Notas de Consulta Online Parte 2

...

Algoritmos y Estructuras de Datos II

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

- Me encargan n pedidos. Por cada pedido i me pagarán un monto m_i.
- Cada pedido i requiere una cantidad h_i de harina.
- Solo tengo una cantidad total H de harina.
- Debo elegir qué pedidos me conviene hacer, de manera que gane más dinero.

Ejemplo. 4 pedidos:
m_1 = 50, h_1 = 300
m_2 = 70, h_2 = 450
m_3 = 30, h_3 = 300
m_4 = 40, h_4 = 350

H = 750

Idea del backtracking:
* ¿Puedo hacer el pedido 4? Sí. probemos hacerlo. Entonces
  - Ganaré 40
  - Me quedan 400 de harina.
* ¿Puedo hacer ahora el 3? Sí, probemos hacerlo. Entonces
  - Ganaré 70
  - Me quedan 100 de harina.
* ¿Puedo hacer el 2? No.
* ¿Puedo hacer el 1? No.

TOTAL: Gané 70.

Marcha atrás, cambiemos la primera decisión.

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

- Me encargan n pedidos. Por cada pedido i me pagarán un monto m_i.
- Cada pedido i requiere una cantidad h_i de harina.
- Solo tengo una cantidad total H de harina.
- Debo elegir qué pedidos me conviene hacer, de manera que gane más dinero.

Ejemplo. 4 pedidos:
m_1 = 50, h_1 = 300
m_2 = 70, h_2 = 450
m_3 = 30, h_3 = 300
m_4 = 40, h_4 = 350

H = 750

Con el camino anterior gané 70.
Marcha atrás, cambiemos la primera decisión.

* ¿Me alcanza para hacer el pedido 4? Sí, **pero elijo no hacerlo**.
* ¿Me alcanza para hacer el pedido 3? Sí, ok lo hago. Entonces
  - Ganaré 30
  - Me quedan 450 de harina
* ¿Me alcanza para hacer el pedido 2? Sí, ok lo hago. Entoncs
  - Ganaré 100
  - Me queda 0 de harina

TOTAL: Gané 100

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

Voy a definir una función recursiva
panaderia(i,j) = *"Mayor monto que puedo obtener realizando algunos de los pedidos entre 1 e i, de manera tal que la harina necesaria no supere el monto j"*

¿Cuál es la llamada a esta función que me va a obtener la solución que me piden en el enunciado?
panaderia(n , H)

Definamos panaderia:
panaderia(i,j) =

- Si i = 0 ------> 0 - *No tengo pedidos para hacer.*
- Si i > 0 y h_i > j ------> panaderia(i-1,j) - *La harina no me alcanza para el pedido i.*
- Si i > 0 y h_i <= j ------> max(m_i + panaderia(i-1, j - h_i) , panaderia(i-1 , j)) - *La harina me alcanza, pruebo hacerlo o no hacerlo.*

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

panaderia(i,j) =

- Si i = 0 ------> 0 *- No tengo pedidos para hacer.*
- Si i > 0 y h_i > j ------> panaderia(i-1,j) *- La harina no me alcanza para el pedido i.*
- Si i > 0 y h_i <= j ------> max(m_i + panaderia(i-1, j - h_i) , panaderia(i-1 , j))

- La harina me alcanza, pruebo hacerlo o no hacerlo.

Apliquemos la función en el ejemplo de los 4 pedidos.
panaderia (4 , 750) = max (40 + panaderia(3,400) , panaderia(3,750))

Elijo hacer el pedido

Elijo NO hacer el pedido.

Ejemplo. 4 pedidos:
m_1 = 50, h_1 = 300
m_2 = 70, h_2 = 450
m_3 = 30, h_3 = 300
m_4 = 40, h_4 = 350

H = 750

Ahora debería calcular panaderia(3,400) y también panaderia(3,750)

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

panaderia(i,j) =

- Si i = 0 ------> 0 *- No tengo pedidos para hacer.*
- Si i > 0 y h_i > j ------> panaderia(i-1,j) *- La harina no me alcanza para el pedido i.*
- Si i > 0 y h_i <= j ------> max(m_i + panaderia(i-1, j - h_i) , panaderia(i-1 , j)) *- La harina me alcanza, pruebo hacerlo o no hacerlo.*

Ejemplo de Leandro. 4 pedidos:
m_1 = 50, h_1 = 300
m_2 = 70, h_2 = 450
m_3 = 30, h_3 = 300
m_4 = 40, h_4 = 350

