BACKTRACKING

Recordemos que en algunos casos, el algoritmo voraz para resolver un determinado problema, no encontraba la solución óptima.

¿Qué hacemos entonces cuando no tenemos un algoritmo voraz que encuentre la solución óptima? Usamos backtracking.

El backtracking es una técnica de programación que consiste en:

Elige un candidato para agregar al conjunto solución, se fija si por ese camino obtiene la solución óptima y probablemente en algún momento tenga que volver hacia atrás en esa decisión y preguntarse que hubiera pasado si elegir este candidato, elegía este otro?

Los algoritmos de backtracking intentan todas las formas posibles de llegar a la solución y se quedan con la que sea óptima. Por esta razón, al backtracking se le suele decir “fuerza bruta”.

OBSERVACIONES:

- Siempre que haya una solución óptima, el backtracking la va a encontrar.
- Claramente, en general son algoritmos ineficientes.
- Los algoritmos de backtracking son recursivos.
- Como lo único interesante de estos algoritmos es la función recursiva, podemos definir la función matemáticamente (es la notación que vamos a usar nosotros).
- En todos los algoritmos de backtracking, en un momento yo tengo que tomar una decisión. Por ejemplo, en el caso de la moneda que está a continuación, esta decisión está en el hecho de usar o no la denominación i-ésima.

Problema de la moneda con backtracking:

cambio(i, j) = “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones $d_1, d_2, \dots, d_i$”

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

Traduciendo a palabras, cada uno de los casos:

- Si j = 0, es decir si tenemos que pagar el monto 0, entonces es claro que necesitamos 0 monedas.
- Si j > 0 e i = 0, es decir si tenemos que pagar un monto mayor a cero pero no tenemos ninguna denominación, entonces no tengo forma de pagar ese monto, y entonces devuelvo infinito (representando que no hay una solución).

- Si $d_i > j$, $j > 0$ e $i > 0$, es decir si tengo que pagar un monto mayor a 0, tengo denominaciones pero la denominación i-ésima se pasa del monto que quiero pagar, entonces esa denominación no me sirve, y por lo tanto llamo a la función cambio pero sin esa denominación.
- Por último, si $d_i <= j$, $j > 0$ e $i > 0$, es decir si tengo que pagar un monto mayor a 0, tengo denominaciones y la denominación i-ésima es menor o igual al monto que quiero pagar, es decir que me sirve, entonces en ese caso hago el backtracking propiamente dicho:
    - cambio(i - 1, j): pruebo encontrar la solución sin utilizar esa denominación
    - 1 + cambio(i, j - d_i): pruebo encontrar la solución usando esa denominación, lo que significa que ya usé una moneda (de ahí el +1) y ahora debo pagar el monto j - d_i, que es el valor de la denominación, y sigo considerando todas las denominaciones (cambio(i, j - d_i)).

  Luego, para encontrar la solución óptima al problema, se calcula el mínimo entre estas “dos” opciones.

Ejercicio 1 (revisar):

Modifique el código del algoritmo que resuelve el problema de la moneda utilizando backtracking, de manera que devuelva qué monedas se utilizan, en vez de solo la cantidad.

Algoritmo original (devuelve en r la cantidad de monedas que se utilizan):

```
fun cambio (d: array[1..n] of nat, i, j: nat) ret r: nat
     if j = 0 then
          r:= 0
     else if i = 0 then
          r:=∞
     else if d[i] > j then
          r := cambio(d,i-1,j)
     else
          r := min(cambio(d, i-1, j), 1 + cambio(d, i, j - d[i]))
     fi
end fun
```

Algoritmo modificado (devuelve un conjunto cuyos elementos son las denominaciones de las monedas que se utilizan) :

```
fun cambio (d: array[1..n] of nat, i, j: nat) ret S: Set of nat
     var S1, S2: Set of nat
     S := empty_set()
     S1 := empty_set()
     S2 := empty_set()
     if j = 0 then
          S := empty_set()
     else if i = 0 then
          S :=
```

```
    else if d[i] > j then
         S := cambio(d,i-1,j)
    else
         S1 := cambio(d, i-1, j)

    fi
end fun
```

Ejercicio 2:

En un extraño país las denominaciones de la moneda son 15, 23 y 29, un turista quiere comprar un recuerdo pero también quiere conservar el mayor número de monedas posible. Los recuerdos cuestan 68, 74, 75, 83, 88 y 89. Asumiendo que tiene suficientes monedas para comprar cualquiera de ellos, ¿cuál de ellos elegirá? ¿Qué monedas utilizará para pagarlo? Justificar claramente y mencionar el método utilizado.

