# Pizarra de Prácticos y Consultas

Algoritmos II

# Práctico 2021-06-07

Algoritmos II

# Práctico 3.3 - Ejercicio 5

Tenemos:

- n amigos, cada uno tiene un día de partida $p_i$, un día de regreso $r_i$, y paga $m_i$ por día.

Ejemplo: $n = 6$

$p1=1$ $r1=2$ $m1=3$
$p2=3$ $r2=4$ $m2=2$
$p3=4$ $r3=7$ $m3=5$
$p4=2$ $r4=8$ $m4=6$
$p5=6$ $r5=8$ $m5=8$
$p6=4$ $r6=8$ $m6=5$

A diferencia de el práctico 3.1, acá debo probar todas las combinaciones posibles.

# Práctico 3.3 - Ejercicio 5

- Tenemos: n amigos, cada uno tiene un día de partida p_i, un día de regreso r_i, y paga m_i por día.
- Queremos: maximizar la plata ganada.

1. Debemos definir una función recursiva que me calcule soluciones parciales al problema.

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

2. Llamada principal: rataColuda(0)
3. Definición recursiva de la función (o sea, el programa).

# Práctico 3.3 - Ejercicio 5

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

2. Definición recursiva de la función (o sea, el programa).

rataColuda(d) = ( si $\forall$i: p_i < d → 0

| si no → rataColuda(d+1) {- el dia d no lo presto -}

max

maximo_{i tal que p_i = d} (

m_i * (r_i - p_i + 1) + rataColuda(r_i+1) )

{- se lo presto al amigo i -}

)

# Práctico 3.3 - Ejercicio 5

p1=**1** r1=2 m1=3
p2=**3** r2=4 m2=2
p3=**4** r3=7 m3=5
p4=**2** r4=8 m4=6
p5=**6** r5=8 m5=8
p6=**4** r6=8 m6=5

rataColuda(**0**)
= rataColuda(1) {- no se va nadie el dia 0 -}
= rataColuda(2) {- no lo alquilo -}
max
(2*3 + rataColuda(3)) {- lo alquilo al 1 -}
----------------------
rataColuda(2) = rataColuda(3) max (7*6 + rataColuda(9) )
---------------------- (LLAMADAS DUPLICADAS)

rataColuda(3) = rataColuda(4) max (2*2 + rataColuda(5))
----------------------
rataColuda(4) = rataColuda(5) max
(4*5 + rataColuda(8)) max {- amigo 3 -}
(5*5 + rataColuda(9)) {- amigo 6 -}
----------------------
rataColuda(5) = rataColuda(6)
----------------------
rataColuda(6) = rataColuda(7) max (3*8 + rataColuda(9))
----------------------
rataColuda(7) = 0
rataColuda(8) = 0
rataColuda(9) = 0

Ya llegamos hasta las hojas de este árbol de llamadas recursivas. Ahora resolvemos hacia

# Práctico 3.3 - Ejercicio 5

p1=**1** r1=2 m1=3
p2=**3** r2=4 m2=2
p3=**4** r3=7 m3=5
p4=**2** r4=8 m4=6
p5=**6** r5=8 m5=8
p6=**4** r6=8 m6=5

**RESULTADO FINAL: 42.**

**El sentido de la vida, el universo y todo lo demás**

rataColuda(**0**)
= rataColuda(1) {- no se va nadie el dia 0 -}
= 42 {- amigo 4 -}
max
(2*3 + 28) {- amigos 1, 2 y 5 -}
= **42 {- amigo 4 -}**

----------------------

rataColuda(2) = 28 {- amigos 2 y 5 -}
max 42 {- amigo 4 -}
= 42 {- amigo 4 -}

----------------------

rataColuda(3) = 25 {- amigo 6-}
max (2*2 + 24) {- amigos 2 y 5 -}
= 28 {- amigos 2 y 5 -}

----------------------

rataColuda(4) = 24 max {- amigo 5 -}
(4*5 + 0) max {- amigo 3 -}
(5*5 + 0) {- amigo 6 -}
= 25 {- amigo 6 -}

----------------------

rataColuda(5) = 24 {- amigo 5-}

----------------------

rataColuda(6) = 0 max (3*8 + 0)
= 24 {- amigo 5 -}

----------------------

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

# Práctico 3.3 - Ejercicio 5: Reflexiones finales

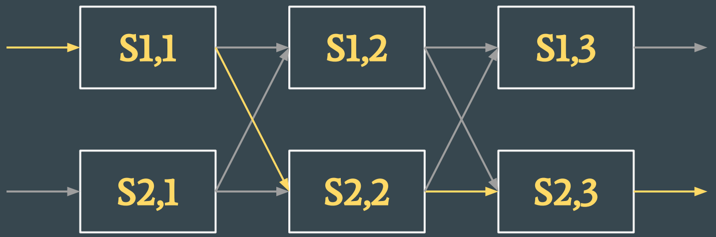
¿Qué complejidad algorítmica tiene este programa? En cada paso tengo posiblemente varias llamadas recursivas (al estilo fib que tiene siempre dos). Esto me crea un arbol de llamadas que crece exponencialmente.