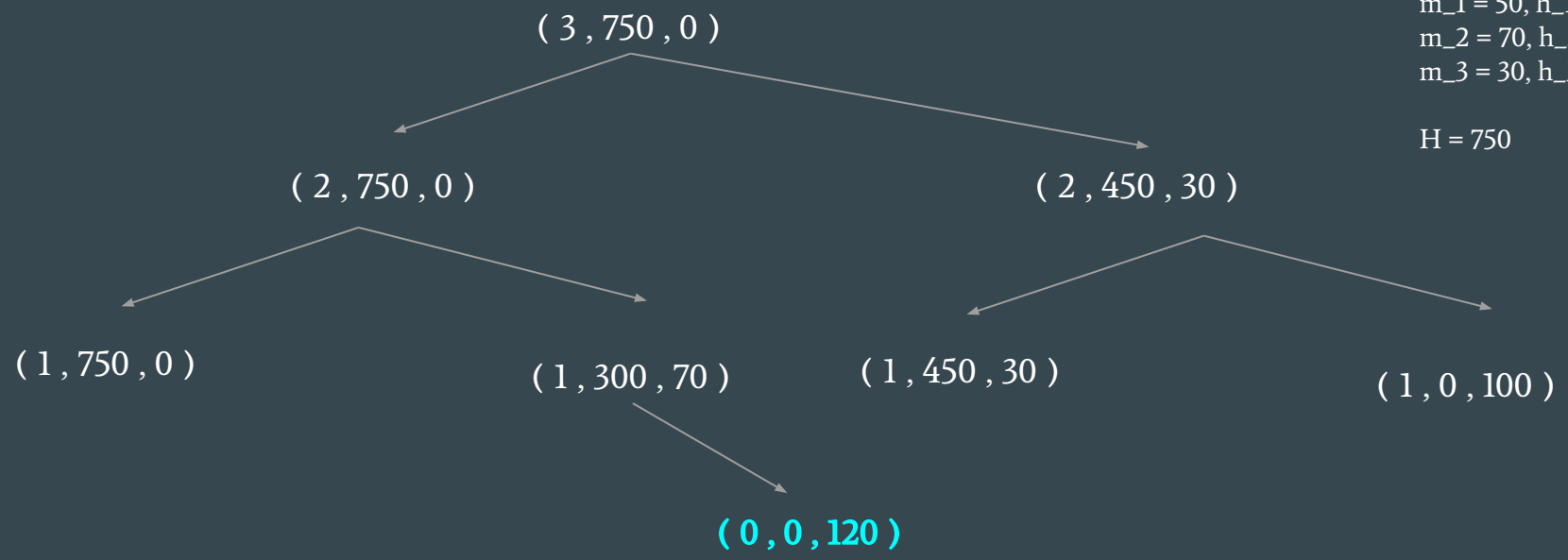
H = 800

panaderia (4 , 800) = max (40 + pan(3,450) , pan(3,800))
Luego,
40 + pan(3,450) = 40 + max (30 + pan(2,150) , pan(2,450))

Práctico 3.3. Ejercicio 3: Pedidos de panadería. Grafo

Ejemplo. 3 pedidos:
m_1 = 50, h_1 = 300
m_2 = 70, h_2 = 450
m_3 = 30, h_3 = 300

H = 750



Práctico 3.3. Ejercicio 3: Pedidos de panadería.

panaderia(i,j) =

- Si i = 0 ------> 0 *- No tengo pedidos para hacer.*
- Si i > 0 y h_i > j ------> panaderia(i-1,j) *- La harina no me alcanza para el pedido i.*
- Si i > 0 y h_i <= j ------> max(m_i + panaderia(i-1, j - h_i) , panaderia(i-1 , j))
  - La harina me alcanza, pruebo hacerlo o no hacerlo.

```
type Pedido = tuple
                h : Nat
                m : Nat
              end tuple
```

Práctico 3.3. Ejercicio 3: Pedidos de panadería.

panaderia(i,j) =

- Si i = 0 ------> 0 *- No tengo pedidos para hacer.*
- Si i > 0 y h_i > j ------> panaderia(i-1,j) *- La harina no me alcanza para el pedido i.*
- Si i > 0 y h_i <= j ------> max(m_i + panaderia(i-1, j - h_i) , panaderia(i-1 , j)) *- La harina me alcanza, pruebo hacerlo o no hacerlo.*

```
fun panaderia(p : array[1..n] of Pedido, i : Nat, j : Nat) ret r : Nat

  if       (i = 0)      then r := 0
  else if  (p[i].h > j) then r := panaderia(p,i-1,j)
  else r := max( p[i].m + panaderia(p,i-1,j-p[i].h),
                 panaderia(p,i-1,j))
  fi
end fun
```

Práctico 3.3: Ejercicio 1

```
{- devolvemos un par (n, l) a donde n : nat, l : List of nat -}
fun cambio(d:array[1..n] of nat, i,j: nat)ret r: nat x List of nat
  var r1, r2: nat x List of nat

  if j = 0 then r := (0, empty_list())
  else if i = 0 then r := (∞, empty_list())   {- cualquier lista da igual acá -}
  else if d[i] > j then
    r := cambio(d,i-1,j)
  else
    {- acá está lo interesante -}
    r1 := cambio(d,i-1,j)      {- r1 es un par -}
    r2 := cambio(d,i,j-d[i])  {- r2 es un par -}
    if r1.fst < 1 + r2.fst then
      r := r1
    else
      addr(r2.snd, d[i])
      r.fst := 1 + r2.fst
      r.snd := r2.snd
    fi
  fi
end fun
```

Práctico 3.3. Ejercicio 5: Teléfono Satelital.

- Quiero alquilar teléfono satelital por día.
- Tengo n amigos.
- Cada amigo i tiene:
  - día de partida p_i
  - día de regreso r_i.
  - pago por día m_i.
- Quiero obtener el máximo valor alquilando el teléfono.

Práctico 3.3. Ejercicio 5: Teléfono Satelital.

Voy a definir una función recursiva
telefono(i,d) = “máximo monto obtenible al alquilar el teléfono a algunos de los amigos desde el 1 hasta el i, **a partir del día** d”

¿Cuál es la llamada a esta función que me va a obtener la solución que me piden en el enunciado?
telefono(n,1)

Explicación de la idea:
Al llamar a telefono(n,1) calcularé qué pasa si le alquilo al amigo n, y también si NO se lo alquilo.
En el primer caso, luego calcularé qué pasa si se lo alquilo al n-1 o no, PERO SOLO SI ES POSIBLE. Es decir, si el día de partida de n-1 p_(n-1) es mayor a d.

telefono(i,d) =

- Si i = 0 ------> 0 *- No tengo amigos para alquilarle.*
- Si p_i < d ------> telefono(i-1,d) *- No se lo puedo alquilar al i.*
- Si p_i >= d ------> max(telefono(i-1,d) , m_i * (r_i - p_i) + telefono(i-1, r_i + 1))

ESTA SOLUCION SENCILLA SOLO FUNCIONA SI LOS AMIGOS VIENEN ORDENADOS DE MAYOR A MENOR DE ACUERDO AL DIA DE PARTIDA

Programación Dinámica: Problema de la Mochila

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

- Esta función tiene complejidad exponencial (O(2^i))
  - La función se llama dos veces a sí misma al estilo fibonacci.
- Hay cálculos repetidos. Se puede bajar la complejidad.
  - Podemos pre-calcular los valores en una tabla.

Programación Dinámica: Problema de la Mochila

- ¿qué dimensiones tiene la tabla a calcular? Debemos ver el enunciado.
- Llamada principal: mochila(n, W).
- La tabla será de tamaño (n+1)x(W+1) (índices 0..n y 0..W).
- Ejemplo: n = 4, W = 16
  - v1 = 3, v2 = 2, v3 = 3, v4 = 2
  - w1 = 8, w2 = 5, w3 = 7, w4 = 3

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0																	
1																	
2																	
3																	
4																	

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1																	
2																	
3																	
4																	

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0																
2	0																
3	0																
4	0																

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0																
2	0			??										??			
3																	
4	0																

mochila(2,3) = mochila(1, 3)
pues w_i = 5 > j = 3.

mochila(2,13) = max(mochila(1,13), vi+mochila(1,8))
pues w_i = 5 <= j = 13.

