla[0, h] := inf {- no tengo objetos pero sí peso del cual desprenderme -}
        od
        {- caso recursivo -}
        for i := 1 to n do
                for h := 1 to P do
                        if h ≥ p[i] then
                                tabla[i,h] := min( v[ i ] + tabla[i-1,max(0, h- p[ i ])], tabla[i-1, h])
                        else
                                tabla[i,h] := min( v[ i ] + tabla[i-1,0], tabla[i-1, h])
                        fi
od
        od
        result := tabla[n,P]
end fun
```

	0	1	2	3	4	5	6	7
0	0	inf	inf	inf	inf	inf	inf	inf
1	0							
2	0							
3	0							

Ej5) tp3.3) PD

	0	si para todo i, $p_i < d$
teléfono(d)	telefono(d+1)	si no lo presto el dia d
	$max(m_i * (r_i + 1 - p_i) + teléfono(r_i + 1)$	si $p_i = d$

- Parámetros: Son n amigos así que necesitamos:
    - un arreglo m[1..n] que represente el dinero que cada uno de ellos ofrece por dia
    - un arreglo p[1..n] que represente el dia de partida de cada uno de ellos
    - un arreglo r[1..n] que represente el dia de regreso de cada uno de ellos
    - vamos a tomar el ultimo dia de partida del amigo que parte último p[n]
    - también tomamos como parámetro el último día del amigo que regresa último r[n]
    - entonces nuestros casos bases se encuentran en tabla[ultima_partida] a tabla[ultimo_regreso] son 0 pq no podemos generar dinero
    - el resultado se encuentra en tabla[1]

```
fun teléfono(m: array[1..n] of nat,  p: array[1..n] of nat, r: array[1..n] of nat, last_p, last_b: nat) ret : nat
    {- variables -}
    var solucion: array[0..last_b] of nat
    var tabla: array[0..last_b] of nat
    var aux: nat
    {- casos base -}
    {- desde el último día last_p + 1, sabemos que no podemos generar dinero hasta que regresa last_b -}
    for d := last_p+1 to last_b do
        tabla[d] = 0
    od
    {- casos recursivos -}
    for d := last_p downto 1 do
        aux := 0
        for i := 1 to n do
            if p[ i ] == d then
                ingreso_amigo := m[ i ] * ( r[ i ] + 1 - p[ i ] ) + tabla[r[ i ] + 1]
                if ingreso_amigo > aux then
                    aux := ingreso_amigo
                    solución[d] := i
            fi
        od
        tabla[d] := max(aux, tabla[d+1])) {- max entre prestarlo ahora o prestarlo mañana -}
    od
end fun
```

Ej6) tp3.3) PD

Un artesano utiliza materia prima de dos tipos: A y B. Dispone de una cantidad M A y M B de cada una de ellas. Tiene a su vez pedidos de fabricar n productos p1 , . . . , pn (uno de cada uno). Cada uno de ellos tiene un valor de venta v1 , . . . , vn y requiere para su elaboración cantidades a1 , . . . , an de materia prima de tipo A y b1 , . . . , bn de materia prima de tipo B. ¿Cuál es el mayor valor alcanzable con las cantidades de materia prima disponible?

- El enunciado:
  - n productos a fabricar p1 , . . . , pn
  - valor de venta de cada producto v1 , . . . , vn
  - requiere a1 , . . . , an de materia prima de tipo A
  - requiere b1 , . . . , bn de materia prima de tipo B
  - Determinar el MÁXIMO valor alcanzable con la materia prima disponible.
- Identifiquemos qué parámetros toma la función recursiva, y qué calcula en función de esos parámetros.