¿Se puede hace más rápido? Sí, hay muchas llamadas en ese árbol que están duplicadas. Eso hicimos en nuestras cuentas y aceleramos mucho el proceso.

¿A qué complejidad pudimos bajarlo? A lineal. Hicimos un cómputo por cada día.

# 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
(no necesariamente es la solución)

# Práctico 2021-06-09

# Práctico 3.4 - Ejercicio 4: Versión PD del ej. 5 del Pr. 3.3

Llamada principal: rataColuda(0)

Definición recursiva:

rataColuda(d) = ( si $\forall$i: p_i < d → 0 {- CASO BASE -}
| si no → rataColuda(d+1) {- el dia d no lo presto -}
max
maximo_{i tal que p_i = d} (
**m_i * (r_i - p_i + 1) + rataColuda(r_i+1)** )
{- se lo presto al amigo i -}

)

```
fun rataColuda(p : arr[1..n] of nat, r : arr[1..n] of nat,
               m : arr[1..n] of nat,
               ultima_partida : nat, ultimo_dia : nat) ret gano : nat
  var tabla : array[0..ultimo_dia] of nat
  var aux : nat
  {- CASOS BASE -}
  for d := ultima_partida+1 to ultimo_dia do
    tabla[d] := 0
  od
  {- CASO RECURSIVO -}
  for d := ultima_partida downto 0 do
    aux := 0  {- aux: “maximo que puedo ganar prestando hoy d” -}
    for i := 1 to n do
      if p[i] == d do
        aux := aux max (m[i] * (r[i] - p[i] + 1) + tabla[r[i]+1])
      od
    od
    tabla[d] := tabla[d+1] max aux
  od
  gano := tabla[0]
```

# Comentarios

- Usamos dos parámetros extra por comodidad: ultima_partida y ultimo_dia.
- ultima partida es el ultimo dia de partida del amigo que parte ultimo.
- ultima_dia es el ultimo dia de regreso del amigo que regresa ultimo + 1.
- A diferencia de los problemas de la mochila y la moneda:
- Llenamos el arreglo de derecha a izquierda
- Tenemos una cantidad no fija de llamadas recursivas para las cuales necesitamos un for (con un if adentro) que calcula un máximo.
- Así como está, la complejidad es: ultima_partida * n

# Ejemplo:

p1=**1** r1=2 m1=3

p2=**3** r2=4 m2=2

p3=**4** r3=7 m3=5

p4=**2** r4=8 m4=6

p5=**6** r5=8 m5=8

p6=**4** r6=8 m6=5

ultima_partida=6

ultimo_dia=9

casos base

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 24 | 0 | 0 | 0 |

aux := maximo_{i tales que p[i] = 6}  (m[i] * (r[i]-p[i]+1) + tabla[r[i]+1]) = (m[5] * (r[5]-p[5]+1) + tabla[r[5]+1])

= 8*3 + tabla[9] = 24

tabla[6] := tabla[7] max aux = 0 max 24 = 24

# Ejemplo:

p1=**1** r1=2 m1=3

p2=**3** r2=4 m2=2

p3=**4** r3=7 m3=5

p4=**2** r4=8 m4=6

p5=6 r5=8 m5=8

p6=**4** r6=8 m6=5

ultima_partida=6

ultimo_dia=9

casos base

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | 24 | 24 | 0 | 0 | 0 |

aux := maximo_{i tales que p[i] = 5}  (m[i] * (r[i]-p[i]+1) + tabla[r[i]+1]) = 0

tabla[5] := tabla[6] max aux = tabla[6]

# Ejemplo:

p1=**1** r1=2 m1=3

p2=**3** r2=4 m2=2

**<u>p3=4 r3=7 m3=5</u>**

p4=2 r4=8 m4=6

p5=6 r5=8 m5=8

**<u>p6=4 r6=8 m6=5</u>**

ultima_partida=6

ultimo_dia=9

casos base

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 25 | 24 | 24 | 0 | 0 | 0 |

aux := maximo_{i tales que p[i] = 4}  (m[i] * (r[i]-p[i]+1) + tabla[r[i]+1]) =   {-  i es 3 o 6 -}

(5 * 4 + tabla[8]) max (5 * 5 + tabla[9]) = 20 max 25 = 25

tabla[4] := tabla[5] max aux = 24 max 25 = 25

# Práctico 2021-06-14

# Práctico 3.4 - Ejercicio 1

$$
\text{cambio}(i, j) = \begin{cases} 0 & \text{si } j = 0 \\ \infty & \text{si } j > 0 \text{ e } i = 0 \\ \min_{q \text{ en } 0, 1, \dots, \underline{\mathbf{j / d_i}}} (q + \text{cambio}(i - 1, j - q * d_i)) & \text{si } j > 0 \text{ e } i > 0 \end{cases}
$$

- Denimonaciones: $d_1, \dots , d_n$.
- Dar cambio por un monto total $k$.

<u>**Llamada principal:**</u> cambio(n, k).