Antes de empezar, notemos lo siguiente:

"Quiere conservar el mayor número de monedas posible" es equivalente a decir que quiere gastar el menor número de monedas posible.

La cantidad mínima de monedas que necesita para comprarse algo es 3:

Con 3 de 15, no compro nada.
Con 3 de 23, me puedo comprar solo el de 68.
Con 3 de 29, me puedo comprar alguno de los que salen 68,74,75 y 83.
Con 2 de 15 y 1 de 23, no me puedo comprar nada.
Con 2 de 15 y 1 de 29, no me puedo comprar nada.
(...)

Ejercicio 3:

Una panadería recibe n pedidos por importes $m_1, \ldots, m_n$, pero solo 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.

1. Analizo el enunciado:
   - Tenemos n pedidos con importes $m_1, \ldots, m_n$ y harina necesaria para hacerlos $h_1, \ldots, h_n$, respectivamente.
   - Tenemos una cantidad H de harina disponible para hacer pedidos.

   Se desea determinar el MÁXIMO IMPORTE que es posible obtener con la harina disponible. Luego, es claro que se debe calcular un máximo.

2. Identifiquemos los parámetros de la función recursiva y qué calcula en función de estos:

IMPORTANTE: el algoritmo deberá obtener el máximo importe posible de obtener pero no qué pedidos se hacen.

`panaderia(i, har) = “el máximo importe posible que puedo obtener cocinando algunos de los pedidos entre 1 e i de manera tal que la suma de las harinas necesarias para hacer cada uno de los pedidos sea menor o igual a har”`

Luego, digamos cuál es la “llamada” a esta función que resuelve el problema del ejercicio:

```
panes(n, H)
```

3. Defino la función recursiva panaderia:

Siendo i el índice que indica hasta qué pedido voy a considerar y har la harina aún disponible:

	CASOS	**SOLUCIÓN EN CADA CASO**
panaderia (i,har)	i = 0	Es decir, en el caso en el que aún tengo harina disponible para cocinar, pero no tengo pedidos para cocinar. Como no tengo pedidos para cocinar, claramente el máximo importe posible que puedo obtener cocinando nada es 0. **0**
	i > 0, har > 0, h_i > har	Caso en el que quedan pedidos por cocinar (i > 0), hay harina disponible (har > 0) y la harina necesaria para cocinar el pedido i es mayor a la harina disponible (hi > har). En este caso, descarto cocinar el producto i (no me alcanza la harina para cocinarlo): **panaderia(i-1, har)**
	i > 0, har > 0, h_i <= har	Caso en el que quedan pedidos por cocinar (i > 0), hay harina disponible (har > 0) y la harina necesaria para cocinar el pedido i es menor o igual a la harina disponible (h_i <= har): *{- probemos cocinar el pedido i -}* m_i + panaderia(i-1, har-h_i) *{- probemos NO cocinar el pedido i -}* panaderia(i-1, har) *{- debemos hallar el MÁXIMO entre estos dos -}* **max(m_i + panaderia(i-1, har-h_i), globo(i-1, har))**

Ejercicio 4

Usted se encuentra en un globo aerostático sobrevolando el océano cuando descubre que empieza a perder altura porque la lona está levemente dañada. Tiene consigo n objetos cuyos pesos $p_1, \dots, p_n$ y valores $v_1, \dots, v_n$ conoce. Si se desprende de al menos P kilogramos logrará recuperar altura y llegar a tierra firme, y afortunadamente la suma de los pesos de los objetos supera holgadamente P. ¿Cuál es el menor valor total de los objetos que necesita arrojar para llegar sano y salvo a la costa?