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	3
2	0																
3	0																
4	0																

Primero llenamos la fila 1.

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	3
2	0	0	0	0	0	2	2	2	3	3	3	3	3	5	5	5	5
3	0																
4																	

Ahora la fila 2.

max(3, 2 + 0)

max(3, 2 + 3)

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0								0	0	0	0	0	0	0	0	0	0
1								0	3	3	3	3	3	3	3	3	3
2	0	0	0	0	0	2	2	2	3	3	3	3	3	5	5	5	5
3	0	0	0	0	0	2	2	3	3	3	3	3	5	5	5	6	6
4	0																

Ahora la fila 3.

Programación Dinámica: Problema de la Mochila

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

i	vi	wi
1	3	8
2	2	5
3	3	7
4	2	3

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0								0	0	0	0	0	0	0	0	0	0
1								0	3	3	3	3	3	3	3	3	3
2	0	0	0	0	0	2	2	2	3	3	3	3	3	5	5	5	5
3	0	0	0	0	0	2	2	3	3	3	3	3	5	5	5	6	6
4	0	0	0	2	2	2	2	3	4	4	5	5	5	5	5	7	7!!

Ahora la fila 4.

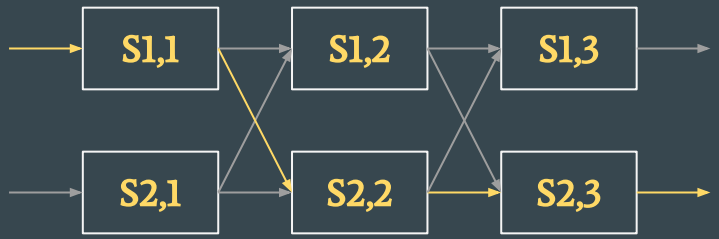
Programación Dinámica: Problema de la Mochila

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

```
fun mochilaPD(v: array[1..n] of nat,w: array[1..n] of nat,W : nat) ret r : nat
  var tabla : array[0..n,0..W] of nat
  // llenemos la tabla
  for j:=1 to W do tabla[0,j] := 0 od
  for i:=0 to n do tabla[i,0] := 0 od
  // ahora completo de arriba hacia abajo, y en cada fila de izq a der
  for i:=1 to n do
    for j:=1 to W do
      if w[i]>j then tabla[i,j] := tabla[i-1,j]
      else tabla[i,j] := max( tabla[i-1,j] , v[i] + tabla[i-1,j-w[i]] )
      fi
    od
  od
  r := tabla[n,W]
end fun
```

Práctico 3.3. Ejercicio 8: Fábrica de autos

- Debo fabricar un auto mediante **n** estaciones o **etapas.**
- Cada etapa la puedo realizar el la línea 1 o en la línea 2.
- La fabricación de una etapa j tiene costo a_1,j si se realiza en la línea 1, y a_2,j en la línea 2.
- Si realizo la etapa j en la estación 1, y quiero hacer la etapa j+1 en la estación 2, debo pagar un costo extra de t_i,j. Igual el caso análogo.



Ejemplo. n = 3.
* a_1,1 = 20 a_2,1 = 25 t_1,1 = 5 t_2,1 = 5

* a_1,2 = 14 a_2,2 = 12 t_1,2 = 6 t_2,2 = 4

* a_1,3 = 16 a_2,3 = 22

¿cuántos recorridos posibles hay? 2 * 2 * 2 = 2^3 = 8

¿cuánto cuesta fabricar el auto según ese recorrido?
El recorrido es: S_1,1 S_2,2 S_2,3

a_1,1 + t_1,1 + a_2,2 + a_2,3
20 + 5 + 12 + 22 = 59

Práctico 3.3. Ejercicio 8: Fábrica de autos

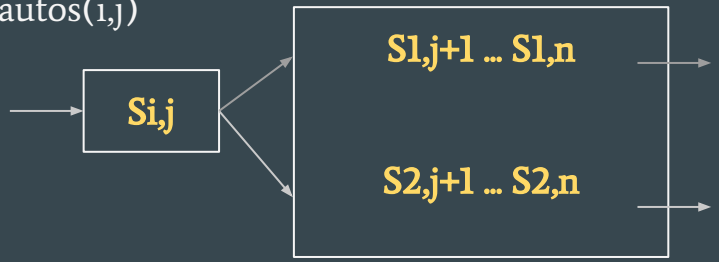
Voy a definir una función recursiva. Dos alternativas:

1. autos(i,j) = " Menor costo de fabricar el auto **desde** la estación j en la línea i, hasta la estación n, en alguna de las dos líneas "
2. autos(i,j) = " Menor costo de fabricar el auto **desde** la estación 1, **hasta** la estación j, haciendo la misma en la línea i "

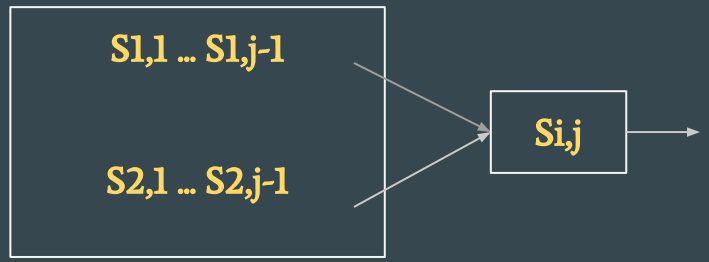
¿Cuál es la llamada o expresión con esta función que me va a obtener la solución que me piden en el enunciado?