En esta versión, “cambio” en una sola recursión considera todas las cantidades posibles de monedas de una denominación fija (incluso cantidad cero).

<u>**Ejemplo:**</u> $d_1=25$, $d_2 = 50$ || $\text{cambio}(2, 200) = \min_{q \text{ en } 0, 1, 2, 3, 4 = j / d_2}$ ......

# Pasamos a programación dinámica

¿Qué forma tiene la tabla que voy a llenar? Es una tabla de (n+1) x (k+1) tamaño.
¿En qué orden la tengo que llenar? Las filas de arriba hacia abajo, las columnas no importa el orden.

> ¿Porqué? Para llenar cambio(i, j) debo mirar elementos en posiciones cambio(<u>**i - 1**</u>, j - q * d_i), o sea siempre en la **fila de arriba a la actual**.

Tipo de la función:


fun cambio(d : array[1..n] of nat, k : of nat) ret r : nat

# Pasamos a programación dinámica

```
fun cambio(d : array[1..n] of nat, k : of nat) ret r : nat
  var tabla : array[0..n,0..k] of nat
  var mimin : nat
  {- casos base -}
  for i := 0 to n do
    tabla[i, 0] := 0
  od
  for j := 1 to k do
    tabla[0, j] := infinito
  od
  {- caso recursivo -}
  ....
```

# Pasamos a programación dinámica

```
  ...
  {- caso recursivo -}
  for i := 1 to n do
    for j := 1 to k od
      {- acá hay que calcular un min con varios q posibles -}
      mimin := infinito
      for q := 0 to (j / d[i]) do
        mimin :=  mimin `min` (q + tabla[i-1, j-q*d[i]])
      od
      tabla[i, j] := mimin
    od
  od
  {-  ahora, tabla[i, j] = cambio(i, j) para todo i, j  -}
  r := tabla[n, k]
end fun
```

# Parte 2: Calculamos la solución además del mínimo

Vamos a necesitar otra tabla para ir guardando las soluciones para todos los i, j posibles. Tendremos:

tabla[i, j] = cambio(i, j)

solucion[i, j] = “lista de monedas correpondiente al problema que minimiza cambio(i, j).”

(con solucion[i, j] solo alcanzaria ya que los elements de tabla se puede calcular a partir de los de solucion).

# Versión con cálculo de la solución

```
fun cambio(d : array[1..n] of nat, k : of nat)
           ret r : List of nat
  var tabla : array[0..n,0..k] of nat
  var solucion : array[0..n,0..k] of (List of nat)
  var mimin, q_min : nat
  {- casos base -}
  for i := 0 to n do
    tabla[i, 0] := 0
    solucion[i, 0] := empty_list()
  od
  for j := 1 to k do
    tabla[0, j] := infinito  {- no hay solucion -}
  od
  {- caso recursivo -}
```

# Pasamos a programación dinámica

```
...
{- caso recursivo -}
for i := 1 to n do
  for j := 1 to k od
    {- acá hay que calcular un min con varios q posibles -}
    mimin := infinito
    for q := 0 to (j / d[i]) do
      if q + tabla[i-1, j-q*d[i]] < mimin:
        mimin :=  q + tabla[i-1, j-q*d[i]]
        q_min := q
    od
    tabla[i, j] := mimin
    ...
```

# Pasamos a programación dinámica

```
...
    {- quiero poner q veces d[i] para el q elegido,
       y ademas los elementos de solucion[i-1,j-q*d[i]]. -}
    if mimin < infinito then
        solucion[i, j] := armar_solucion(q_min, d[i],
                                         solucion[i-1,j-q_min*d[i]])
    fi
  od
od
{-  ahora, tabla[i, j] = cambio(i, j) para todo i, j  -}
{-  y solucion[i, j] tiene la solucion correpondiente -}
if tabla[n, k] < infinito then
  r := copy_list(solucion[n, k])
fi
    {- si no hay solucion no sabemos qué se devuelve -}
```

# Pasamos a programación dinámica

```
  {- ACÁ FALTA LIBERAR MEMORIA!! -}
  for i := 0 to n do
    for j := 0 to k do
      if tabla[i, j] < infinito then
        destroy_list(solucion)
      fi
    od
  od
end fun
```

**<u>Pregunta:</u>** ¿puede suceder que para calcular solucion[i, j] quiera usar yo otro valor solucion[i-1, j-...] que no haya sido inicializado? NO porque nos cuidamos de eso.

