Notas de Consulta Online
Parte 1

Algoritmos y Estructuras de Datos II

Practico 2.1: Ejercicio 5c

```
fun lex_compare(a, b: array[1..n] of nat) ret resultado: ord
  if lex_less(a, b) ->
    resultado := menor
  else if lex_less(b, a) ->
    resultado := mayor
  else ->
    resultado := igual
  end if
end fun
```

Ejercicio 6, práctico 2.1

```
fun sum_matrices(a , b : array[1..n,1..m] of nat) ret r :
array[1..n,1..m] of nat

   for fila := 1 to n do

     for columna := 1 to m do

         r[fila,columna] := a[fila,columna] + b[fila,columna]

     od

   od

 end fun
```

Practico 2.1: Ejercicio 5c

Otra versión:

```
fun lex_compare(a, b: array[1..n] of nat) ret resultado: ord
  resultado := igual
  if lex_less(a, b) ->
    resultado := menor
  else if lex_less(b, a) ->
    resultado := mayor
  end if
end fun
```

Practico 2.1: Ejercicio 5c

ESTO ESTA MAL. PORQUE?:

```
fun lex_compare(a, b: array[1..n] of nat) ret resultado: ord
  resultado := igual
  if lex_less(a, b) ->
    resultado := menor
  else ->
    resultado := mayor
  end if
end fun
```

Practico 2.1: Ejercicio 3(a)

```
fun peso_prom(a : array[1..n] of person) ret  w : float
  var sum : nat
  sum := 0
  for i := 1 to n do
    sum := sum + a[i].weight
  od
  w := sum / n
end fun
```

Ejercicio 2, Práctico 2.2

```
type List of T = tuple
                   elems  : array[1..N] of T
                   size  : nat
                 end tuple


fun empty() ret lista : List of T
   lista.size := 0
end fun
```

Ejercicio 2, Práctico 2.2   (cotinuación)

```
{- PRE : lista.size < N -}
proc addl(in/out lista : List of T, e : T)



end proc
```

Supongamos N = 7

lista.elems = [1,4,2,6,7,3,2]   ---->   [5,1,4,2,7,3,2]

lista.size = 3   ---->   4

Quiero agregar el elemento 5 a la izquierda.

```
{- PRE : lista.size < N -}
proc addr(in/out lista : List of T, e : T)



end proc

Supongamos N = 7

lista.elems = [1,4,2,6,7,3,2]                ---->   [1,4,2,5,7,3,2]

lista.size = 3                               ---->   lista.size = 4
```

Quiero agregar el elemento 5 a la DERECHA.

Practico 2.2 Ejercicio 4

(b)

```
type Tablero = tuple
  equipo_A: Counter
  equipo_B: Counter
end tuple

proc anotar_varios_A(in/out t: Tablero, in n: nat)
  for i := 1 to n do
    {- incr(t): NO porque t es Tablero pero incr toma Counter -}
    incr(t.equipo_A)
  od
end proc
```

Practico 2.2 Ejercicio 4

(b)

```
type Tablero = tuple
  equipo_A: Counter
  equipo_B: Counter
end tuple

proc castigar_A(in/out t: Tablero, in n: nat)
  for i := 1 to n do
    {- OJO: debe valer not is_init(t.equipo_A) -}
    if not is_init(t.equipo_A) ->
      decr(t.equipo_A)
    else
      Skip
    end if
  od
end proc
```

Practico 2.2 Ejercicio 4

(b)

```
{- TRATEMOS DE TERMINAR DE ITERAR ANTES SI N ES MUCHO MAS GRANDE QUE EL CONTADOR -}
proc castigar_A(in/out t: Tablero, in n: nat)
  Var i: nat
  i := 1
  do i <= n && not is_init(t.equipo_A) ->   {- do con posible terminación anticipada -}
    decr(t.equipo_A)
    i := i + 1
  od
end proc
```

ESTA FUNCIÓN ANDA MUCHO MÁS RÁPIDO SI LLAMO A castigar_A(t, 1000000000) cuando el equipo A tiene pocos tantos.

Practico 2.2 Ejercicio 4

```
(c)
Type Tablero = tuple
  equipo_A: nat
  equipo_B: nat
end tuple

proc anotar_varios_A(in/out t: Tablero, in n: nat)
  t.equipo_A := t.equipo_A + n
end proc
```

Practico 2.2 Ejercicio 2: tail

```
type List of T = tuple
                   elems  : array[1..N] of T
                   size  : nat
                  end tuple
# ejemplo: N=5, lista.elems = [10,-2,5,7,-1], lista.size = 3. Cuál es la lista? [10,-2,5]
                     [10,-2,5,8,5], lista.size = 3. Cuál es la lista?
# en el ejemplo: tail(lista) debe dar lista.elems = [-2, 5, 5, 7, -1] lista.size = 2.

{- PRE: not is_empty(lista) -}
proc tail(in/out lista: List of T)
  {- queremos correr elementos a la izquierda: desde el 2do hasta el lista.size-ésimo -}
  for i := 1 to lista.size-1 do
    lista.elem[i] := lista.elem[i+1]
  od
  lista.size := lista.size - 1
end proc

# anda con un elemento? SÍ!
```

Practico 2.2 Ejercicio 2: take

```
type List of T = tuple
                  elems  : array[1..N] of T
                  size  : nat
                 end tuple
# ejemplo: N=5, lista.elems = [10,-2,5,7,-1], lista.size = 3. Cuál es la lista? [10,-2,5]

# PREGUNTA: qué da take(lista, 1) para este ejemplo? lista.elems y lista.size??
# respuesta posible: lista.elems = [10,-2,5,7,-1], lista.size = 1.

# otro ejemplo: take(lista, 10) va a dar lista.elems = [10,-2,5,7,-1], lista.size = 3.

proc take(in/out lista: List of T, in n: nat)
  lista.size := n - 1   <- ESTA BIEN ESTO? ANDA EN TODOS LOS CASOS?
end proc

Qué da take(lista, 1) en este caso? lista.size = 0. ESTÁ BIEN ESTO? NO. DEBE SER lista.size = 1
Qué da take(lista, 2) en este caso? lista.size = 1. ESTÁ BIEN ESTO? NO. DEBE SER lista.size = 2
```

Practico 2.2 Ejercicio 2: take

```
type List of T = tuple
                   elems  : array[1..N] of T
                   size  : nat
                  end tuple
# ejemplo: N=5, lista.elems = [10,-2,5,7,-1], lista.size = 3. Cuál es la lista? [10,-2,5]

# otro ejemplo: take(lista, 10) va a dar lista.elems = [10,-2,5,7,-1], lista.size = 3.

proc take(in/out lista: List of T, in n: nat)
  {- lista.size := n -}
  if n <= lista.size ->
    Lista.size := n
  else ->
    {- la lista queda igual -}
    skip
  fi
end proc
```

Supongamos quiero hacer “unión” de los conjuntos representados por

s = [1,3,4,5]   s0 = [2,3,5,6]