1. Entendiendo el enunciado:
   - Tenemos n objetos con pesos $p_1, \dots, p_n$ y valores $v_1, \dots, v_n$.
   - Tenemos P kilogramos que debemos perder para que el globo no se caiga.
   - La suma de los pesos de los objetos supera holgadamente a P.

   Se desea determinar el **MENOR VALOR total de los objetos** que debo tirar para que el globo no se caiga. Claramente, hay que calcular un mínimo.

2. Pensemos un ejemplo:
   4 objetos:
   $p_1 = 3, v_1 = 10$
   $p_2 = 5, v_2 = 8$
   $p_3 = 2, v_3 = 9$
   $p_4 = 3, v_4 = 6$

   P = 7

¿Cómo resolvemos el problema usando backtracking? Tengo que probar TODAS las combinaciones:

- ➔ Tirar todo. Pierdo los 13 kilos de peso, y 33 “puntos” de valor.
- ➔ No tirar nada. Pierdo 0 kilos de peso, 0 “puntos” de valor, pero el globo se cae.
- ➔ (... varias combinaciones más que no son solución porque el globo se cae …)
- ➔ Tirar 1 y 2. Pierdo 8 kilos, y 18 “puntos” de valor.
- ➔ Tirar 1, 2, y 3. Pierdo 10 kilos, y 27 “puntos” de valor.
- ➔ Tirar 1, 3 y 4. Pierdo 8 kilos, y 25 “puntos” de valor.
- ➔ Tirar 2 y 3. Pierdo 7 kilos, 17 “puntos” de valor.
- ➔ Tirar 2, 3 y 4. Pierdo 10 kilos, 23 “puntos” de valor.
- ➔ Tirar 2 y 4. Pierdo 8 kilos, 14 “puntos” de valor.

Backtracking prueba TODAS las posibilidades y se queda con la que cumple con el criterio óptimo, en este caso, con la opción que tiene valor de pérdida mínimo. O sea, en este caso:

Tirar 2 y 4, perdiendo **14 puntos de valor**.

3. Identifiquemos los parámetros de la función recursiva y qué calcula en función de estos:
   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 la suma de sus pesos sea mayor o igual que h”

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

```
globo(n,P)
```

4. Definamos la función `globo`:

Siendo i el índice que indica hasta qué objeto voy a considerar y h el peso del que todavía me debo desprender:

	CASOS	**SOLUCIÓN EN CADA CASO**
globo(i,h)	h<=0	**0** (no me hace falta tirar ningún objeto, y por lo tanto no pierdo valor)
	i = 0, h > 0	Cuando todavía tengo que desprenderme de cierto peso (mayor a cero) pero no tengo más objetos de los cuales desprenderme. Caso en el que no se resuelve el problema. Por lo tanto, si se llega a este caso no debería influir en la solución, que es equivalente a decir que este caso debe dar un resultado que pierda siempre en el min. Luego, en este caso, la solución es: **+∞**
	i > 0, h > 0	*{- probemos tirar el objeto -}* v_i + globo(i-1, h-p_i) *{- probemos NO tirar el objeto -}* globo(i-1, h) *{- debemos hallar el mínimo entre estos dos -}* **min(v_i + globo(i-1, h-p_i), globo(i-1, h))**

Ejemplo:
Sean los objetos:

Objeto 1: $p_1 = 2$, v_1
Objeto 2: $p_2 = 4$, v_2
Objeto 3: $p_3 = 10$, v_3

Ejecución paso a paso:

globo(3,8)

= {caso recursivo}

min(v_3 + **globo(2, -2)**, globo(2, 8))

={caso base 1}

min(v_3, **globo(2, 8)**)

={caso recursivo}

min(v_3, min(v_2 + **globo(1, 4)**, globo(1, 8)))

={caso recursivo}

min(v_3, min(v_2 + **min(v_1 + globo(0, 2), globo(0, 4))**, globo(1, 8)))