# Pasamos a programación dinámica

```
fun armar_solucion(q: nat, d: nat, sol_ant: List of nat)
                   ret sol: List of nat
  {- copiamos la solucion anterior y agregamos q monedas de
     denomiacion d (podría ser q=0) -}
  sol := copy_list(sol_nat)
  for i := 1 to q do
    addr(sol, d)
  od
end fun
```

# Práctico 2021-06-16

# TP Entregable: Ejercicio 2

Ejemplo:

- 4 oficinas.
- Costos $C_1=10$, $C_2=4$, $C_3=30$.
- Preferencias:

| p^i_j (oficina \ color) | 1 (rojo) | 2 (blanco) | 3 (verde) |
|---|---|---|---|
| 1 | 12 | 6 | 23 |
| 2 | -1 | 2 | 9 |
| 3 | 24 | 12 | 4 |
| 4 | 0 | 6 | 34 |

- Ejemplo: pintamos así: la of. 1 con el color 3, la 2 con el color 1, la 3 con el 3 y la 4 con el 2. Relación preferencia / costo:

# TP Entregable: Ejercicio 2

Ejemplo:

- Ejemplo: pintamos así: la of. 1 con el color 3, la 2 con el color 1, la 3 con el 3 y la 4 con el 2. Relación preferencia / costo:
  p^1_3 / C_3 + p^2_1 / C_1 + p^3_3 / C_3 + p^4_2 / C_2
  = 23 / 30 + (-1) / 10 + 4 / 30 + 6 / 4 = “algo” (completar)
  (esta es una forma de pintar, no necesariamente la que maximiza el objetivo)

| p^i_j (oficina \ color) | 1 (rojo) | 2 (blanco) | 3 (verde) |
|---|---|---|---|
| 1 | 12 | 6 | 23 |
| 2 | -1 | 2 | 9 |
| 3 | 24 | 12 | 4 |
| 4 | 0 | 6 | 34 |

# Práctico 3.3 - Ejercicio 7: Dos mochilas

Recordemos como era el de una mochila:
- Mochila con capacidad W
- n objetos con valores v_1, ... , v_n y pesos w_1, ... , w_n
- llenar la mochila con objetos logrando el máximo valor posible.

```
mochila(i, j) = ( 0                                          →  si j = 0 ó i = 0
                | mochila(i-1, j)                            →  si w_i > j
                | max(mochila(i-1, j), v_i + mochila(i-1, j - w_i))   → si w_i <= j
                )
```

Llamada principal: mochila(n, W)

# Práctico 3.3 - Ejercicio 7: Dos mochilas

Ahora veámoslo con dos mochilas:
- Mochilas con capacidad W1 y W2.
- n objetos con valores v_1, ... , v_n y pesos w_1, ... , w_n
- llenar la mochila con objetos logrando el máximo valor posible.

**Función recursiva:** 2mochilas(i, j, k) = "máximo valor posible al guardar algunos objetos 1 ... i en dos mochilas con capacidad restante j (la 1ra) y k (la 2da)."

**Llamada principal:** mochila(n, W1, W2)

# Práctico 3.3 - Ejercicio 7: Dos mochilas

**Definición recursiva:**

```
mochila(i, j, k) = ( 0                          →  si i = 0
                   | 0                          →  si j = 0 y k = 0
                   | mochila(i-1, j, k)         →  si w_i > j y w_i > k
                          {- no entra en ninguna de las dos mochilas -}
              | max(mochila(i-1, j, k), v_i + mochila(i-1, j - w_i, k)) → si w_i <= j y w_i > k
                          {- entra sólo en la mochila 1 -}
              | max(mochila(i-1, j, k), v_i + mochila(i-1, j, k - w_i)) → si w_i > j y w_i <= k
                          {- entra sólo en la mochila 2 -}
              | max(mochila(i-1, j, k), v_i + mochila(i-1, j - w_i, k),
                                 v_i + mochila(i-1, j, k - w_i)) →si w_i <= j y w_i <= k
                          {- entra en ambas mochilas -}
              )
```

# Pasaje a Programación Dinámica con 2 mochilas

¿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 pisos (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
  var tabla : array[0..n,0..W1,0..W2] of nat
  {- casos base -}
  for j := 0 to W1 do
    for k := 0 to W2 do
      tabla[0, j, k] := 0  {- planta baja -}
    od
  od
  for i := 0 to n do
    tabla[i, 0, 0] := 0    {- columna 0,0 de todo el edificio -}
  od
  ....
```

```
  ....
  for i := 1 to n do    {- casos recursivos -}
    for j := 0 to W1 do
      for k := 0 to W2 do
        if j == 0 and k == 0 then
          skip  {- tabla[i, 0, 0] era caso base -}
        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(tabla[i-1, j, k], v[i] + tabla[i-1, j-w[i], k])
        else if w[i] > j and w[i] <= k then
          tabla[i, j, k] := max(tabla[i-1, j, k], v[i] + tabla[i-1, j, k-w[i]])
        else if w[i] <= j and w[i] <= k then
          tabla[i, j, k] := max(tabla[i-1, j, k], v[i] + tabla[i-1, j-w[i], k]
                                                , v[i] + tabla[i-1, j, k-w[i]])
      od
    od
  od
  {- devolver llamada principal -}
  r := tabla[n, W1, W2]
end fun
```

```
{- VERSIÓN NO TAN LARGA -}
fun 2mochilas(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
  var mimax : nat
  for j := 0 to W1 do
    for k := 0 to W2 do
      tabla[0, j, k] := 0  {- planta baja -}
    od
  od
  ....
```