Luego de llamar a addr para cada elemento de s0 quedaría

s = [1,2,3,4,5,6]

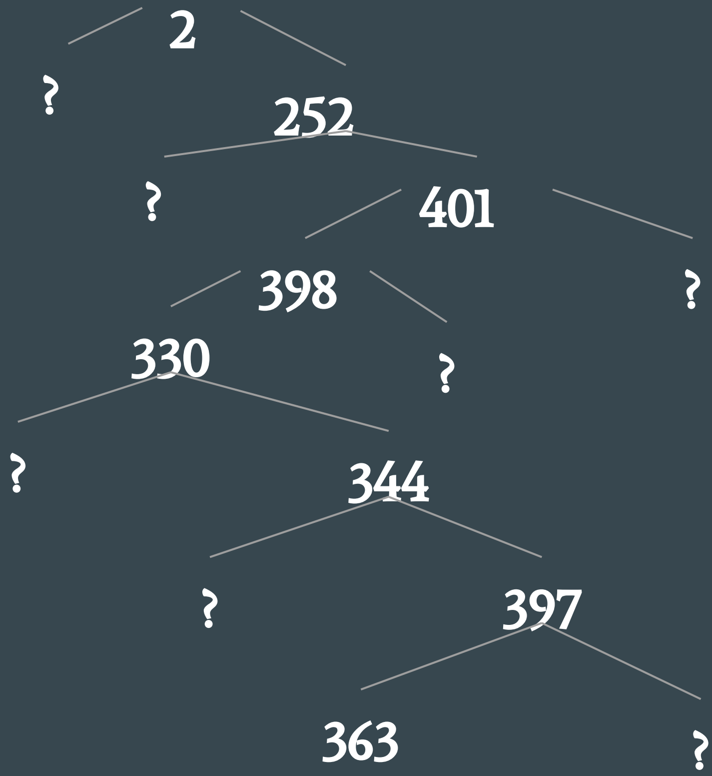
{ PRE: s y s0 están ordenadas y sin repeticiones }

proc union(in / out s : Set of T, in s0 : Set of T)

…..

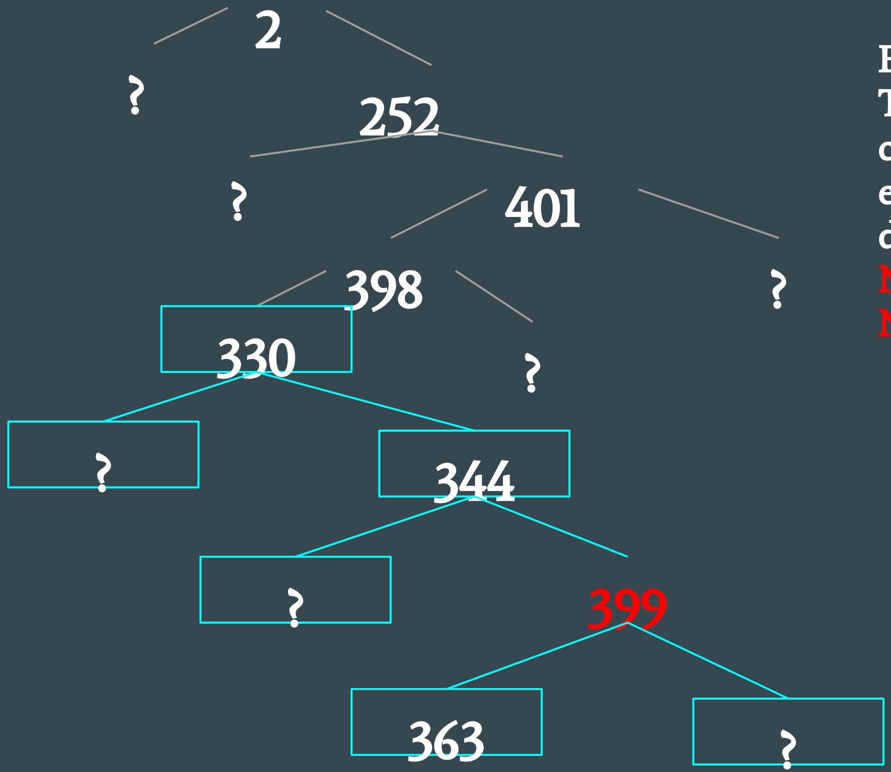
{POST: s está ordenado y sin repeticiones, y todo elemento de s0 está en s}

Practico 2.3: Ejercicio 6a: Buscar el 363: 2, 252, 401, 398, 330, 344, 397, 363.



Este sí es un ABB bien formado.

Practico 2.3: Ejercicio 6a: Buscar el 363: 2, 252, 401, 398, 330, 344,399, 363.



Este es o no es?
Todo lo que está en celeste está como subarbol izquierdo del elemento 398. Todos los elementos deben ser menores que 398.
No puede estar el 399.
No es un ABB bien formado.

Practico 2.3: Ejercicio 3b: Colas más eficientes

Ayuda 1: agregar un campo más.

```
type Queue of T = tuple
  elems: array[0, N-1] of T
  size: nat    {- este campo indica la cantidad de elementos de la cola -}
  start: nat   {- este campo indica a donde empieza la cola en el arreglo -}
end tuple
```

```
Ejemplo: N=7. elems = [-10, -2, 5, -1, 7, 6, 10]. size = 4. start = 0.
Desencolar en el 3a: elems = [-2, 5, -1, -1, 7, 6, 10]. size = 3.
¿Podemos cambiar a donde empieza la cola? Agregar un campo start!
Desencolar en el 3b: elems = [-10, -2, 5, -1, 7, 6, 10]. size = 3. start = 1.
Código de dequeue:
  q.size := q.size - 1
  q.start := q.start + 1
```

Practico 2.3: Ejercicio 3b: Colas más eficientes

Ejemplo: N=7. elems = [-10, -2, 5, -1, 7, 6, 10]. size = 3. start = 1.

Ahora encolamos tres elementos más: el -4, el 0 y el 12:

elems = [-10, -2, 5, -1, -4, 0, 12]. size = 6. start = 1.

Y desencolamos dos veces:

elems = [-10, -2, 5, -1, -4, 0, 12]. size = 4. start = 3.

¿puedo encolar más elementos? Ayuda 2: usar aritmética modular! (???)

Pensar luego del ultimo elemento del arreglo, sigue el primero.

Probemos encolar usando esta idea: Agregamos el 47: Quedaría:

elems = [47, -2, 5, -1, -4, 0, 12]. size = 5. start = 3.

¿Porqué modular? El 4-ésimo (empezando en 0) elemento de la cola está en la posición:

(start + 4) mod N = (3 + 4) mod 7 = 0

Hay que pensar cómo implementar las operaciones enqueue y dequeue para que ande.

Probemos encolar usando esta idea: Agregamos el 47: Quedaría:
elems = [47, -2, 5, -1, -4, 0, 12]. size = 5. start = 3.