- Llamada función recursiva:

art(i , ma , mb) “ MÁXIMA ganancia obtenible fabricando i productos con ma materia prima tipo A y mb materia prima tipo B”

- Llamada función principal:

art(n,MA,MB) “MÁXIMA ganancia obtenible fabricando n productos con MA materia prima tipo A y MB materia prima tipo B”

- Función recursiva

art(i, ma, mb) =

0	, si i = 0
$\max(v_i + art(i - 1, ma - a_i, mb - b_i), art(i - 1, ma, mb))$	, si $i > 0 \wedge ma \geq a_i \wedge mb \geq b_i$
$art(i - 1, ma, mb)$	,si $i > 0 \wedge ma < a_i \vee mb < b_i$
	,si $ma \geq a_i \wedge b_i > mb \wedge i > 0$
	,sí $mb \geq b_i \wedge a_i > ma \wedge i > 0$

- Función con programación dinámica:
- Parámetros:
  - n productos

- p: array[1..n] of nat (arreglo de los n productos)
- v: array[1..n] of nat (valor de los n productos)
- MA y MB of nat (materia prima disponible)
- a: array[1..n] of nat (lo que cuesta hacer un producto de MA)
- b: array[1..n] of nat (lo que cuesta hacer un producto de MB)
- tabla[n,MA,MB] hacemos una tabla que contemple los n productos, con cada valor posible para MA Y MB

```
fun artesano(p: array[1..n] of nat, v: array[1..n] of nat, a: array[1..n] of nat, b: array[1..n] of nat, MA, MB, n: nat) ret max_value: nat
    {- variables -}
    var tabla: array[0..n,0..MA,0..MB] of nat
    {- casos base -}
    for j := 0 to MA do
        for k := 0 to MB do
            tabla[0,j,k] := 0  {- no tengo pedidos para fabricar -}
        od
    od
    {- caso recursivo -}
    for i := 1 to n do
        for j := 1 to MA do
            for k := 1 to MB do
              if j ≥ a[i] and k ≥ b[i] then
                 tabla[i,j,k] := max( v[i] + tabla[i-1, j - a[i], k - b[i]], tabla[i-1, j, k])
              else
                 tabla[i,j,k] := tabla[i-1, j, k]
              fi
            od
        od
    od
    max_value := tabla[n,MA,MB]
end fun
```

Ej7) tp3.3) PD

$$
moc(i,w1,w2) \begin{cases} 0 & \text{,si } (w1 = 0 \land w2 = 0) \lor i = 0 \\ moc(i-1,w1,w2) & \text{,si } p_i > w1,\ w2 > 0 \land i > 0 \\ \max(v_i + moc(i-1, w1, w2 - p_i), moc(i - 1, w1, w2)) & \text{,si } w2 \geq p_i > w1 \land i > 0 \\ \max(v_i + moc(i-1, w1 - p_i, w2), moc(i-1, w1, w2)) & \text{, si } w1 \geq p_i > w2 \land i > 0 \\ \max(v_i + moc(i-1, w1 - p_i, w2), v_i + moc(i-1, w1, w2 - p_i), moc(i-1, w1, w2)) & \text{, si } w1 \geq p_i \land w2 \geq p_i \land i > 0 \end{cases}
$$

¿Qué forma tiene la tabla que voy a llenar?

Va a ser un arreglo de 3 dimensiones: [0..n, 0..W1, 0..W2].

¿En qué orden la tengo que llenar? Los pesos (1ra dimensión) se deben llenar de abajo hacia arriba, ya que para calcular para el piso i siempre uso valores del piso anterior i-1.

Para las otras dos dimensiones no importa el orden ya que nunca voy a necesitar usar valores dentro de un mismo piso (siempre voy a mirar valores del piso de abajo).