```
  ....
  for i := 1 to n do    {- casos recursivos -}
    for j := 0 to W1 do
      for k := 0 to W2 do
        if j == 0 and k == 0 then
          tabla[i, 0, 0] := 0  {- columna 0,0 de todo el edificio -}
        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
  {- devolver llamada principal -}
  r := tabla[n, W1, W2]
end fun
```

# Consulta 5/7/2021

•••

# Ejercicio 2 - Parcial Viejo

El profe de algoritmos 2 tiene n medias diferentes, con n n´umero par (digamos n = 2m). Hay una tabla P[1..n, 1..n] tal que P[i, j] es un n´umero que indica cu´an parecida es la media i con la media j. Tenemos P[i, j] = P[j, i] y P[i, i] = 0. Dar un algoritmo que determine la mejor manera de aparear las n medias en m pares. La mejor manera significa que la suma total de los P[i, j] lograda sea lo mayor posible. Es decir, si decidimos aparear i1 con j1, i2 con j2, . . . . im con jm, la sumatoria P[i1, j1]+P[i2, j2]+...P[im, jm] debe ser lo mayor posible. Un apareamiento debe aparear exactamente una vez cada media.

- n medias (con n par)
- valores P[i, j] indicado parecido entre media i y media j
- queremos armar m (= n / 2) pares de medias maximizando la suma de parecidos.

# Ejercicio 2 - Parcial Viejo

- n medias (con n par)
- valores P[i, j] indicado parecido entre media i y media j
- queremos armar m (= n / 2) pares de medias maximizando la suma de parecidos.

Función recursiva: aparear(S) = “suma de parecidos del mejor apareamiento considerando medias sacadas del conjunto S.”

Llamada principal: aparear({1, 2, ... , n})

Definición recursiva:

aparear(S) = ( 0 → si S es vacío

| max_{i, j en S tal que i != j} P[i,j] + aparear(S / {i, j}) → si S es no vacío

)

# Ejercicio 3 - 8/2/2021

**(a)** s(v, p) -- devuelve el índice del primer elemento de p tal que es mayor que el siguiente, partiendo de la posición v. **PRE: v > 0**

t(p) -- devuelve la cantidad de segmentos de largo máximo que están ordenados de manera creciente dentro del arreglo.

u(p) -- me dice si el arregló está ordenado de manera creciente

# Ejercicio 3 - 8/2/2021

**(b)** s(v, p) -- recorre el arreglo desde la posición v y avanza hasta que encuentra el primer elemento que es mayor que el siguiente.

t(p) -- recorre el arreglo en segmentos ordenados, usando s para determinar dónde termina cada segmento, contando la cantidad de segmentos recorridos.

u(p) -- llama a t y se fija si el resultado es <= 1 (igual 0 no va a dar nunca)

# Ejercicio 3 - 8/2/2021

**(c)** s(v, p) -- mejor caso: O(1), cuando el primer elemento a considerar es mayor al siguiente. peor caso: O(n - v), cuando está ordenado desde v hasta el final.

t(p) -- mejor y peor caso: O(n), ya que recorre el arreglo desde la posición cero hasta la última posición (por medio de s).

**Ejemplo:** p = [34, 67, -12, 4, 11]  ¿qué devuelve t? 2
¿cuántas llamadas a s hace t? 2, una de 2 pasos y otra de 3 pasos, total: 5 pasos.

u(p) -- igual a la de t(p): O(n)

# Ejercicio 2 - Final 18/6/2018

- Tenemos dados $c_1, ..., c_n, d_1, ..., d_k$.

  $m(i, j) = ( c_i \rightarrow \text{si } j = k$

  $\quad | \; d_j \rightarrow \text{si } i = n \;\text{ y }\; j < k$

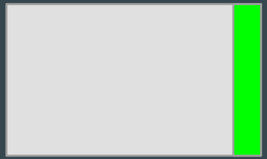
  $\quad | \; m(i, j + 1) + m(i + 1, j)$
- Llamada principal: $m(1, 1)$

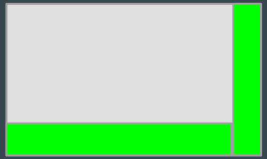
Pasaje a programación dinámica:

- tabla : array[1..n, 1..k] of int.
- orden de llenado: para cada celda, debo tener calulado el valor de la celda a la derecha $(i + 1, j)$ y de la celda de abajo $(j + 1, i)$.

  debería llenar de abajo hacia arriba y de derecha a izquierda.

```
fun m(c : array[1..n] of int, d : array[1..k] of int) ret r : int
  var tabla: array[1..n, 1..k] of int
  for i := 1 to n do
    tabla[i, k] := c[i]
  od
  for j := 1 to k-1 do
    tabla[n, j] := d[j]
  od
  for i := n-1 downto 1 do
    for j := k-1 downto 1 do
      tabla[i, j] := tabla[i, j+1] + tabla[i+1, j]
    od
  od
  r := tabla[1, 1]
end fun
```





# Ejercicio de los Dominós

**Llamada principal:** domino(n, 6)

domino(i, j) =
( si i=0 ---> 0
| si i > 0 ---> maximo_{k = 0...6, k != c_i-1 y k != c_i-2} domino(i-2, k) + P_k,j
)
(definimos artificialmente c_0 = -1).

Programación dinámica:

- tabla: array[0..n, 0..6]
- orden de llenado: de arriba hacia abajo, cada fila no importa en qué orden.