={caso base 2 x2 y aritmética y calculo el min}

min(v_3, min(v_2 + **+infinito**, globo(1, 8)))

={aritmética y calculo el min}

min(v_3, **globo(1, 8)**)

={p_1 es 2, nunca voy a llegar a tirar 8}

min(v_3, +infinito)

={resuelvo min}

v_3

Ejercicio 5:

Sus amigos quedaron encantados con el teléfono satelital, para las próximas vacaciones ofrecen pagarle un alquiler por él. Además del día de partida y de regreso (p_i y r_i) cada amigo ofrece un monto m_i por día. Determinar el máximo valor alcanzable alquilando el teléfono.

Aquí, a diferencia del ejercicio del Práctico 3.1, debo probar todas las combinaciones posibles.

1. Entender el enunciado:
   - Tenemos n amigos, de los cuales conocemos sus días de partida y de regreso (p_i y r_i respectivamente) y el monto m_i por día que cada uno ofrece para alquilar el teléfono.

   Se desea determinar el MÁXIMO VALOR que se puede alcanzar (la mayor cantidad de plata que se puede obtener) alquilando el teléfono.

2. Identifiquemos los parámetros de la función recursiva y qué calcula en función de estos:

`rataColuda(d) = “máxima plata que puedo ganar prestando el teléfono a algunos amigos a partir del día d hasta el último día”`

Luego, digamos cuál es la “llamada” a esta función que resuelve el problema del ejercicio (llamada principal):

`rataColuda(0)` (recordar que los días están representados por números naturales)

3. Definición recursiva de la función:

	CASOS	**SOLUCIÓN EN CADA CASO**
rataColuda(d)	Para todo i, $p_i < d$	Caso en el que el día de partida de todos los amigos es antes del día d. Es decir, todos los amigos ya se fueron y no me queda nadie a quien prestárselo. Luego, no puedo alquilarle el teléfono a nadie y por lo tanto la máxima plata que puedo ganar es 0. **0**
		{- No lo alquilo el día d -} rataColuda(d+1) {-Se lo alquilo al amigo i, cuyo día de partida es d (p_i = d)-} m_i * (r_i - p_i) + rataColuda(r_i + 1) {- Notar que m_i * (r_i - p_i) es la plata que voy a ganar alquilándoselo al amigo i. r_i - p_i es la cantidad de días que se va el amigo i -} Debo calcular el máximo entre estos dos: **max(rataColuda(d+1), m_i * (r_i - p_i + 1) + rataColuda(r_i + 1))**

4. Ejemplo: n = 6

p_1=1, r_1=2, m_1=3
p_2= 3, r_2= 4, m_2= 2
p_3= 4, r_3= 7, m_3= 5
p_4= 2, r_4= 8, m_4= 6
p_5= 6, r_5= 8, m_5= 8
p_6= 4, r_6= 8, m_6= 5

rataColuda(0) = rataColuda(1) {- no se va nadie el día 0, está rataColuda(1) max 0 implícito -}
= **42 ← SOLUCIÓN FINAL**

---

rataColuda(1) = rataColuda(2) {- No lo alquilo -}
max
(3*2 + rataColuda(3)) {- Se lo alquilo al amigo 1 (es el que se va el día 1) -}
= 42 max (6 + 28)
= 42 max 34
= 42

---

rataColuda(2) = rataColuda(3)
max
(6*7 + rataColuda(9)) {- Se lo alquilo al amigo 4 (que se va el día 2) -}
= 28 max (42 + 0)
= 28 max 42
= 42

---

rataColuda(3) = rataColuda(4)
max
(2*2 + rataColuda(5)) {- Se lo alquilo al amigo 2 (se va el día 3) -}
= 25 max (4 + 24)
= 25 max 28
= 28

---

rataColuda(4) = rataColuda(5)
max
(5*4 + rataColuda(8) max {- amigo 3 -}
(5*5 + rataColuda(9))
= 24 max ((20 + 0) max (25 + 0))

= 24 max (20 max 25)
= 24 max 25
= 25

---

rataColuda(5) = rataColuda(6) {- no se va nadie el día 5 -}
= 24

---

rataColuda(6) = rataColuda(7)
max
(8*3 + rataColuda(9))
= 0 max (24 + 0)
= 0 max 24 = 24

---

rataColuda(7) = 0
rataColuda(8) = 0
rataColuda(9) = 0

Ahora, resuelvo hacia arriba.