Tipo de la función:
fun 2mochilas(v : array[1..n] of nat, w : array[1..n] of nat, W1, W2 : nat) ret r : nat

```
fun 2mochilas(v : array[1..n] of nat, w : array[1..n] of nat, W1, W2 : nat) ret r : nat
    {- variables -}
    var tabla: array[0..n, 0…W1, 0…,W2] of nat
    var decisión: array[0..n,0..W1, 0..W2] of nat
    var solucion: array[1..n] of nat
    var j_aux: nat
    var k_aux: nat

    {- casos base -}
    for j := 0 to w1 do
        for k := 0 to w2 do
            tabla[0,j,k] := 0
            decisión[0,j,k] := -1 {- no hay solución -}
        od
    od
    for i := 0 to n do
        tabla[i,0,0] := 0
        decisión[0,j,k] := -1 {- no hay solución -}
    od
    {- casos recursivos -}
    for i := 1 to n do
        for j := 0 to W1 do
            for k := 0 to W2 do
                if j == 0 and k == 0 then
                    skip
                else if w[ i ] > j and w[ i ] > k then
                    tabla[i,j,k] := tabla[i-1,j,k]
                    decisión[i,j,k] := 0 {- no se agregamos a ninguna -}
                else if  k ≥ w[ i ] and w[ i ] > j then
                    if v[i] + tabla[i-1,j,k - w[i]] > tabla[i-1,j,k] then
                        tabla[i,j,k] := v[i] + tabla[i-1,j,k - w[i]]
                        decisión[i,j,k] := 2 {- agregamos a mochila 2 -}
                    else
                        tabla[i,j,k] := tabla[i-1,j,k]
                        decisión[i,j,k] := 0 {- no se agrega a ninguna -}
                    fi
                else if j ≥ w[ i ] and w[ i ] > k then
                    if v[i] + tabla[i-1,j - w[i], k] > tabla[i-1,j,k] then
                        tabla[i,j,k] := v[i] + tabla[i-1,j - w[i], k]
                        decisión[i,j,k] := 1 {- agregamos a mochila 1 -}
                    else
                        tabla[i,j,k] :=  tabla[i-1,j,k]
```

```
                                        decisión[i,j,k] := 0 {- no se agrega a ninguna -}
                                fi
                        else if j ≥ w[ i ] and k ≥ w[ i ] then
                                if v[i] + tabla(i-1, j - w[i] , k) >  v[i] + tabla(i-1, j, k - w[i] )
                        and v[i] + tabla(i-1, j - w[i] , k) > tabla( i-1, j, k) then
                                        tabla[i,j,k] :=   v[i] + tabla(i-1, j - w[i] , k)
                                        decisión[i,j,k] := 1 {- agregamos a mochila 1 -}
                                else if  v[i] + tabla(i-1, j, k - w[i] ) >  tabla( i-1, j, k) then
                                        tabla[i,j,k] :=   v[i] + tabla(i-1, j, k - w[i] )
                                        decisión[i,j,k] := 2 {- agregamos a mochila 2 -}
                                else
                                        tabla[i,j,k] :=  tabla( i-1, j, k)
                                        decisión[i,j,k] := 0 {- no se agrega a ninguna -}
                                fi
                        od
                od
        od
        {- reconstruir la solución -}
        for i := 1 to n do
                solución[i] := 0 {- inicializamos la solución en 0 -}
        od
        j_aux := W1
        k_aux := W2
        for i := n downto 1 do
                if decision[i,j_aux,k_aux] == 1 then
                        solución[i] := 1
                        j_aux = j_aux - w[i]
                else if decision[i,j_aux,k_aux] == 2 then
                        solución[i] := 2
                        k_aux = k_aux - w[i]
                fi
        od
end fun
```

Ej8) tp3.3) PD

Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene n estaciones de trabajo, $S_{1,1}$, . . . , $S_{1,n}$ para la primera y $S_{2,1}$, . . . , $S_{2,n}$ para la segunda. Dos estaciones $S_{1,i}$ y $S_{2,i}$ (para i = 1, . . . , n), hacen el mismo trabajo, pero lo hacen con costos $a_{1,i}$ y $a_{2,i}$ respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por n estaciones de trabajo $S_{i1\ ,1}$, $S_{i2\ ,2}$ . . . , $S_{in\ ,n}$ no

necesariamente todas de la misma línea de montaje (i_k = 1, 2). Si el automóvil está en la estación $S_{i,j}$, transferirlo a la otra línea de montaje (es decir continuar en $S_{i',j+1}$ con i' != i) cuesta $t_{i,j}$. Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.

- El enunciado:
  - 2 líneas de ensamblaje
  - n estaciones de trabajo por cada línea
  - $S_{1,1}$, . . . , $S_{1,n}$ para la primera y $S_{2,1}$, . . . , $S_{2,n}$ para la segunda
  - hacen el mismo trabajo pero con costos $a_{1,i}$ y $a_{2,i}$
  - Para fabricar un auto pasamos por n estaciones, no deben ser de la misma línea de ensamblaje
  - Pero cambiar de línea de ensamblaje cuesta $t_{i,j}$
  - Encontrar el costo mínimo usando ambas líneas
- Parámetros:
  - La función tomará las n estaciones por las cuales debe pasar el auto para ser ensamblado.