EJERCICIO: ENCOLAR OTRO ELEMENTO MÁS: EL 77. ¿CÓMO QUEDA EL ARREGLO, SIZE Y START?

elems = [47, 77, 5, -1, -4, 0, 12]
size = 6
start = 3
¿qué código acabo de ejecutar para el ENQUEUE?

```
q.elems[???] := 77   <- qué va en los ???? EN EL 3A ERA: q.size. AHORA?
q.size := q.size + 1
```

Empezar en start: Avanzar size. Si me paso, cómo hago?

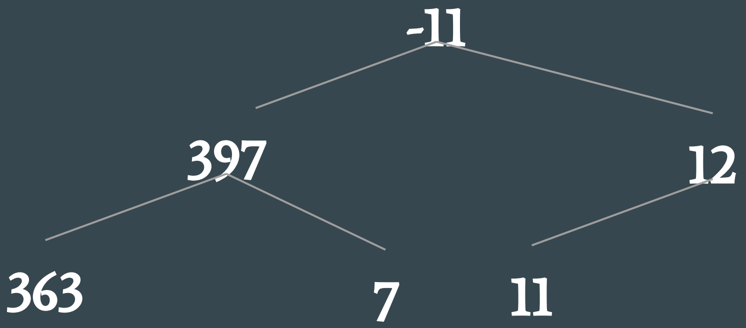
```
[47, -2, 5, -1, -4, 0, 12]
             3^  4  5   6  7  8 me pasé del tamaño del arreglo, cómo sigo?
                        6  0  1 O MEJOR DICHO
0    1^  o sea 8 mod N. DE NUEVO EL CÓDIGO PARA ENCOLAR (ORDEN CONSTANTE):
q.elems[(q.start + q.size) mod N] := e
q.size := q.size + 1
```

EJERCICIO PARA UDS: IMPLEMENTAR DEQUEUE USANDO ESTA MISMA IDEA (ORDEN CTTE).

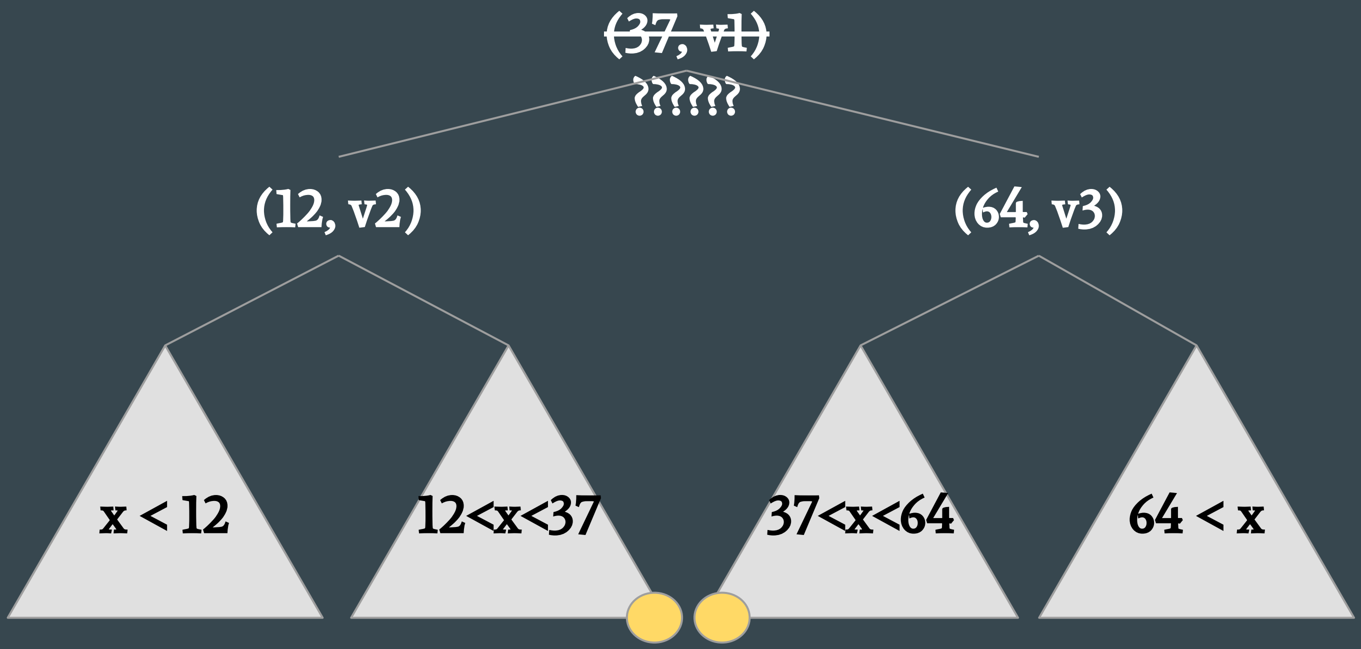
Árboles:

Tenemos dos árboles: a1 y a2:

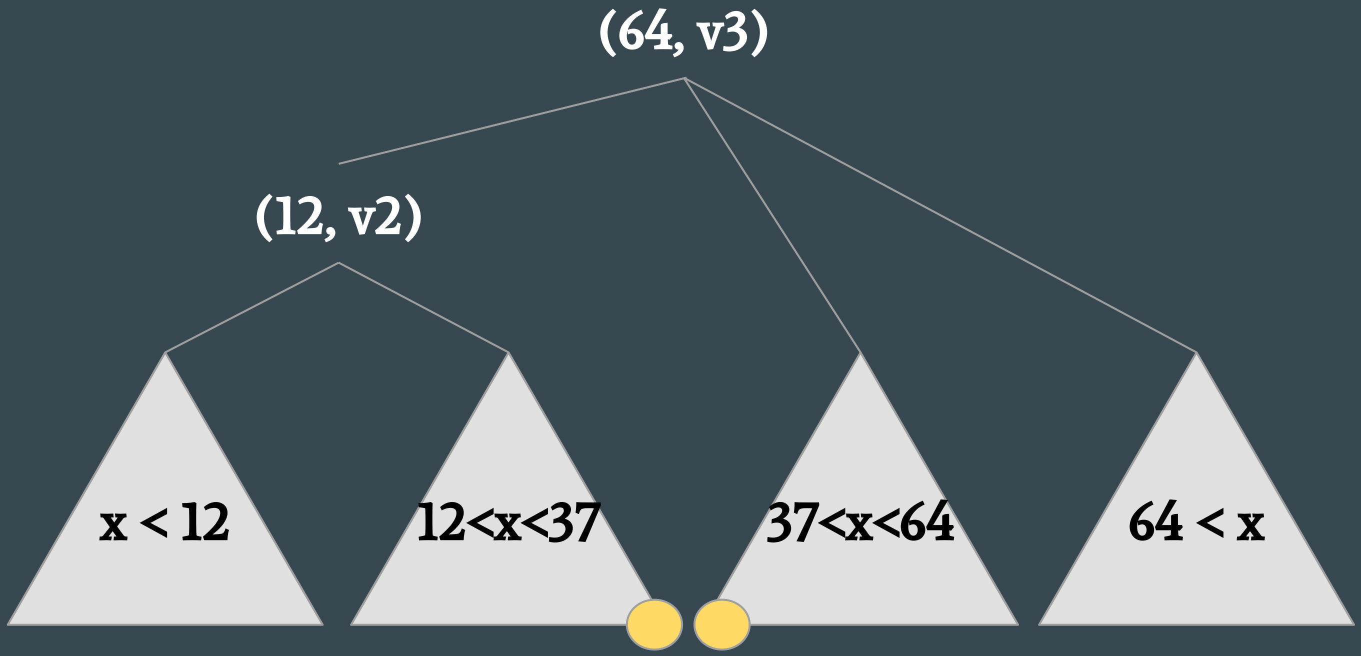
Resultado de la función node(a1, -11, a2): Un nuevo árbol:



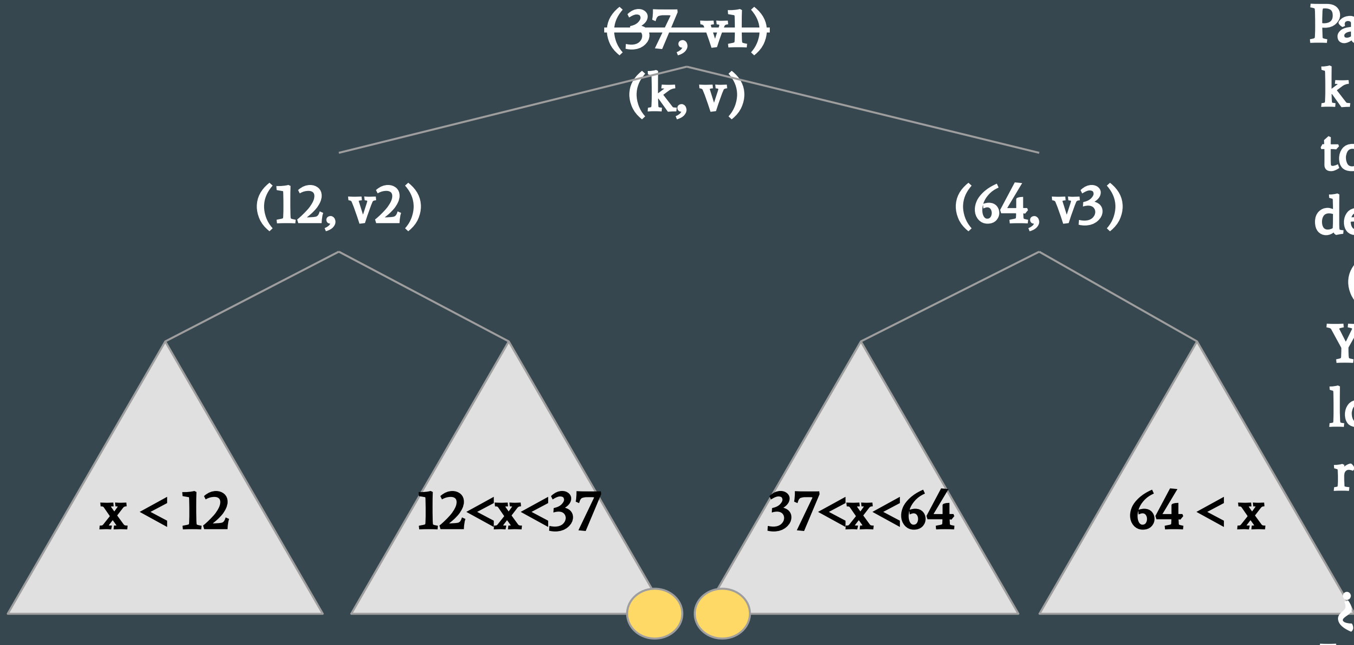
Diccionarios: Eliminar elemento



Diccionarios: Eliminar elemento (MALA IDEA)