```
fun dominos(P : array[0..6,0..6] of int,
            c : array[1..n] of int) ret r : int
  var tabla : array[0..n,0..6] of int
  var mimax : int

  for j := 0 to 6 do
    tabla[0, j] := 0
  od
...
```

```
...
  for i := 2 to n do
    if i % 2 == 0 then
      for j := 0 .. 6 do
        mimax := -infinito
        for k := 0 .. 6 do
          if k != c[i-1] && k != c[i-2] then
            mimax := mimax ‘max’ (domino[i-2, k] + P[k, j])
          fi
        od
        tabla[i, j] := mimax
      od
    fi
  od
  r := tabla[n, 6]
end fun
```

# Ejercicio 4 - 6/8/2014

- n objetos con pesos p_1 ... p_n
- n cajas de capacidad P
- minimizar el numero de cajas a emplear

**Función recursiva:** m(i, j1, . . . , jn) = “menor número de cajas que se necesitan emplear para embalar los objetos 1, . . . , i cuando las capacidades restantes de las cajas son j1, . . . , jn.”

**Llamada principal:** m(n, P, ..., P)

# Ejercicio 4 - 6/8/2014

**Función recursiva:** m(i, j1, . . . , jn) = "menor número de cajas que se necesitan emplear para embalar los objetos 1, . . . , i cuando las capacidades restantes de las cajas son j1, . . . , jn."

**Definición:** m(i, j1, ..., jn) = ( |{ k : j_k < P, k = 1,...,n }| → si i = 0

# esto es, la cantidad de cajas que no están vacías

| min_{ k = 1, ... , n con j_k >= p_i }

m(i-1, j_1, ...., j_k - p_i, ..., j_n) → si i > 0

# probar metiendo el objeto en

# todas las cajas en las que entra

# Consulta 27/7/2021

# Ejercicio 1 - Final 7/7/2021 Tema 1 - Carrera

- tiempo T fijo para cada cambio de ruedas
- n sets de ruedas
- t_1, ... , t_n tiempos por vuelta para cada set de ruedas
- v_1, ... , v_n vida útil en cant. de vueltas para cada set de ruedas
- m vueltas

Dar tres cosas:

- tipo de la función y especificación (explicación en palabras de lo que calcula)
- llamada principal
- definición recursiva (en notación matemática, **no se usa lenguaje imperativo**)

- Tipo de la función y especificación:

  mejor_tiempo(i, j) = “tiempo de carrera mínimo considerando ~~el juego de ruedas i~~ **los juegos de ruedas 1 , ... , i** para hacer j vueltas.”

- Llamada principal: mejor_tiempo(n, m)

- Definición recursiva:

```
mejor_tiempo(i, j) = ( si j = 0             →  0
                    | si j > 0 & i = 0 →  infinito
                    | si j > 0 & i > 0 →   min(
   T + (v_i ‘min’ j) * t_i + mejor_tiempo(i - 1, max(j - v_i, 0) ),   # usar el i-ésimo set
   mejor_tiempo(i - 1, j)                                              # no usarlo
                                                    )
```

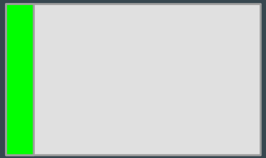
- Otra definición recursiva:

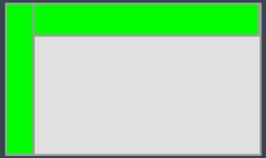
```
mejor_tiempo(i, j) = ( si j = 0              →  0
                     | si j > 0 & i = 0 →  infinito
                     | si j > 0 & i > 0 & v_i > j→     min(
  T + j * t_i ,                                    # usar el i-ésimo set
  mejor_tiempo(i - 1, j)                           # no usarlo
                                                        )
                     | si j > 0 & i > 0 & v_i <= j→   min(
  T + v_i * t_i + mejor_tiempo(i - 1, j - v_i),   # usar el i-ésimo set
  mejor_tiempo(i - 1, j)                           # no usarlo
                                               )
```

# Programación dinámica

- Tabla: 2 dimensiones: una de 0 a n, la otra de 0 a m. O sea (n+1)x(m+1).
- Orden de llenado: las filas de arriba hacia abajo (1ro la fila 0, dps la fila 1, y así), las columnas no importa el orden.