  - Tenemos la opción de hacerlo en una de dos líneas
  - Llamada recursiva:
    - car(i , $S_{j,i}$)
    - Costo mínimo de fabricar un auto en i estaciones en $S_{j,i}$ línea de ensamblaje
  - Llamada principal
    - car(n , $S_{j,i}$)
    - Costo mínimo de fabricar un auto en n estaciones en $S_{j,n}$ línea de ensamblaje
- Función matemática:

  Casos: ya no tengo estaciones de trabajo

  tengo estaciones de trabajo:

  yo tengo el auto en $S_{1,i}$

  si $a_{2,i} + t_{i,2} < a_{1,i}$ (debería cambiar a $S_{2,i}$)

  yo tengo el auto en $S_{2,i}$

  si $a_{1,i} + t_{i,1} < a_{2,i}$ (debería cambiar a $S_{1,i}$)

  yo tengo el auto en $S_{1,i}$

  si $a_{2,i} + t_{i,2} \geq a_{1,i}$ (debería quedarme en $S_{1,i}$)

  yo tengo el auto en $S_{2,i}$

  si $a_{1,i} + t_{i,1} \geq a_{2,i}$ (debería quedarme en $S_{2,i}$)

car(i , $S_{j,i}$) =

| 0 , si i = 0 v $S_{j,i} = 0$

| min($a_{2,i} + t_{i,2}$ + car(i - 1, $S_{2,i-1}$), $a_{1,i}$ + car(i - 1, $S_{j,i-1}$)) , si i > 0 ^ $a_{2,i} + t_{i,2} < a_{1,i}$

| min($a_{1,i} + t_{i,1}$ + car(i - 1, $S_{1,i-1}$), $a_{2,i}$ + car(i - 1, $S_{j,i-1}$))

$$
\begin{cases}
, \text{si } i > 0 \wedge a_{1,i} + t_{i,1} < a_{2,i} \\
\quad \min(a_{1,i} + car(i-1, S_{1,i-1}), car(i-1, S_{1,i-1})) \\
, \text{si } i > 0, j = 1, a_{2,i} + t_{i,2} \geq a_{1,i} \\
\quad \min(a_{2,i} + car(i-1, S_{2,i-1}), car(i-1, S_{2,i-1})) \\
, \text{si } i > 0, j = 2, a_{1,i} + t_{i,1} \geq a_{2,i}
\end{cases}
$$

Programación dinámica:

- Parámetros que toma la función:
  - lineas: array[1,2, 1..n] of nat (n estaciones de trabajo para S1 y S2)
  - costos: array[1..2,1..n] of nat (costos de S1 y S2 respecto al n)
  - transf: array[1,2, 1..n] of nat (costos de cambiar de 1 a 2 respecto al n)
  - Encontrar el costo mínimo usando ambas líneas
  - devolvemos un r: nat como costo mínimo
- Prototipo:

fun autos(lineas: array[1,2, 1..n] **of** nat, costos: array[1..2,1..n] **of** nat, transf: array[1,2, 1..n] **of** nat) **ret** resultado : nat

- Funcion:

```
fun autos( n: nat, a: array[1..2,1..n] of nat, t: array[1..n,1..n] of nat ) ret resultado : nat
    {- variables -}
    línea[1..2, 1..n] of nat
    {- casos bases -}
    linea[1,1] := a[1,1]
    línea[2,1] := a[2,1]
    {- casos recursivos -}
    for i := 2 to n do
        {- estando en la primer línea -}
        línea[1,i] := min(linea[1, i-1] + a[1,i], línea[2,i-1] + t[2,i] + a[2,i] )
        {- estando en la segunda línea -}
        línea[2,i] := min(línea[2, i-1] + a[2,i], línea[1,i-1] + t[1,i] + a[1,i] )
    od
    resultado := min(linea[1,n]m línea[2,n])
end fun
```

Ej9) tp3.3) PD

- Enunciado:
  - La ficha puede moverse a:
    - La casilla que está inmediatamente arriba,
    - La casilla que está arriba y a la izquierda (si la ficha no está en la columna extrema izquierda)

        - La casilla que está arriba y a la derecha (si la ficha no está en la columna extrema derecha)
    - Cada casilla tiene asociado un número entero cij (i, j = 1, . . . , n) que indica el puntaje a asignar cuando la ficha esté en la casilla.