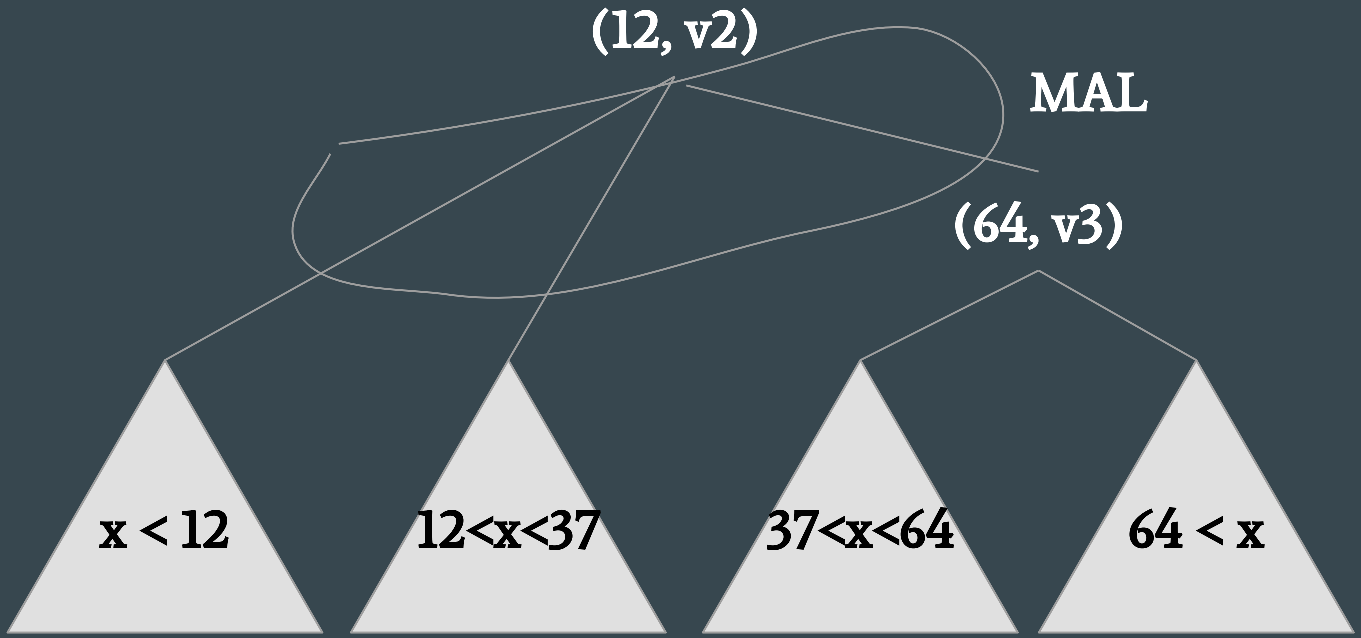


Diccionarios: Eliminar elemento: versión que anda

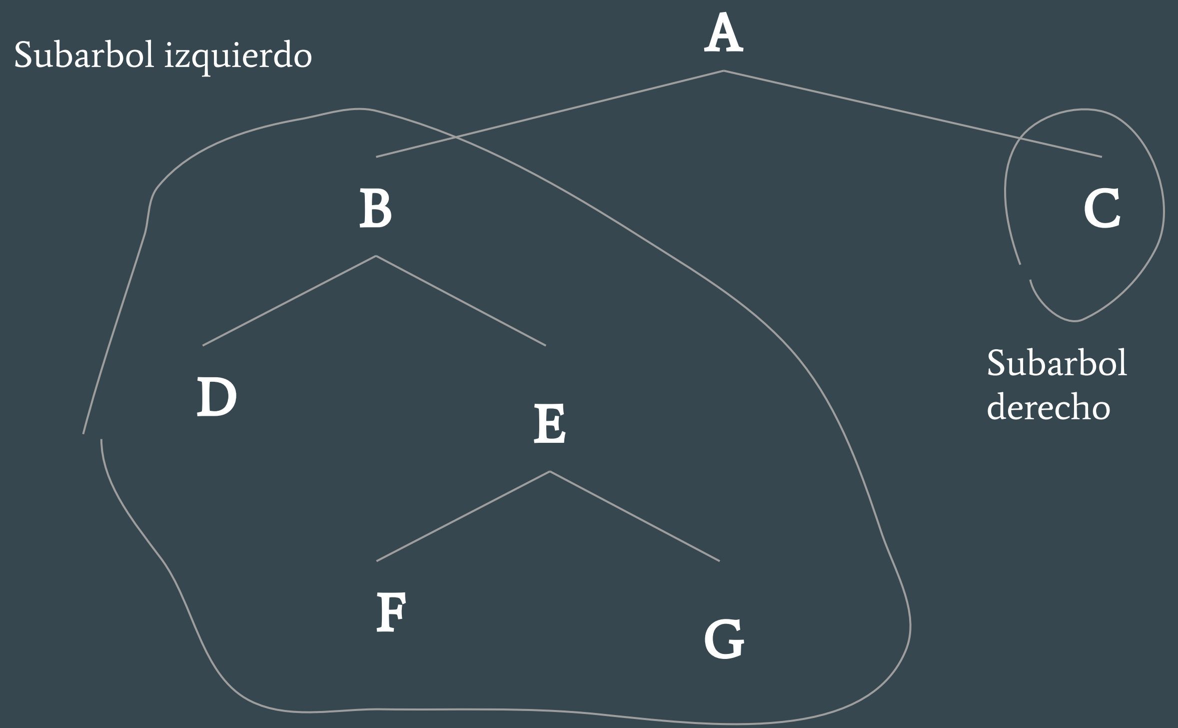


Para que sea un ABB k deber ser mayor a todos los elementos de la rama izquierda (van del 12 al 36). Y menor que todos los elementos de la rama derecha (van del 38 al 64). ¿de dónde lo saco? Los dos circulitos lo cumplen!

Diccionarios: Eliminar elemento: Versión que no anda



Arboles binarios: is_path, subtree_at y height



is_path(t, [L,L,L]) = False
is_path(t, [L,R,L]) = True

subtree_at con el camino
[Left,Right]

height: 4

h_left = 3
h_right = 1

height = (h_left max h_right) + 1

Recorrer Árboles: Idea general con recursividad

```
implement Tree of T where

type Node of T = tuple
  left: pointer to (Node of T)
  value: T
  right: pointer to (Node of T)
end tuple

type Tree of T= pointer to (Node of T)
```

```
{- esta funcion va a recorrer todo
el arbol -}
fun f(t: Tree of T)
  if is_empty_tree(t) ->
    {- caso base: hacer algo -}
  else
    {- el arbol es no vacío -}

    {- Llamar a f sobre el subarbol
izquierdo y/o sobre el derecho. -}
    . . . . f(t->left)  . . . .
    . . . . f(t->right)  . . . .
end fun
```

Recorrer Árboles: Altura de un Árbol

```
implement Tree of T where

type Node of T = tuple
  left: pointer to (Node of T)
  value: T
  right: pointer to (Node of T)
end tuple

type Tree of T= pointer to (Node of T)
```

```
fun height(t: Tree of T) ret h: nat
  var h_left, h_right: nat

  if is_empty_tree(t) ->
    h := 0
  else
    h_left := height(t->left)
    h_right := height(t->right)
    h := (h_left max h_right) + 1
  fi
end fun
```

ACÁ: h_left y h_right son variables auxiliares!!

Recorrer Árboles: is_path

```
implement Tree of T where

type Node of T = tuple
  left: pointer to (Node of T)
  value: T
  right: pointer to (Node of T)
end tuple

type Tree of T= pointer to (Node of T)

type Direction = enumerate
                  Left
                  Right
                 end enumerate

Type Path = List of Direction
```

```
{- Devuelve True si p es un camino válido en t -}
fun is_path(t: Tree of T, p: Path) ret b:Bool
  var d: Direction
  Var aux: Path
  if is_empty_tree(t) ->
    b := False
  else
    if is_empty_list(p) ->
      b := True
    else
      d := head(p)
      aux := list_copy(p)
      tail(aux)
      if d = Left ->
        b := is_path(t->left, aux)
      else ->
        b := is_path(t->right, aux)
      list_destroy(aux)
    end if
   end if
end fun
```

Ejercicio 4, Práctico 3.1: Tipos de datos

```
fun salvar_ballenas(bs : Set of Ballena,t : Nat) ret ls : List of Ballena
{- IDEA: Llevo un conjunto con aquellas ballenas que están aún varadas y con vida. -}
  var vivas : Set of Ballena
      b : Ballena
  vivas := set_copy(bs)
  ls := empty_list()
  do (not is_empty_set(vivas)) →
    {- Invariante: toda ballena b del conjunto vivas satisface b.time > 0
       y b no está en la lista ls -}
     b := seleccionar_ballena(vivas)
     addr(ls,b)
     elim(vivas,b)
     descontar_tiempo(vivas,t)
     quitar_muertas(vivas)
  od
  set_destroy(vivas)
end fun
```

Recorrer Árboles: subtree_at

```
implement Tree of T where

type Node of T = tuple
  left: pointer to (Node of T)
  value: T
  right: pointer to (Node of T)
end tuple

type Tree of T= pointer to (Node of T)

type Direction = enumerate
                  Left
                  Right
                 end enumerate

Type Path = List of Direction
```

```
{- PRE: True -}
fun subtree_at(t: Tree of T, p: Path) ret st : Tree of T
  var dir : Direction
  var p_tail : Path

  if is_empty_tree(t) ->
    st := t
  else
    {- el árbol es no vacío, qué hago? -}
    if is_empty_list(p) ->
      st := t
    else
      dir := head(p)
      p_tail := list_copy(p)
      tail(p_tail)
      if dir == Left ->
        st := subtree_at(t->left, p_tail)
      else dir == Right ->
        st := subtree_at(t->right, p_tail)
      fi
      list_destroy(p_tail)
    fi
end fun
```

Ejercicio 4, Práctico 3.1: salvemos las ballenas

- Cada ballena “i” tiene un tiempo que le queda de vida s_i minutos.
- Salvar a una ballena requiere un tiempo t minutos.
- La ballena no muere mientras está siendo llevada al mar.