1. min (autos(1,1) , autos(2,1))
2. min (autos(1,n) , autos(2,n))

1. autos(i,j)



2. autos(i,j)



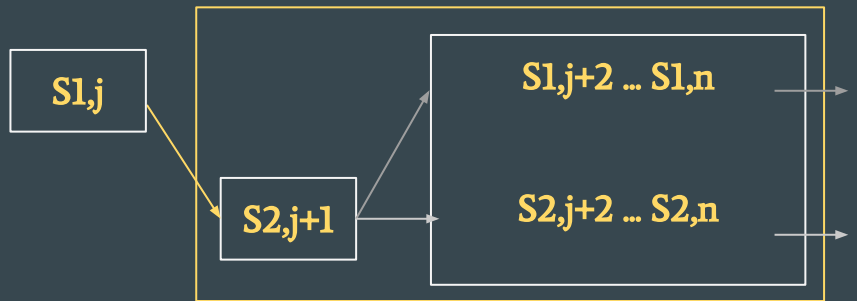
Práctico 3.3. Ejercicio 8: Fábrica de autos

Definamos la función:
Alternativa 1.
autos(i,j) =

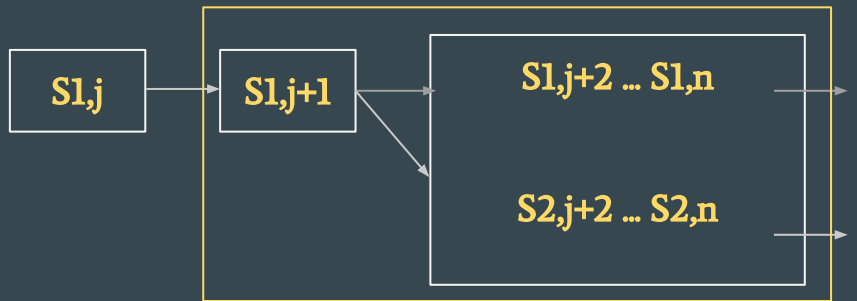
- Si j = n ---> a_i,n
- Si j < n, i = 1 ---> a_1,j + min (autos(1,j+1) , autos(2,j+1) + t_1,j)
- Si j < n, i = 2 ---> a_2,j + min (autos(2,j+1) , autos(1,j+1) + t_2,j)

1. autos(i,j). Caso j < n, i = 1: Mínimo entre dos opciones:

a_1,j + t_1,j + autos(2,j+1)



a_1,j + autos(1,j+1)



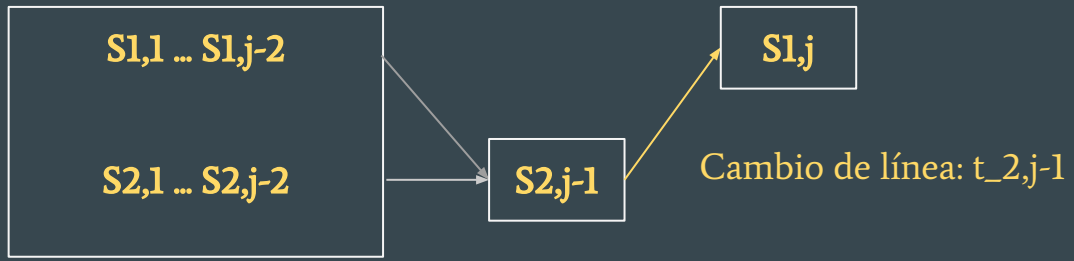
Práctico 3.3. Ejercicio 8: Fábrica de autos

Alternativa 2.
autos(i,j) = " Menor costo de fabricar el auto **desde** la estación 1, **hasta** la estación j, haciendo la misma en la línea i"

Definamos la función:
autos(i,j) =

- j = 1 ---> a_i,1
- j > 1, i = 1 ---> a_1,j + min (autos(1,j-1) , autos(2,j-1) + t_2,j-1)
- j > 1, i = 2 ---> a_2,j + min (autos(2,j-1) , autos(1,j-1) + t_1,j-1)

2. autos(i,j), caso j > 1, i =1, segundo término del min:



Programación Dinámica Alternativa 1:

autos(i,j) =

- Si j = n ---> a_i,n
- Si j < n, i = 1 ---> a_1,j + min (autos(1,j+1) , autos(2,j+1) + t_1,j)
- Si j < n, i = 2 ---> a_2,j + min (autos(2,j+1) , autos(1,j+1) + t_2,j)

```
fun autosPD(a: array[1..2,1..n] of nat, t: array[1..2,1..n-1] of nat) ret r : nat
  var tabla: array[1..2,1..n] of nat      {- tabla[i,j] = autos(i,j) -}

  for i := 1 to 2 do
    tabla[i,n] := a[i,n]      {- o directamente: tabla[1,n] := a[1,n]; tabla[2,n] := a[2,n] --}
  od
  for j := n-1 downto 1 do
    {- fila 1 columna j -}
    tabla[1,j] := a[1,j] + min(tabla[1,j+1], tabla[2,j+1] + t[1,j])
    {- fila 2 columna j -}
    tabla[2,j] := a[2,j] + min(tabla[2,j+1], tabla[1,j+1] + t[2,j])
  od
  r := min(tabla[1,1], tabla[2,1])
end fun
```

Práctico 3.3. Ejercicio 7: Dos mochilas

- Tengo n objetos.
- Cada objeto i tiene valor v_i y peso w_i.
- Tengo dos mochilas, con capacidad W1 y W2.
- Debo elegir qué objetos meter entre las dos mochilas de manera de que el valor total sea máximo.

¿Qué casos debería considerar?
Supongamos que estoy “viendo” el objeto i. Supongamos que en la mochila 1 queda j de capacidad y en la mochila 2, k de capacidad.