```
fun mejor_tiempo(t : array[1..n] of real, v : array[1..n] of real, T:
real, m: int ) ret r : real
  var tabla : array[0..n,0..m] of real
  {- llenar j = 0 con 0’s -}
  for i := 0 to n do
    tabla[i,0] := 0.0
  od
  {-- llenar j > 0 & i = 0 con infinito --}
  for j := 1 to m do
    tabla[0, j] := infinito
  od
```





```
  for i := 1 to n do
    for j := 1 to m do    {- este for puede ser en cualquier orden -}
      tabla[i,j] := min(
             T + t[i] * (v[i] ‘min’ j) + tabla[i - 1,(j-v[i]) ‘max’ 0] ,
             tabla[i - 1, j] )
    od
  od
  r := tabla[n,m]
end fun
```

# Ejercicio 1 - Final 7/7/2021 Tema 1 - Vacunas

- n personas
- v_1 , ... , v_n en {AZ, SV, PF} indicando la primera dosis de cada persona
- d_k con k en {AZ, SV, PF} indicando la cantida de 2da dosis de cada vacuna
- p_i,j con i,j en {AZ, SV, PF} indica porcentaje de inmunidad que da 1ra dosis i + 2da dosis j.

- Especificación de la función:
  mejor_inmunidad(i , a, s, f) = "máxima suma de porcentajes de inmunidad vacunando a las personas 1 , ... , i con dosis disponibles a, s, f de las vacunas AZ, SV y PF respectivamente."

- Llamada principal: mejor_inmunidad(n, d_AZ , d_SV , d_PF )

- Definición recursiva (solución larga):

```
mi(i , a, s, f) =
        ( i = 0    → 0
        | i > 0 & a = s = f = 0   → -infinito
        | i > 0 & a > 0 & s = f = 0 → p_{v_i,AZ} + mi(i-1, a-1, s, f) # solo tengo AZ
        ...  # salteo solo tengo SV, solo tengo PF
         | i > 0 & a > 0 & s > 0 & f = 0 → max(
                p_{v_i,AZ} + mi(i-1, a-1, s, f) ,  # AZ
                p_{v_i,SV} + mi(i-1, a, s-1, f) )  # SV
         .... # salteo AZ/PF y SV/PF
         | i > 0 & a > 0 & s > 0 & f > 0 → max(
                p_{v_i,AZ} + mi(i-1, a-1, s, f) ,  # AZ
                p_{v_i,SV} + mi(i-1, a, s-1, f) ,  # SV
                p_{v_i,PF} + mi(i-1, a, s, f-1) )  # PF
          )
```

- Definición recursiva (solución más corta):

mi(i , a, s, f) =

( i = 0 → 0

| i > 0 & (a = -1 ó s = -1 ó f = -1) → -infinito

| i > 0 & (a >= 0 | s >= 0 | f >= 0) → max(

p_{v_i,AZ} + mi(i-1, a-1, s, f) , # AZ

p_{v_i,SV} + mi(i-1, a, s-1, f) , # SV

p_{v_i,PF} + mi(i-1, a, s, f-1) ) # PF

)

- en el caso recursivo, usamos sin chequear si quedan o no
- no hay problema si quedan 0 de alguna/s, total se pueden usar las otras
- pero si alguna vale -1, es que usé más de las que tenía entonces da -infinito.

Ejemplo: mi(7, 0, 0, 9) = max(p_x + **mi(-1,0,9)** , p_y + mi(0,-1,9), p_z + mi(0,0,8) )

= max( -infinito, -infinito, p_z + mi(0,0,8) )

= p_z + mi(0,0,8)

# Programación Dinámica

- Tabla:
  - 1ra solución (larga): 4 dimensiones: 0..n , 0...d_AZ, 0...d_SV, 0...d_PF.
  - 2da solución (corta): 4 dimensiones: 0...n, -1...d_AZ, -1...d_SV, -1...d_PF.
- Orden de llenado: llenamos las filas (1ra dimensión) de arriba hacia abajo. para el resto de las dimensiones no importa el orden (ya que siempre voy a mirar la fila anterior que está toda llena) (es lo mismo que con las ruedas).

# Con la versión larga: (igual se acorta en imperativo)

```
fun vacunar(v: array[1..n] of nat, d: array[1..3] of nat, p:
array[1..3,1..3] of real) ret r: real
  var tabla: array[0..n,0..d[1],0..d[2],0..d[3]]
  for a := 0 to d[1] do
    for s := 0 to d[2] do
      for f := 0 to d[3] do
        tabla[0,a,s,f] := 0
      od
    od
  od