    - El puntaje final se obtiene sumando el puntaje de todas las casillas recorridas por la ficha
    - Determinar el máximo y el mínimo puntaje que se puede obtener en el juego
- Parámetros:
    - La función deberá tomar la casilla en la cual está parado
    - Llamada función recursiva:
        - up_min($T_{i,j}$) nos da el mínimo valor desde $T_{n,j'}$ hasta $T_{i,j}$
        - up_max($T_{i,j}$) nos da el máximo valor desde $T_{n,j'}$ hasta $T_{i,j}$
    - Llamada función principal:
        - up_min($T_{n,j}$) nos da el mínimo valor desde $T_{n,j'}$ hasta $T_{1,j}$
        - up_max($T_{n,j}$) nos da el máximo valor desde $T_{n,j'}$ hasta $T_{1,j}$
- Función matemática:
  Casos:
    La ficha ya esta arriba ($T_{1,j}$) i = 1
    La ficha está en la columna extrema izquierda k = 1
    La ficha está en la columna extrema derecha k = n
    La ficha no está en ninguna columna extrema y no estamos arriba i != 1

$$
\begin{aligned}
up\_min(T_{i,k}) \;|\; & 0 && \text{, si } i = 0\\
|\; & \min(c_{i,k} + up\_min(T_{i-1,k}),\ c_{i,k} + up\_min(T_{i-1,k+1})) && \text{, si } k = 1,\ i \neq 1\\
|\; & \min(c_{i,k} + up\_min(T_{i-1,k}),\ c_{i,k} + up\_min(T_{i-1,k-1})) && \text{, si } k = n,\ i \neq 1\\
|\; & \min(c_{i,k} + up\_min(T_{i-1,k}),\ c_{i,k} + up\_min(T_{i-1,k-1}),\ c_{i,k} + up\_min(T_{i-1,k+1})) && \text{, si } k \neq 1,n
\end{aligned}
$$

$$
\begin{aligned}
up\_max(T_{i,k}) \;|\; & 0 && \text{, si } i = 0\\
|\; & \max(c_{i,k} + up\_max(T_{i-1,k}),\ c_{i,k} + up\_max(T_{i-1,k+1})) && \text{, si } k = 1,\ i \neq 1\\
|\; & \max(c_{i,k} + up\_max(T_{i-1,k}),\ c_{i,k} + up\_max(T_{i-1,k-1})) && \text{, si } k = n,\ i \neq 1\\
|\; & \max(c_{i,k} + up\_max(T_{i-1,k}),\ c_{i,k} + up\_max(T_{i-1,k-1}),\ c_{i,k} + up\_max(T_{i-1,k+1})) && \text{, si } k \neq 1,n
\end{aligned}
$$

```
fun min_up( valor_casilla: array[1..n,1..n] of nat ) ret min_valor: nat
    {- variables -}
    var tablero[1..n, 1..n]
    {- caso base -}
    for j := 1 to n do
        tablero[n,j] = valor_casilla[n,j]
```

```
        od
        {- caso recursivo -}
        for i := n-1 downto 1 do
                for k := 1 to n do
                        if k == 1 then
                                tablero[i,k] := valor_casilla[i,k] + min(tablero[i+1, k],
                                tablero[i+1,k+1])
                        else if k == n then
                                tablero[i, k] = valor casilla[i,k] + min(tablero[i+1,k], tablero[i+1,
                                k-1])
                        else
                                tablero[i, k] = valor_casilla[i, k] + min(tablero[i+1, k],
                                tablero[i+1, k-1], tablero[i+1, k+1])
                        fi
                od
        od
        {- calculamos el mínimo de la fila superior -}
        min_aux := tablero[1,1]
        for k := 2 to n do
                if tablero[1,k] < min_aux then
                        min_aux := tablero[1,k]
                fi
        od
        result := min_aux
end fun
```

Para up_max es lo mismo solo que cambiando todo min por max xd