Opciones:
* w_i > j y w_i > k (no entra en ninguna)
* w_i <= j y w_i > k (entra **solo** en la 1)
* w_i > j y w_i <= k (entra **solo** en la 2)
* w_i <= j y w_i <= k (entra en ambas)

Práctico 3.3. Ejercicio 7: Dos mochilas

Voy a definir una función recursiva
2mochilas(i,j,k) = *"máximo valor posible al guardar algunos objetos entre el 1 y el i, dado que a la mochila 1 le queda de capacidad j, y a la mochila 2 le queda de capacidad k"*

¿Cuál es la llamada a esta función que me va a obtener la solución que me piden en el enunciado?
2**mochilas(**n, W_1, W_2)

Definamos la función:
2mochilas(i,j,k) =

- i = 0 -------> 0
- i > 0, w_i > j, w_i > k -------> 2mochilas(i-1 , j , k)
- i > 0, w_i<=j, w_i > k -------> max(2mochilas(i-1 , j , k) , v_i + 2mochila(i-1 , j-w_i , k))
- i > 0, w_i>j, w_i <= k -------> max(2mochilas(i-1 , j , k) , v_i + 2mochila(i-1 , j , k-w_i))
- i > 0, w_i<=j, w_i <= k ------->

max(2mochilas(i-1 , j , k), v_i + 2mochila(i-1 , j-w_i , k) , v_i + 2mochila(i-1 , j , k-w_i))

Práctico 3.3. Ejercicio 7: Dos mochilas con PD

- **¿qué dimensiones tiene la tabla a calcular? Debemos ver el enunciado.**
- Llamada principal: 2mochilas(n, W1, W2).
- La tabla será de tamaño (n+1)x(W1+1)x(W2+1) (índices 0..n, 0..W1, 0..W2).

2mochilas(i,j,k) =

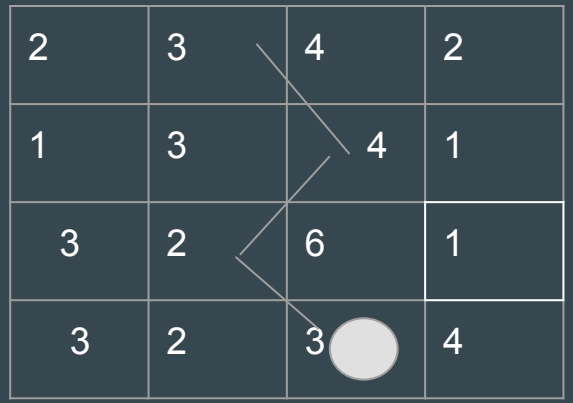
- i = 0 -------> 0
- i > 0, w_i > j, w_i > k -------> 2mochilas(i-1 , j , k)
- i > 0, w_i<=j, w_i > k -------> max(2mochilas(i-1 , j , k) , v_i + 2mochila(i-1 , j-w_i , k))
- i > 0, w_i>j, w_i <= k -------> max(2mochilas(i-1 , j , k) , v_i + 2mochila(i-1 , j , k-w_i))
- i > 0, w_i<=j, w_i <= k ------->
  max(2mochilas(i-1 , j , k), v_i + 2mochila(i-1 , j-w_i , k) , v_i + 2mochila(i-1 , j , k-w_i))

```
fun 2mochilasPD(v: array[1..n] of nat, w: array[1..n] of nat, W1, W2 : nat) ret r : nat
  var tabla: array[0..n,0..W1,0..W2] of nat
 {- tabla[i,j,k] = 2mochilas(i,j,k) = "máximo valor posible bla bla bla..." -}
 {- ¿en qué orden llenamos esta tabla? ¿qué podemos llenar primero? -}
 {- en la dimensión 1 tenemos que ir de 0 hacia adelante -}
  for j := 0 to W1 do
    for k := 0 to W2 do
      tabla[0,j,k] := 0
    od
  od
  for i := 1 to n do
    for j := 0 to W1 do
      for k := 0 to W2 do
        if w[i] > j and w[i] > k ->       tabla[i,j,k] := tabla[i-1,j,k]
        else if w[i] <= j and w[i] > k ->
              tabla[i,j,k] := max( tabla[i-1, j, k] , v[i] + tabla[i-1, j-w[i], k] )
        else if {- EJERCICIO: COMPLETAR!! -}
        fi
      od
```

Práctico 3.3. Ejercicio 9: Juego “up”

- Tenemos un tablero de n filas por n
- columnas.
- Cada casillero del tablero tiene un puntaje asociado, c_ij.
- El puntaje total es el de cada casillero por el que haya pasado la ficha.
- Se pide encontrar la mejor jugada posible, teniendo que elegir también desde dónde empiezo.

Ejemplo



¿Qué puntaje hice con ese juego? 12

¿Cuál es el máximo puntaje que puedo obtener? Sería 4 + 6 + 4 + 4 = 18.

Práctico 3.3. Ejercicio 4: Globo Aerostático

Voy a definir una función recursiva
globo(i, p) = *“ el menor valor que es posible perder descartando algunos de los objetos entre 1 e i, de manera que la suma de los pesos de esos objetos sea igual o mayor a p”*

¿Cuál es la llamada o expresión con esta función que me va a obtener la solución que me piden en el enunciado?
globo(n, P)

Definamos la función:
globo(i, p) =

- Si p = 0 --> 0
- Si i = 0 and p > 0 --> infinito // me hundo. Sólo queda rezar.
- Si i > 0 --> min(globo(i-1, p), // ← si no lo tiro
  v_i + globo(i-1, max(**p - p_i, 0)**)) // ← si lo tiro.

Otra forma: Dos casos recursivos:

- i > 0 and p_i <= p --> min(globo(i-1, p), v_i + globo(i-1, **p - p_i**))
- i > 0 and p_i > p --> min(globo(i-1, p), v_i)

Práctico 3.3. Ejercicio 9: Juego “up”

Voy a definir una función recursiva
mejor_juego(i,j) = *“Mayor puntaje obtenible jugando al up, desde el casillero [i,j] hasta algún casillero de la última fila, es decir, hasta [n,k], donde k está entre 1 y n”*

¿Cuál es la llamada o expresión con esta función que me va a obtener la solución que me piden en el enunciado?
max (mejor_juego(1,1), mejor_juego(1,2),.....,mejor_juego(1,n)) =

Definamos la función:
mejor_juego(i,j) =

- Si i = n -------> c_n,j
- Si i < n, j = 1 -------> c_i,1 + max (mejor_juego(i+1 , 1) , mejor_juego(i+1 , 2))
- Si i < n, j = n -------> c_i,n + max (mejor_juego(i+1 , n-1) , mejor_juego(i+1,n))
- Si i < n, 1<j<n ------> c_i,j + max (mejor_juego(i+1,j-1) , mejor_juego(i+1,j) , mejor_juego(i+1,j+1))

Examen Final de Práctica (24 julio 2019). Ejercicio 1

Especificación:

```
spec Urna where

constructors
 fun urna_vacia() ret Urna
 proc votar_x(in/out u : Urna)
 proc votar_y(in/out u : Urna)
 proc votar_blanco(in/out u : Urna)

operations
 proc juntar_urnas(in/out u : Urna, in v : Urna)
 ... {- el resto acá -}
```

Examen Final de Práctica (24 julio 2019). Ejercicio 1

Implementación:

```
implement Urna where

type Urna = tuple
              votos_x : nat
              votos_y : nat
              votos_blanco : nat
            end tuple
```

Examen Final de Práctica (24 julio 2019). Ejercicio 2

* ¿Qué hace la función q?