  ...
```

```
for i := 1 to n do
  for a := 0 to d[1] do
    for s := 0 to d[2] do
      for f := 0 to d[3] do
        mimax := -infinito
        if a > 0 then
          mimax := mimax max (p[v[i],1] + tabla[i-1,a-1,s,f])
        fi
        if “tengo de SV” then
          mimax := mimax max (?????)
        fi
        if “tengo de PF” then
          mimax := mimax max (?????)
        fi
        tabla[i,a,s,f] := mimax
      od
    od
  od
od
r := tabla[n,d[1],d[2],d[3]]  end fun
```

# Consulta 10/8/2021

# Ejercicio de backtracking del 8/2/2021

2. (Backtracking) Se tienen $n$ objetos de peso $p_1, \ldots, p_n$ respectivamente. Se tiene una mochila de capacidad $K$. Dar un algoritmo que utilice backtracking para calcular el menor desperdicio posible de la capacidad de la mochila, es decir, aquél que se obtiene ocupando la mayor porción posible de la capacidad, sin excederla. Definir primero en palabras la función aclarando el rol de los parámetros o argumentos.

- n objetos de peso p_1 , ... , p_n
- mochila de capacidad K
- calcular menor desperdicio posible de la capacidad de la mochila

Hacer tres cosas:

- especificar la función usando lenguaje natural
- dar llamada principal
- dar definición recursiva

- n objetos de peso $p_1, \ldots, p_n$
- mochila de capacidad K (finita)
- calcular menor desperdicio posible de la capacidad de la mochila

1. especificar la función usando lenguaje natural:

mochila(i,j) = "el menor desperdicio posible de la mochila con capacidad j considerando __1 ... i objetos__" **(mal: el i-ésimo objeto, i objetos) (bien pero distinto: i ... n objetos, (n- i) ... n objetos, etc)**

2. dar llamada principal: mochila(n, K)

3. dar definición recursiva:

mochila(i, j) = ( si j = 0 $\rightarrow$ 0
| si j > 0 y i = 0 $\rightarrow$ j
| si j > 0 y i > 0 y $p_i > j \rightarrow$ mochila(i-1, j)
| si j > 0 y i > 0 y $p_i <= j \rightarrow$ mochila(i-1, j - $p_i$) __**min**__ mochila(i-1, j)

# Pasaje a programación dinámica:

- Dimensiones de la tabla: (n+1)x(K+1)
- Orden de llenado de la tabla: llenaremos las filas de arriba hacia abajo, las columnas de izquierda a derecha. ¿Se puede llenar en otro orden? sí, las columnas dentro de una fila en cualquier orden.
- Código:

```
fun mochila(p: array[1..n] of nat, K: nat) ret r : nat
 var tabla: array[0..n,0..K] of nat
 {- casos base-}
 for i := 0 to n do
   tabla[i, 0] := 0
 od
 for j := 1 to K do
   tabla[0, j] := j
 od    {- SIGUE -}
```

```
  {- recursiones-}
  for i := 1 to n do
    for j := 1 to K do    {- podría ser: for j := K downto 1 do .... -}
      if p[i] > j then
        tabla[i, j] := tabla[i-1, j]
      else    {- acá p[i] <= j -}
        tabla[i, j] := tabla[i-1, j - p[i]]  min  tabla[i-1, j]
    od
  od
  {- devolver llamada princial-}
  r := tabla[n, K]
end fun
```

# Final 29/7/2021

2. Te encontrás frente a una máquina expendedora de café que tiene un letrero que indica claramente que la máquina "no da vuelto". Buscás en tu bolsillo y encontrás exactamente $n$ monedas, con las siguientes denominaciones enteras positivas: $d_1, d_2, \ldots, d_n$. Una rápida cuenta te transmite tranquilidad: te alcanza para el ansiado café, que cuesta $C$. Teniendo en cuenta que la máquina no da vuelto, dar un algoritmo que determine el menor monto posible que sea mayor o igual al precio del café.

- tenemos n monedas con denominaciones d_1 , ... , d_n (enteros positivos)
- café cuesta C
- pagar lo menos posible que sea >= C con mis monedas

1. especificación de la función general:
   cafe(i, j) = "minimo pago >= j usando las monedas 1, ..., i"

2. llamada principal: cafe(n, C)

3. Definición recursiva:

```
cafe(i, j) = ( si j = 0 → 0
             | si j > 0 && i = 0 → infinito
             | si j > 0 && i > 0 →  min(
                  d_i + cafe(i-1, max(0, j - d_i)) ,        {- usamos la i-ésima moneda-}
                  cafe(i-1, j)                              {- no usamos la i-ésima moneda -}
               )
             )
```

**Otra posibilidad:**

```
cafe(i, j) = ( si j = 0 → 0
             | si j > 0 && i = 0 → infinito
             | si j > 0 && i > 0 && j < d_i →  min(d_i ,  cafe(i-1, j))
             | si j > 0 && i > 0 && j >= d_i →min(d_i + cafe(i-1, j - d_i) ,
                                                  cafe(i-1, j) )
             )
```

# 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

| | | | | |
|---|---|---|---|---|
| fila 4 | 2 | 3 | 4 X | 2 |
| fila 3 | 1 | 3 | 4 X | 1 |
| fila 2 | 3 | 2 | 6 X | 1 |
| fila 1 | 3 | 2 | 3 | 4 X |

¿Qué puntaje hice con ese juego? 12

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

1. Especificación de la función general:
   maximo_puntaje(i, j) = "maximo puntaje obtenible partiendo desde la posición i, j, a donde i indica la fila y j la columna."

2. Llamada principal: maximo_puntaje(1, 1) max .... max maximo_puntaje(1, n)

3. Definición recursiva:

```
maximo_puntaje(i, j) = ( i = n → c_{i, j}
                       | i < n → c_{i, j} + max(
                                     maximo_puntaje(i+1, j),
                                     maximo_puntaje(i+1, max(j-1, 0)),
                                     maximo_puntaje(i+1, min(j+1, n)),
                                 )
                        )
```

# Otra solución posible (2020)

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) )