Criterio de selección: Elijo en cada momento a la ballena que le queda menos tiempo de vida.

Ejemplo: Tengo 3 ballenas. Con tiempos respectivos de vida 30, 9, 15. El tiempo t de rescate es 5.

Con el criterio opuesto al que propusimos, elijo a la primer ballena. La salvo, entonces a las otras dos les queda tiempo de vida 4 y 10 respectivamente.

En el segundo paso, elijo a la tercer ballena. Y me queda una varada que se muere mientras llevé a la tercera.
CONCLUSIÓN: pude salvar 2 ballenas.

Con el criterio propuesto: ¿qué ballena salvo primero?
Salvo a la segunda. El resto entonces queda con tiempo de vida: 25 y 10.
Luego salvo a la tercer ballena, y me queda varada solamente la primera, a la que le quedan 20 minutos de vida.
Luego la salvo y pude salvarlas a las 3.

Ejercicio 4, Práctico 3.1: Tipos de datos

Defino el tipo Ballena:

```
type Ballena = tuple
                   id : Nat
                   time : Nat
                 end tuple

fun salvar_ballenas(bs : Set of Ballena,t : Nat) ret ls : List of Ballena
```

Ejercicio 4, Práctico 3.1: Algoritmo

```
proc descontar_tiempo(in/out bs : Set of Ballena,in t : Nat)
  var bs_aux : Set of Ballena
      b : Ballena
  bs_aux := set_copy(bs)
  do (not is_empty_set(bs_aux))
    b := get(bs_aux)
    elim(bs,b)
    b.time := 0 `max` (b.time - t)
    add(bs,b)
    elim(bs_aux,b)
  od
  set_destroy(bs_aux)
end fun
```

RESTA implementar seleccionar_ballena y quitar_muertas

Ejercicio 4, Práctico 3.1: VERSIÓN 2: Con arreglos

Defino el tipo Ballena:

```
type Ballena = tuple
                   id : Nat
                   time : Nat
                 end tuple

fun salvar_ballenas(bs : array[1..N]  of Ballena,t : Nat)
                                                      ret ls : List of Ballena
```

IDEA:

Ordeno el arreglo bs de acuerdo al criterio de selección. En este caso: ordeno
crecientemente de acuerdo al tiempo de vida.
Llevo la cuenta del tiempo desde que comencé a salvar ballenas.
Recorro el arreglo, actualizando en cada momento el tiempo, y cuando avanzo,
si el tiempo de vida de la ballena siguiente es mayor al tiempo acumulado,
la puedo salvar, caso contrario, la salteo y por tanto queda descartada.

Ejercicio 4, Práctico 3.1: VERSIÓN 2: Con arreglos

```
fun salvar_ballenas(bs : array[1..N] of Ballena, t : Nat)
                                                       ret ls : List of Ballena
  var t_acum : Nat

  sort_ballenas(bs)   {- ACA USO EL CRITERIO DE SELECCIÓN -}
  t_acum := 0
  ls := empty_list()
  for i := 1 to N do
    if bs[i].time > t_acum  {- si la ballena i-ésima está viva -}
      then addr(ls,b[i])    {- la salvo -}
           t_acum := t_acum + t
      else skip
    fi
  od
end fun
```

Ejercicio 4, Práctico 3.1: VERSIÓN 2: Con arreglos

Para implementar sort_ballenas:

Uso cualquier algoritmo de ordenación visto, por ejemplo selection_sort,
cambiando la parte donde diga

```
                        a[i] < a[i+1]   por ejemplo, por
                        a[i].time < a[i+1].time
```

Ejercicio 3 Práctico 3.1: viaje en auto

- Hay n+1 ciudades por las que debo pasar.
- Se conoce la distancia d_i (en km), entre la localidad c_(i-1) y c_i.
- El auto tiene autonomía de A km.
- Inicialmente el tanque está vacío

EJEMPLO: 4 localidades. A = 300 km.
d_1 = 200
d_2 = 90
d_3 = 200

¿cuál es la menor cantidad de cargas necesarias? Y ¿en qué ciudades?

En c0 cargo seguro. Llego a c1 y me quedan 100 km de autonomía.
¿cargo en c1?
No cargo, porque me alcanza hasta c2.
Llego a c2 y me quedan 10 km de autonomía.
¿cargo en c2?
Sí, porque de lo contario no llego hasta c3.
Luego llego a c3 y me quedan 100 km de autonomía.

Solución: Cargo en c0 y c2.

¿Cuál es el criterio para decidir si cargo combustible?

Ejercicio 9 Práctico 3.1: sobredosis de limón

- Hay n bares
- Cada bar i tiene un precio regular P_i de la pinta, y un número H_i de cantidad de horas Happy desde las 18.
- En Happy Hour la pinta vale el 50%.
- Se toman 2 pintas por hora desde las 18 hasta las 2 am.
- No se cuenta el tiempo para moverse entre bares.
- Objetivo: Tomar 16 pintas al menor precio posible.

EJEMPLO: Tres bares

n = 3

P_1 = 200 P_2 = 180 P_3 = 250

H_1 = 2 H_2 = 1 H_3 = 3

18 a 19: Bar 2 en HH (llevo gastado 180 en 2 pintas)

19 a 20: Bar 1 en HH (llevo gastado 380 en 4 pintas)

20 a 21: Bar 3 en HH (llevo gastado 630 en 6 pintas)

21 a 2: Bar 2 sin HH (llevo gastado 2430 en 16 pintas)

TOTAL gastado: 2430

ALGORITMOS VORACES. Dos enfoques para implementar

- **Con conjuntos**
  En cada paso elijo un elemento de acuerdo al criterio. Lo quito del conjunto, y también descarto los que ya no pueden ser solución.
- **Con arreglos**
  Ordeno el arreglo de acuerdo al criterio. Luego recorro el arreglo y para cada elemento decido si es parte de la solución o no.

Ejercicio 2, pr 3.1

Considere el problema de dar cambio. Pruebe o dé un contraejemplo: si el valor de cada moneda es al menos el doble de la anterior, y la moneda de menor valor es 1, entonces el algoritmo voraz arroja siempre una solución óptima.

Ejemplo 1 (Lucas) Denominaciones:
1, 3, 7, 15, 32
Tengo que pagar 37. El algoritmo me devuelve:

32, 3, 1, 1 ----> TOTAL 4 monedas
Pero podía pagar con
15, 15, 7

Recordemos: selecciono moneda de la mayor denominación que no se pase de lo que tengo que pagar.

HIPÓTESIS: Es falso.
Si es así, debo encontrar contraejemplo.

Ejemplo 2. Denominaciones
1, 15, 46
Tengo que pagar 60
El algoritmo da que necesito 15 monedas. Pero se puede hacer con 4 de 15.