La función **q** toma un arreglo de longitud n y un natural i. Devuelve el mismo natural i si es impar, y sino, devuelve el índice del menor elemento del arreglo desde la posición i, de entre todos los que se encuentren en posiciones pares..

* ¿Cómo lo hace?

En caso que “i” sea par, recorre el arreglo solo en las posiciones pares, de izquierda a derecha, desde la posición i, y calcula el índice del menor de esos elementos.

* El orden de la función es (n-i)/2.
* Un nombre más adecuado para q: min_pos_even_from

Examen Final de Práctica (24 julio 2019). Ejercicio 2

* ¿Qué hace el procedimiento p?

Toma un arreglo a de n elementos. Ordena de menor a mayor los elementos que se encuentran en posiciones pares de a.

* ¿Cómo lo hace?

Recorre el arreglo de izquierda a derecha, desde la posición 1. Para cada elemento, si está en posición impar lo deja igual, sino, lo intercambia por el mínimo de entre los que están en posiciones pares.

* El orden es n^2. O sea, es cuadrático.
* Un nombre más adecuado para p: selection_even_sort

Examen Final de Práctica (24 julio 2019). Ejercicio 3

- N ambientes de la casa. Limpiar cada ambiente “i” le lleva t_i de tiempo, y le retribuye una valoración v_i.
- Dispone de T tiempo para limpiar ambientes. Puede limpiarlos parcialmente.

Criterio de selección voraz:
Elijo el ambiente que tenga mayor valor por unidad de tiempo.

Tarea: Terminarlo.

Examen Final de Práctica (24 julio 2019). Ejercicio 4

- N propuestas de gobierno.
- Cada propuesta i, genera satisfacción p_i, y desagrado q_i.
- Debe elegir K propuestas, entre las N.
- Se deben seleccionar K propuestas con satisfacción máxima, sin que el desagrado total supere un monto M.

Ejemplo. N = 4. K = 2. M = 45.
p_1 = 30, q_1 = 24
p_2 = 20, q_2 = 10
p_3 = 50, q_3 = 32
p_4 = 25, q_4 = 18

Es backtracking, así que debo “probar” con todos los subconjuntos de K propuestas cuya suma total de desagrado no supere M.

Una opción serán las propuestas 1 y 2, cuya popularidad total será de 50 y el desagrado de 34.
Las propuestas 1 y 3 no pueden ser elegidas juntas, pues su desagrado supera M.
Otra opción válida son las propuestas 1 y 4, cuya popularidad es 55, y el desagrado 42.

IMPORTANTE: No pide que minimicemos el desagrado. Solo debemos asegurar que no supera M.
Pide que maximicemos la suma total de satisfacción, siempre y cuando el desagrado no supere M.

Examen Final de Práctica (24 julio 2019). Ejercicio 4

Voy a definir una función recursiva
propuestas(i,k,m) = *"La mayor satisfacción popular obtenible al elegir* k *propuestas de entre las propuestas* 1 *hasta la* i, *sin que la suma de sus desagrados supere el monto* m*"*

¿Cuál es la llamada o expresión con esta función que me va a obtener la solución que me piden en el enunciado?
propuestas(N,K,M)

Definamos la función "propuestas":
propuestas(i,k,m) =

- Si i = 0, k = 0 ----> 0
- i = 0, k > 0 ----> - ∞
- i > 0, k = 0 ----> 0 *{- equivalentemente: propuestas(i-1, k, m) -}*
- i > 0, k > 0, q_i > m ----> propuestas(i-1,k,m)
- i > 0, k > 0, q_i <= m ----> max (p_i + propuestas(i-1, k-1, m-q_i) , propuestas(i-1,k,m))

EJERCICIO: Pasar a Programación Dinámica

Examen Final de Práctica (4 julio 2017). Ejercicio 1

Procedimiento **p**.
* ¿qué hace? *Dado un arreglo de naturales, lo ordena de manera creciente.*
* ¿cómo lo hace? *Busca la posición del mínimo elemento y luego lo intercambia por el elemento de la primera posición. Luego busca el siguiente mínimo elemento y lo intercambia por el que está en la segunda posición, y así sucesivamente.*
* Orden: n^2
* p es selection sort.

Función **f**.
* ¿qué hace? *Dado un arreglo de naturales, devuelve otro con los índices en donde estarían los elementos del primer arreglo, si estuviera ordenado de manera creciente. Es decir, para todo índice i entre 1 y n-1 vale que a[b[i]] <= a[b[i+1]].*
* ¿cómo lo hace? *En cada paso selecciona la posición del elemento mínimo del arreglo a, pero en vez de hacer intercambios en "a" lo hace en "b", donde están guardadas las posiciones.*
* Orden: n^2
* cambiar nombres.

Examen Final de Práctica (4 julio 2017). Ejercicio 3

- n habitaciones, consecutivas de 1 a n.
- Carrito con capacidad de 5 desayunos.
- Nos dicen cuántos desayunos hay que dar a cada habitación.
- Se debe dar el orden en que se entregan los desayunos de manera de recorrer la mínima distancia, entregando todo.

n = 3.
d_1 = 2
d_2 = 1
d_3 = 4

Criterio propuesto: Voy primero a la habitación con menor cantidad de pedidos. Siempre cargo el carrito al máximo.

Primer viaje:
Voy hasta habitación 2, completo pedido. Luego desde ahí hasta la 1, completo pedido. Luego desde 1 a 3, me van a faltar 2 pedidos más.

Segundo viaje:
Voy hasta habitación 3 y completo.