Ejercicio 6:

Un artesano utiliza materia prima de dos tipos: A y B. Dispone de una cantidad MA y MB de cada una de ellas. Tiene a su vez pedidos de fabricar n productos $p_1, \ldots, p_n$ (uno de cada uno). Cada uno de ellos tiene un valor de venta $v_1, \ldots, v_n$ y requiere para su elaboración cantidades $a_1, \ldots, a_n$ de materia prima de tipo A y $b_1, \ldots, b_n$ de materia prima de tipo B. ¿Cuál es el mayor valor alcanzable con las cantidades de materia prima disponible?

5. Analizando el enunciado:
   - Tenemos n productos que fabricar. Cada pedido tiene un valor de venta $v_1, \ldots, v_n$ y requiere para su elaboración cantidades $a_1, \ldots, a_n$ de materia prima de tipo A y $b_1, \ldots, b_n$ de materia prima de tipo B.
   - Tenemos una cantidad disponible MA y MB de cada materia prima.

   Se desea fabricar algunos de los productos de forma tal de alcanzar el mayor valor con las cantidades de materia prima disponible. Es claro que hay que calcular un máximo.

6. Identifiquemos los parámetros de la función recursiva y qué calcula en función de estos:
   IMPORTANTE: el algoritmo deberá obtener el mayor valor alcanzado pero no qué productos se fabricaron.

artesanias(i, ma, mb) = “el mayor valor posible que se puede obtener fabricando algunos de los productos entre 1 e i de forma tal que la suma de la materia prima a requerida para cada producto sea menor a igual a ma y la suma de la materia prima a requerida para cada producto sea menor a igual a mb”

Luego, la llamada a esta función recursiva que resuelve el problema del ejercicio es:

artesanias(n, MA, MB)

7. Definamos la función artesanias:

Siendo i el índice que indica hasta qué producto voy a considerar, ma la cantidad de materia prima a disponible, y mb la cantidad de materia prima b disponible:

	CASOS	**SOLUCIÓN EN CADA CASO**
artesanias (i, ma, mb)	ma = 0, mb = 0	Caso en el que no tengo ni materia prima a ni materia prima b disponible. Entonces, no puedo fabricar ningún producto, y por lo tanto el máximo (y único) valor posible que puedo alcanzar fabricando 0 objetos es 0. **0** {- OBS.: Este caso no es estrictamente necesario. -}
	i = 0	Caso en el que no tengo ningún producto para fabricar. Entonces, el máximo (y único) valor posible que puedo alcanzar fabricando 0 objetos es 0. **0**
	i > 0, ma ≥ 0, mb ≥ 0, a_i > ma o b_i > mb	Caso en el que tengo algún producto para fabricar (i > 0), tengo materia prima a y materia prima b disponible (ma > 0, mb > 0), pero la materia prima necesaria para fabricar el producto i es mayor a la materia prima disponible de alguno de los dos tipos (a_i > ma o b_i > mb). En este caso, descarto el producto i para fabricarlo (no me alcanza la materia prima): **artesanias(i-1, ma, mb)**
	i > 0, ma ≥ 0, mb ≥ 0, a_i ≤ ma, b_i ≤ mb	Caso en el que tengo algún producto para fabricar (i > 0), tengo materia prima a y materia prima b disponible (ma > 0, mb > 0), y la materia prima necesaria para fabricar el producto i es menor o igual a la materia prima disponible de ambos tipos (a_i <= ma, b_i <= mb).