Ejemplo 3: denominaciones 1, 4, y 9. Pagar 12.
El algoritmo da 9,1,1,1 pero se puede con 4,4,4.

Ejercicio 5, pr 3.1. Prestar el teléfono satelital

- Tengo n amigos.
- Cada amigo i tiene un día p_i de partida y r_i de regreso (dos naturales).
- No puedo prestar el teléfono a dos amigos simultáneamente.
- Cuando el teléfono regresa desde un amigo, puedo volver a prestarlo al día siguiente.
- Objetivo: prestar el teléfono al mayor número posible de amigos.

Amigo 1 p= 3 r=7
Amigo 2 p=6 r=10
Amigo 3 p=12 r=14
Amigo 4 p=8 r=13
Amigo 5 p=14 r=27

Le puedo prestar el teléfono a 3 amigos:
Amigo 1, 4 y 5.

Criterio de selección:
1. Menor fecha de regreso.
2. Menor duración del viaje, es decir, menor r-p.
3. Menor fecha de partida.

Analicemos criterio 2:
Primero elijo el amigo 3.
Luego elijo al 1. Y no puedo elegir a más nadie.

El criterio 2 NO es óptimo.
El 3 tampoco.

El criterio óptimo es el primero.

Ejercicio 5, pr 3.1. Tipos de datos

Defino el tipo Amigo:

```
type Amigo = tuple
               nombre : String
               partida : Nat
               regreso : Nat
              end tuple

fun prestar_tel(as : array[1..N] of Amigo) ret ls : List of Amigo
```

Idea:

* Ordenar el arreglo de acuerdo al día de regreso de cada amigo.
* Recorro el arreglo de izquierda a derecha.
* Guardo en una variable t el día de regreso del último amigo al que se lo presté.
* En cada paso del ciclo: Chequeo si el amigo de la posición “i” tiene fecha de partida > t. En ese caso lo presto. Y actualizo t.

Ejercicio 5, pr 3.1. Tipos de datos

```
fun prestar_tel(as : array[1..N] of Amigo) ret ls : List of Amigo
  var t : Nat
  var as_aux : array[1..N] of Amigo

  t := 0   {- t es el día de regreso del último amigo al q le presté -}
  ls := empty_list()
  copy_array(as,as_aux)

  sort_amigos(as_aux)   {- se ordena de acuerdo de la fecha de regreso -}
  for i:= 1 to N do
    if as_aux[i].partida > t then
      addr(ls,as_aux[i])
      t := as_aux[i].regreso
    fi
  od
end fun
```

Ejercicio 5, pr 3.1. Versión 2. Tipos de datos

Defino el tipo Amigo:

```
type Amigo = tuple
               nombre : String
               partida : Nat
               regreso : Nat
             end tuple

fun prestar_tel(as : Set of Amigo) ret ls : List of Amigo
```

Idea:

* Guardo en una variable t el día de regreso del último amigo al que se lo presté.
* Un ciclo mientras el conjunto de candidatos no sea vacío: elijo al amigo con fecha de regreso más chica. Luego lo elimino del conjunto. Lo agrego a la solución. Luego elimino del conjunto todos los que no son factibles.

Ejercicio 5, pr 3.1. Tipos de datos

```
fun prestar_tel(as : Set of Amigo) ret ls : List of Amigo
  var t : Nat
  var as_aux : array[1..N] of Amigo
  var a : Amigo

  t := 0
  ls := empty_list()
  as_aux := set_copy(as)

  do (not is_empty_set(as_aux))
    {- INVARIANTE: Todos los elementos de as_aux tienen partida mayor a t -}
    a := select_amigo(as_aux)
    elim(as_aux,a)
    addr(ls,a)
    t := a.regreso
    elim_no_factibles(as_aux,t)
  od
  set_destroy(as_aux)
end fun
```

Ejercicio 5, pr 3.1. Tipos de datos

```
proc elim_no_factibles( in/out as : Set of Amigo, in t : Nat)
  var as_aux : Set of Amigo
  var a : Amigo

  as_aux := set_copy(as)
  do (not is_empty_set(as_aux))
    a := get(as_aux)
    if (a.partida <= t)   {- chequeo si a NO es factible -}
      then elim(as,a)
    fi
    elim(as_aux,a)
  od
  set_destroy(as_aux)
end proc
```

Ejercicio 5, pr 3.1. Tipos de datos

```
{- PRE: not is_empty_set(as) -}
fun select_amigo(as : Set  of Amigo) ret a : Amigo
  var as_aux : Set  of Amigo
  var m : Nat
  var b : Amigo

  m := +inf
  as_aux := set_copy(as)

  do (not is_empty_set(as_aux))
    b := get(as_aux)
    if b.regreso < m
      then m := b.regreso
           a := b
    fi
    elim(as_aux,b)
  od
  set_destroy(as_aux)
end fun
```

Practico 3.1: Ejercicio 7

- n sobrevivientes.
- Cada sobreviviente i consume c_i de oxígeno por minuto.
- Rescatar demora t minutos.
- Disponemos de un total de C oxígeno.
- Queremos salvar la mayor cantidad de sobrevivientes.
- Rescatamos de a 2 por vez.

Ejemplo.
5 sobrevivientes. C es 85. t es 3.

$c_1 = 3$ $\quad$ $c_4 = 4$
$c_2 = 5$ $\quad$ $c_5 = 2$
$c_3 = 2$

¿a quiénes rescato?
Idea:
* primero rescato a 3.
Cuánto me queda de oxígeno al regresar de ese rescate?
c_1 consumió 9, c_2 15, c_4 12 y c_5 6.
TOTAL consumido = 42
Entonces luego de rescatar a 3, me quedan 43 de oxígeno.

El criterio de salvar al que MENOS consume no parece ser adecuado.

CRITERIO propuesto: Salvar a quienes consumen MÁS.

Ejercicio 7(b), pr 3.1. Tipos de datos

Defino el tipo Submarinerx:

```
type Submarinerx = tuple
                    nombre  : String
                    oxigeno : Float
                  end tuple

fun submarino(as : array[1..N] of Submarinerx,C : Float, t : Nat, m : Nat) ret ls : List of
Submarinerx
```

Idea:

* Ordeno el arreglo as decrecientemente de acuerdo al valor de “oxigeno” de cada submarinerx.
* En una variable llevaré la cuenta de cuánto oxígeno queda.
* Luego recorro el arreglo de izquierda a derecha mientras me quede oxígeno. En cada paso avanzo de a “m” lugares, salvando a esos “m” sumarinerxs. Luego descuento del restante de oxígeno lo que consumieron todos los demás sumarinerxs que están en las siguientes posiciones del arreglo.