Distancias: De 0 a 2, de 2 a 1, de 1 a 3, de 3 a 0, de 0 a 3, de 3 a 0.
TOTAL = 14

Examen Final de Práctica (4 julio 2017). Ejercicio 3

- n habitaciones, consecutivas de 1 a n.
- Carrito con capacidad de 5 desayunos.
- Nos dicen cuántos desayunos hay que dar a cada habitación.
- Se debe dar el orden en que se entregan los desayunos de manera de recorrer la mínima distancia, entregando todo.

n = 3.
d_1 = 2
d_2 = 1
d_3 = 4

Criterio propuesto: Atiendo primero los pedidos de las habitaciones más cercanas.

Primer viaje:
Voy hasta habitación 1, completo pedido. Luego de la 1 a la 2, completo pedido. Luego de 2 a 3 y completo una parte.

Segundo viaje:
Voy hasta habitación 3 y completo.

Distancias: De 0 a 1, de 1 a 2, de 2 a 3, de 3 a 0.
De 0 a 3, de 3 a 0.
TOTAL = 12

Examen Final de Práctica (4 julio 2017). Ejercicio 3

- n habitaciones, consecutivas de 1 a n.
- Carrito con capacidad de 5 desayunos.
- Nos dicen cuántos desayunos hay que dar a cada habitación.
- Se debe dar el orden en que se entregan los desayunos de manera de recorrer la mínima distancia, entregando todo.

n = 3.
d_1 = 2
d_2 = 1
d_3 = 4

Criterio óptimo: Atiendo primero los pedidos de las habitaciones más lejanas.

Primer viaje:
Voy hasta habitación 3, completo pedido.
Luego desde 3 a 2, completo pedido.

Segundo viaje:
Voy hasta habitación 1 y completo pedido.

Distancias: De 0 a 3, de 3 a 2, de 2 a 0. De 0 a 1, de 1 a 0.
TOTAL = 8

Examen Final de Práctica (4 julio 2017). Ejercicio 3

Elegimos tipos de datos.

Propuesta 1: Devuelvo lista de pares indicando número de habitación y cantidad de pedidos entregados. No tengo directamente cuántos viajes hice, ni qué distancia recorrí.

Propuesta 2: Lista de listas de pares. En cada lugar de la lista “grande” represento un viaje desde la cocina. Cada tupla de la lista “de adentro” dice qué pedidos entregué en ese viaje.

```
type Entrega = tuple
                 hab : nat
                 cant : nat
               end
```

```
fun desayunos(p : array[1..n] of nat) ret r : List of (List of Entrega)
```

Examen Final de Práctica (4 julio 2017). Ejercicio 3

```
fun desayunos(p : array[1..n] of nat) ret r : List of (List of Entrega)
  var en_carro : nat
  var p_rest : array[1..n] of nat
  var viaje_actual : List of Entrega
  var entrega_actual : Entrega

  for i:=1 to n do p_rest[i] := p od
  en_carro := 5
  r := empty_list()
  viaje_actual := empty_list()
  addr(r,viaje_actual)

  for h := n downto 1 do
    do (p_rest[h] > 0) →
      if (en_carro >= p_rest[h]) then
        entrega_actual.hab := h
        entrega_actual.cant := p_rest[h]
        addr(viaje_actual,entrega_actual)
        p_rest[h] := 0
      else
        entrega_actual.hab := h
        entrega_actual.cant := en_carro
        p_rest[h] := p_rest[h] - en_carro
        addr(viaje_actual,entrega_actual)
        viaje_actual := empty_list()

      fi
```

Examen Final de Práctica (4 julio 2017). Ejercicio 4

- Tenemos n MONEDAS con denominaciones d1,d2,...,dn.
- Tenés que pagar un valor C por el café, y la suma total de los valores de las monedas alcanza.
- Se pide encontrar el MENOR MONTO a pagar que sea mayor o igual a C.

n = 5. C = 9
d1 = 2
d2 = 2
d3 = 1
d4 = 6
d5 = 5

Backtracking va a probar todas las formas posibles de llegar a un valor mayor o igual a 9 con las monedas que tengo.

Posibles valores con los que puedo pagar:

10 (d1 + d2 + d4)
9 (d1 + d3 + d4)
15 (d1 + d2 + d4 + d5)
16 (d1 + d2 + d4 + d5)
.....

9 (d1 + d2 + d5)

....

Examen Final de Práctica (4 julio 2017). Ejercicio 4

Especifiquemos la función que obtendrá el resultado al problema.

menor_monto(i,j) = “Menor monto posible a pagar utilizando las monedas con denominaciones d1,....,di, tal que la suma de las denominaciones de las monedas elegidas sea mayor o igual a j“

La llamada principal es
menor_monto(n,C)

menor_monto(i,j) =

- Si j <= 0 -----> 0
- Si i = 0, j > 0 -----> +∞
- Si i > 0, j > 0 -----> min (d_i + menor_monto(i-1, j - d_i) , menor_monto(i-1 , j))

Consulta 7/12/2020 - Práctico 3.4 - Ejercicios 1 y 2

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

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

Parámetros de la función:

- d_1, ... , d_n denominaciones
- monto k a pagar
- llamada principal: **cambio(n, k)**

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

```
fun cambio(d : array[1..n] of nat, k : nat) ret r : nat
  var cam: array[0..n,0..k] of nat  // n+1 filas, k+1 columnas
  for i := 0 to n do cam[i, 0] := 0 od
  for j := 1 to k do cam[0, j] := inf od
  for i := 1 to n do                # llenamos cada fila
    for j := 1 to k do              # en fila i, llenar columnas
      min_cam := inf
      for q := 0 to (j % d[i]) do
        min_cam := min(min_cam, q + cam[i-1, j-q*d[i]])
      od
      cam[i, j] := min_cam
    od
  od
  r := cam[n, k]
end fun
```

Orden de llenado del algoritmo

0 (0)	**inf (3)**	inf (4)	inf (5)	inf (6)
0 (1)	**??? (7)**	??? (8)	??? (9)	??? (10)
0 (2)	??? (11)	??? (12)	??? (13)	??? (14)

Pregunta: **Funcionaría llenando por columnas primero?**

(i.e. invirtiendo el orden de los dos for)

Funcionaría llenando por columnas primero? Sí, se puede invertir el orden de los for.

0 (0)	**inf (3)**	inf (4)	inf (5)	inf (6)
0 (1)	??? (7)	??? (9)	??? (11)	??? (13)
0 (2)	??? (8)	??? (10)	??? (12)	??? (14)

Ejercicio 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.