		{- pruebo fabricando el objeto i -} v_i + artesanias(i-1, ma - a_i, mb - b_i) {- pruebo no fabricar el objeto i -} artesanias(i-1, ma, mb) {- debo calcular el máximo entre estas dos opciones -} **max(v_i + artesanias(i-1, ma-a_i, mb-b_i), artesanias(i-1, ma, mb))**

Ejercicio 7:

En el problema de la mochila se buscaba el máximo valor alcanzable al seleccionar entre n objetos de valores v_1, … , v_n y pesos w_1, … , w_n, respectivamente, una combinación de ellos que quepa en una mochila de capacidad W. Si se tienen dos mochilas con capacidades W_1 y W_2, ¿cuál es el valor máximo alcanzable al seleccionar objetos para cargar en ambas mochilas?

Recordemos primero cómo era el problema de la mochila con una mochila:

- Tenemos una mochila de capacidad W.
- Tenemos n objetos **no fraccionables** de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$.
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza **el mayor valor posible** sin que su peso exceda la capacidad W de la mochila.

La solución con backtracking al problema de la mochila era:

Definimos $m(i, j) =$ "mayor valor alcanzable sin exceder la capacidad j con objetos $1, 2, \ldots, i$."

$$m(i,j) = \begin{cases} 0 & j = 0 \\ 0 & j > 0 \wedge i = 0 \\ m(i-1,j) & w_i > j > 0 \wedge i > 0 \\ \max(m(i-1,j), v_i + m(i-1, j-w_i)) & j \geq w_i > 0 \wedge i > 0 \end{cases}$$

El resultado al problema lo obtenemos llamando a $m(n, W)$.

1. Analizando el enunciado de este ejercicio, veámoslo con dos mochilas:
   - Tenemos DOS mochilas con capacidades W_1 y W_2.
   - Tenemos n objetos de valores v_1, … , v_n y pesos w_1, … , w_n, respectivamente.

2. Identifiquemos los parámetros de la función recursiva y qué calcula en función de estos:

`mochilas(i, j, w) = “máximo valor posible que se puede alcanzar guardando algunos de los objetos entre 1 e i en dos mochilas con capacidad j y w respectivamente, de forma tal que no se exceda la capacidad de ninguna de las dos mochilas”`

Luego, es claro que la llamada principal de la función recursiva es:

`mochilas(n, W₁, W₂)`

3. Definamos la función mochilas:

Siendo i el índice que indica hasta qué objeto voy a considerar, j la capacidad de la mochila y k la capacidad de la mochila 2:

	CASOS	**SOLUCIÓN EN CADA CASO**
mochilas (i,j,k)	i = 0	Caso en el que no tengo ningún objeto por guardar. Entonces, el único valor que puedo alcanzar guardando 0 objetos es 0. **0**
	j = 0, k = 0	Caso en el que en ninguna de las dos mochilas queda capacidad disponible. Por lo tanto, no puedo guardar ningún objeto. Luego, el único valor que puedo alcanzar guardando 0 objetos es 0. **0**
	w_i > j, w_i > k	Caso en el que el peso del objeto i supera la capacidad disponible en ambas mochilas. Entonces, es claro que no puedo guardar ese objeto en ninguna de las dos mochilas. Luego, lo descarto: **mochilas(i-1, j, k)**
	i > 0, w_i <= j, w_i > k	Caso en el que en el objeto i entra en la primera mochila, pero no en la segunda: {- elijo guardar el objeto i en la primera mochila -} v_i + mochilas(i-1, j-w_i, k) {- elijo no guardar el objeto i en la primera mochila -} mochilas(i-1, j, k) Debo calcular el máximo entre estas dos opciones: **max (v_i + mochilas(i-1, j-w_i, k), mochilas(i-1, j, k))**