Ejercicio 7(b), pr 3.1. Tipos de datos

```
fun submarino(as : array[1..N] of Submarinerx, C : Float, t : Nat, m : Nat) ret ls : List of
Submarinerx
  var ox: Float
  var bs: array[1..N] of Submarinerx
  var i: Nat

  copy_array(bs, as)
  sort_submarinerxs(bs)  {- procedimiento auxiliar -}
  ox := C
  ls := empty_list()
  i := 1
  do ox > 0 and i <= N ->
    for j := i to min(i+m-1, N) do   {- uso min: me cuido de no pasarme del arreglo -}
      addr(ls, bs[j])
    end
    i := min(i + m, N + 1)
    ox := ox - consumo_oxigeno(bs, t, i)  {- función auxiliar -}
  od
end fun
```

Ejercicio 2, pr 3.2

$w((1,2)) = 7$	$w((2,3)) = 4$	$w((3,6)) = 4$	$w((5,6)) = 6$
$w((1,6)) = 3$	$w((2,4)) = 2$	$w((3,8)) = 6$	$w((6,7)) = 5$
$w((1,7)) = 5$	$w((2,5)) = 1$	$w((4,6)) = 8$	$w((8,5)) = 2$
$w((1,3)) = 3$	$w((3,4)) = 5$	$w((5,4)) = 3$	$w((8,7)) = 3$

vértice inicial = 1

		1 2 3 4 5 6 7 8
Paso 1.	C = {1}	$D = [\underline{0},7,3,\infty,\infty,3,5,\infty]$
Paso 2.	C = {1,3}	$D = [\underline{0},7,\underline{3},8,\infty,3,5,9]$
Paso 3.	C = {1,3,6}	$D = [\underline{0},7,\underline{3},8,\infty,\underline{3},5,9]$
Paso 4.	C = {1,3,6,7}	$D = [\underline{0},7,\underline{3},8,\infty,\underline{3},\underline{5},9]$
Paso 5.	C = {1,3,6,7,2}	$D = [\underline{0},\underline{7},\underline{3},8,8,\underline{3},\underline{5},9]$
Paso 6.	C = {1,3,6,7,2,4}	$D = [\underline{0},\underline{7},\underline{3},\underline{8},8,\underline{3},\underline{5},9]$
Paso 7.	C = {1,3,6,7,2,4,5}	$D = [\underline{0},\underline{7},\underline{3},\underline{8},\underline{8},\underline{3},\underline{5},9]$
Paso 8.	C = {1,3,6,7,2,4,5,8}	$D = [\underline{0},\underline{7},\underline{3},\underline{8},\underline{8},\underline{3},\underline{5},\underline{9}]$

Termino porque no hay nodos que no estén en C.

Practico 3.2: Ejercicio 3

- K ciudades: 1, 2, ... K.
- L litros de nafta
- E[i, j]: costo en litros de nafta de i a j.
- Conjunto de ciudades C
- Ciudad de partida: v.
- Calcular **el conjunto D** de ciudades de C que se podrían visitar con L litros.

IDEA:
- Hacer dijkstra partiendo de v.
- Quedarnos con las ciudades con costo <= L.
- Pero, sólo las que están en C.

A = [n1,n2,....,nK] resultado de correr Dijkstra desde v.
* Recorro cada elemento de A. Pregunto si el valor de A[i] es menor o igual a L y si “i” pertenece a C..
En caso que sí agrego “i” a D.
Caso contrario no hago nada y avanzao al siguiente índice.

Practico 3.2: Ejercicio 3. Código

```
fun vacaciones(E: array[1..K,1..K] of nat, v: nat, C: Set of nat, L: nat)
    ret D: Set of nat
  var A: array[1..K] of nat

  A := dijkstra(E, v)
  D := empty_set()
  for i := 1 to K ->
    if A[i] <= L and member(i, C) ->
      add(D, i)
    fi
  end for
end fun
```

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:
$m_1 = 50$, $h_1 = 300$
$m_2 = 70$, $h_2 = 450$
$m_3 = 30$, $h_3 = 300$
$m_4 = 40$, $h_4 = 350$

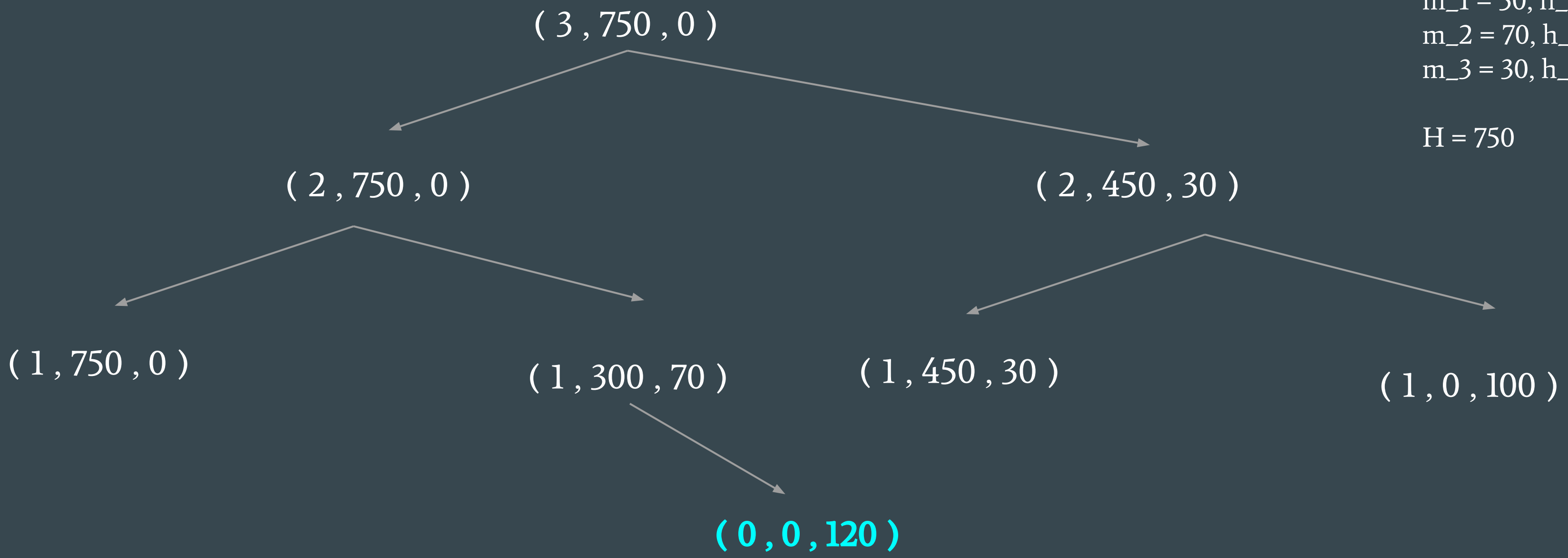
$H = 800$

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 $h_i \leq j$ ------> 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.
- 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

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

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

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

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

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

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																

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

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

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

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

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																	

max(3, 2 + 0)

max(3, 2 + 3)

Ahora la fila 2.

Programación Dinámica: Problema de la Mochila

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

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

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

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
(no necesariamente es la solución)

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.

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)

¿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.

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

2	3	4	2
1	3	4	1
3	2	6	
3	2	3	4

¿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 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))