Respuesta: No se puede porque para llenar una entrada necesitamos tener los valores de la fila anterior desde la misma columna hasta la columna 0.

Práctico 1.3 - Ejercicio 3

```
fun minimo(a : array[1..n] of nat, i, k : nat)  ret m : nat
    if i = k then m := a[i]
    else
        j := (i + k) div 2
        m := min(minimo(a, i, j), minimo(a,j+1,k))
    fi
end fun
```

```
fun DyV(x) ret y
    if x suficientemente pequeño o simple then y:= ad_hoc(x)
    else descomponer x en x1, x2, ..., xa
        for i:= 1 to a do yi:= DyV(xi) od
        combinar y1, y2, ..., ya para obtener la solución y de x
    fi
end fun
```

- a : número de llamadas recursivas: **a = 2**.
- b : relación entre el tamaño de x y el de las x_i. b = |x| / |x_i|. **b = 2.**
- k : orden de descomponer y combinar es **n^k. posibilidades: k = 0 (orden 1) k = 1 (orden n) k = 2 (orden n^2) .... SIEMPRE ES UN MIN ENTRE DOS NÚMEROS SIN IMPORTAR N. LUEGO orden 1, o sea k = 0.**

Práctico 1.3 - Ejercicio 3

$$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

Con a = 2, b = 2 y k = 0, tenemos que a > b^k. Luego el orden es

n ^ (log_b a) = n ^ 1 = n

Respuesta final: **orden n (lineal)**.

Examen Final de Práctica (24 julio 2019). Ejercicio 4

4. En una localidad cordobesa, vive el José Agustín Goytisolo quien, apenado por los padecimientos de la mayoría de los vecinos, decide comprometerse en solucionarlos postulándose a la intendencia. Gracias a su entusiasmo y creatividad, en pocos minutos enumera una larga lista de N propuestas para realizar. Pronto descubre que a pesar de que cada una de ellas generaría una satisfacción popular $p_1, p_2, \ldots, p_N$ también provocaría desagrado $q_1, q_2, \ldots, q_N$ en el sector más acomodado de la sociedad local. En principio, el desagrado de cada propuesta es insignificante en número de votos ya que la alta sociedad no es muy numerosa. Pero a José le interesa cuidar su relación con este sector, ya que el mismo tiene suficientes recursos como para dificultar su triunfo en caso de proponérselo.

   Pronto descubre que las propuestas elaboradas son demasiadas para ser publicitadas: tantas propuestas (N) generarían confusión en el electorado. Esto lo lleva a convencerse de seleccionar solamente K de esas N propuestas ($K \leq N$). Se dispone, entonces, a seleccionar K de esas N propuestas de forma tal que la suma de satisfacción popular de las K propuestas elegidas sea máxima y que el descontento total de esas K propuestas en la alta sociedad no supere un cierto valor M.

   José Agustín te contrata para que desarrolles un algoritmo capaz de calcular el máximo de satisfacción popular alcanzable con K de esas N propuestas sin que el descontento supere M.

Examen Final de Práctica (24 julio 2019). Ejercicio 4

- N propuestas de gobierno.
- Cada propuesta i, genera satisfacción p_i, y desagrado q_i.
- Debe elegir K propuestas, entre las N.
- Se deben seleccionar K propuestas con satisfacción máxima, sin que el desagrado total supere un monto M.

Ejemplo. N = 4. K = 2. M = 45.
p_1 = 30, q_1 = 24
p_2 = 20, q_2 = 10
p_3 = 50, q_3 = 32
p_4 = 25, q_4 = 18

Es backtracking, así que debo “probar” con todos los subconjuntos de K propuestas cuya suma total de desagrado no supere M.

Una opción serán las propuestas 1 y 2, cuya popularidad total será de 50 y el desagrado de 34.
Las propuestas 1 y 3 no pueden ser elegidas juntas, pues su desagrado supera M.
Otra opción válida son las propuestas 1 y 4, cuya popularidad es 55, y el desagrado 42.

IMPORTANTE: No pide que minimicemos el desagrado. Solo debemos asegurar que no supera M.
Pide que maximicemos la suma total de satisfacción, siempre y cuando el desagrado no supere M.

Examen Final de Práctica (24 julio 2019). Ejercicio 4

Voy a definir una función recursiva

propuestas(i,k,m) = *"La mayor satisfacción popular obtenible al elegir* k *propuestas de entre las propuestas* 1 *hasta la* i, *sin que la suma de sus desagrados supere el monto* m*"*

¿Cuál es la llamada o expresión con esta función que me va a obtener la solución que me piden en el enunciado?
propuestas(N,K,M)

Definamos la función "propuestas":
propuestas(i,k,m) =

- Si i = 0, k = 0 ----> 0
- **i = 0, k > 0** **----> - ∞**
- i > 0, k = 0 ----> 0 {- se junta con la 1ra: **i >= 0, k = 0** -}
- i > 0, k > 0, q_i > m ----> propuestas(i-1,k,m)
- i > 0, k > 0, q_i <= m ----> max (p_i + propuestas(i-1, k-1, m-q_i) , propuestas(i-1,k,m))

EJERCICIO: Pasar a Programación Dinámica

Examen Final de Práctica (24 julio 2019). Ejercicio 4

```
fun propuestas(p : array[1..N], q : array[1..N], K, M : nat) ret r : nat
  var prop : array[0..N,0..K,0..M] of nat
  for i := 0 to N do
    for m := 0 to M do
      prop[i,0,m] := 0
    od
  od
  for k := 1 to K do
    for m := 0 to M do
      prop[0,k,m] := -inf
    od
  od

  { SIGUE ----> }
```

Examen Final de Práctica (24 julio 2019). Ejercicio 4

```
{ ←- CONTINUACIÓN }

  for i := 1 to N do
    for k := 1 to K do
      for m := 0 to M do
        if q[i] > m then
          prop[i,k,m] := prop[i-1,k,m]
        else
          prop[i,k,m] := max(p[i] + prop[i-1,k-1,m-q[i]], prop[i-1,k,m])
        fi
      od
    od
  od
  r := prop[N,K,M]
end fun
```