	$i > 0$, $w_i > j$, $w_i <= k$	Caso en el que en el objeto i solo entra en la segunda mochila, pero no en la primera: {- elijo guardar el objeto i en la segunda mochila -} v_i + mochilas(i-1, j, k-w_i) {- elijo no guardar el objeto i en la segunda mochila -} mochilas(i-1, j, k) Debo calcular el máximo entre estas dos opciones: **max (v_i + mochilas(i-1, j, k-w_i), mochilas(i-1, j, k))**
	$i > 0$, $w_i <= j$, $w_i <= k$	Caso en el que en el objeto i entra tanto en la primera como en la segunda mochila, por lo tanto puedo guardarlo en cualquiera de las dos: {- elijo guardar el objeto i en la primera mochila -} v_i + mochilas(i-1, j-w_i, k) {- elijo guardar el objeto i en la segunda mochila -} v_i + mochilas(i-1, j, k-w_i) {- elijo no guardar el objeto i en ninguna de las dos mochilas-} mochilas(i-1, j, k) Debo calcular el máximo entre estas tres opciones: **max (v_i + mochilas(i-1, j-w_i, k), v_i + mochilas(i-1, j, k-w_i), mochilas(i-1, j, k))**

TRADUCCIÓN A PROGRAMACIÓN DINÁMICA:

¿Qué forma tiene la tabla que voy a llenar?

Va a ser un arreglo de tres dimensiones: [0..n, 0..W_1, 0..W_2]

¿En qué orden se debe llenar la tabla?

Para responder esta pregunta, debemos consultar la definición recursiva.

Los pisos (1ra dimensión) se deben llenar de abajo hacia arriba (en la matriz sería llenar las filas desde arriba hacia abajo), ya que para calcular para el piso i siempre uso valor 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 mochilas(v:array[1..n] of nat, w:array[1..n] of nat, W1:nat, W2:nat) ret
r: nat
```

Implementación:

```
fun mochilas(v:array[1..n] of nat, w:array[1..n] of nat, W1:nat, W2:nat) ret
r: nat
     var tabla:array [0..n, 0..W1, 0..W2] of nat

     {- CASOS BASE -}
     for j:=0 to W1
          for k:=0 to W2
               tabla[0,j,k] := 0        {- planta baja -}
          od
     od
     for i:=0 to n
          tabla[i,0,0] := 0
     od

     {- CASOS RECURSIVOS -}
     for i:=1 to n
       {- en los siguientes ciclos, notar que no empiezo desde 1. Pensar en
     la
       posición (7, 10, 0) -}
       for j:=0 to W1
          for k:=0 to W2
            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]
            else if w[i] <= j and w[i] > k then
               tabla[i,j,k] := max(v[i] + tabla[i-1, j-w[i], k], tabla[i-
1,j,k])
            else if w[i] > j and w[i] <= k then
               tabla[i,j,k] := max(v[i] + tabla[i-1, j, k-w[i]], tabla[i-
1,j,k])
            else if w[i] <= j and w[i] <= k then
               tabla[i,j,k] := max(v[i] + tabla[i-1, j-w[i], k], v[i] +
tabla[i-1,
                                                    j, k-w[i]], tabla[i-
                                1,j,k])
            fi
          od
        od
     od
```

```
     {- devolver llamada prinicpal -}
     r := tabla[n, W1, W2]
end fun
```

Versión más corta:

```
fun mochilas(v:array[1..n] of nat, w:array[1..n] of nat, W1:nat, W2:nat) ret
r: nat
     var tabla:array [0..n, 0..W1, 0..W2] of nat

     {- CASOS BASE -}
     for j:=0 to W1
          for k:=0 to W2
               tabla[0,j,k] := 0        {- planta baja -}
          od
     od
     for i:=0 to n
          tabla[i,0,0] := 0
     od

     {- CASOS RECURSIVOS -}
     for i:=1 to n
       {- en los siguientes ciclos, notar que no empiezo desde 1. Pensar en
     la
       posición (7, 10, 0) -}
       for j:=0 to W1
          for k:=0 to W2
            if j = 0 and k = 0 then
               skip
            else
               mimax := tabla[i-1, j, k]
               if w[i] <= j then
                    mimax := max(mimax, v[i] + tabla[i-1, j-w[i], k])
               fi
               if w[i] <= k then
                    mimax := max(mimax, v[i] + tabla[i-1, j, k-w[i]])
               fi
               tabla[i, j, k] := mimax
            fi
          od
        od
     od
```

```
     {- devolver llamada prinicpal -}
     r := tabla[n, W1, W2]
end fun
```