Introducción al lenguaje de la materia

a. Intercambiar los valores de dos variables.

```
proc swapVar (in/out a, b : T)
  var aux : T
  aux := a
  a := b
  b := aux
end proc
```

b. Ordenar dos números de mayor a menor.

```
proc ordNum (in/out n, m : Int)
  if n > m then
    swapVar(n, m)
  else
    skip
  fi
end proc
```

c. Valor absoluto de un número entero.

```
fun abs (number : Int) ret v_abs : Int
  if number < 0 then
    v_abs := number * (-1)
  else
    v_abs := number
  fi
end fun
```

d. Escribir una función que devuelva el máximo valor entre dos números.

```
fun max_var (n, m : Int) ret v_max : Int
  if n > m then
    v_max := n
  else
    v_max := m
  fi
end fun
```

e. Escribir una función que dado tres números devuelve el número del medio, es decir, ni el máximo ni el mínimo.

```
fun med_var (n, m, s : Int) ret v_med : Int
```

```
  if n > m and m > s then
    v_med := m
  else
    if  m > n and n > s then
    v_med := n
    else
      v_med := s
    fi
  fi
end fun
```

Ordenación elemental (parte 1)

Práctico 1 - Parte 1

1. Escribí algoritmos para resolver cada uno de los siguientes problemas sobre un arreglo a de posiciones 1 a n, utilizando do. Elegí en cada caso entre estos dos encabezados el que sea más adecuado:

a. Inicializar cada componente del arreglo con el valor 0.

```
proc all_0s (out a : array [1..N] of nat)
  var i : Nat
  i := 1
  while i < N do
    a[i] := 0
    i := i + 1
  od
end proc
```

b. Inicializar el arreglo con los primeros n números naturales positivos.

```
proc a_naturals (out a : array [1..N] of Nat)
  var i : nat
  i := 1
  while i < N do
    a[i] := i
    i := i + 1
  od
end proc
```

c. Inicializar el arreglo con los primeros n números naturales impares.

```
proc a_odd (out a : array [1..N] of Nat)
  var i : Nat
  i := 1
  while i < N do
    a[i] := 2 * i - 1
    i := i + 1
  od
end proc
```

d. Incrementar las posiciones impares del arreglo y dejar intactas las posiciones pares.

```
proc a_inc_odd (in/out a : array [1..N] of Nat)
  var i : Nat
  i := 1
  while i < N do
```

```
    if (a[i] mod 2 == 0) then
      skip
    else
      a[i] := a[i] + 1
    fi
    i := i + 1
  od
end proc
```

2. Transforma cada uno de los algoritmos anteriores en uno equivalente que utilice for . . . to .

a. Inicializar cada componente del arreglo con el valor 0.

```
proc all_0s (out a : array [1..N] of nat)
  for i := 1 to N do
    a[i] := 0
  od
end proc
```

b. Inicializar el arreglo con los primeros n números naturales positivos.

```
proc a_naturals (out a : array [1..N] of Nat)
  for i = 1 to N do
      a[i] := i
  od
end proc
```

c. Inicializar el arreglo con los primeros n números naturales impares.

```
proc a_odd (out a : array [1..N] of Nat)
  for i := 1 to N do
      a[i] := 2 * i - 1
  od
end proc
```

d. Incrementar las posiciones impares del arreglo y dejar intactas las posiciones pares.

```
proc a_inc_odd (in/out a : array [1..N] of Nat)
  for i := 1 to N do
    if (a[i] mod 2 == 0) then
      skip
    else
      a[i] := a[i] + 1
    fi
  od
end proc
```

3. Escribí un algoritmo que reciba un arreglo a de posiciones 1 a n y determine si el arreglo recibido está ordenado o no. Explica en palabras que hace el algoritmo y como lo hace.

```
fun is_ordered (a : array [1..N] of Int) ret ordered : Bool
  var i : Nat
  i := 1
  ordered := true
  while (i < (N - 1) and ordered) do
    if a[i] > a[i + 1] then
      ordered := false
    else
      skip
    fi
  od
end fun
```

- ¿Qué hace el algoritmo?

Devuelve true si el arreglo está ordenado, y en caso contrario devuelve false.

- ¿Cómo lo hace?

El algoritmo recorre los índices del arreglo y determina si el elemento en cuestión es mayor al que le sigue y actualiza la variable booleana con false, caso contrario aumenta el índice y vuelve a comparar.

4. Ordena los siguientes arreglos, utilizando el algoritmo de ordenación por selección visto en clase. Mostrar en cada paso de iteración cual es el elemento seleccionado y como queda el arreglo después de cada intercambio.

a. [7, 1, 10, 3, 4, 9, 5]

- Primera iteración i = 1, minp = 2

[1, 7, 10, 3, 4, 9, 5]

- Segunda iteración i = 2, minp = 4

[1, 3, 10, 7, 4, 9, 5]

- Tercera iteración i = 3, minp = 5

[1, 3, 4, 7, 10, 9, 5]

- Cuarta iteración i = 4, minp = 7

[1, 3, 4, 5, 10, 9, 7]

- Quinta iteración i = 5, minp = 7

[1, 3, 4, 5, 7, 9, 10]

- Sexta iteración i = 6, minp = 6

[1, 3, 4, 5, 7, 9, 10]

- Sexta iteración i = 7, minp = 7

[1, 3, 4, 5, 7, 9, 10]

5. Calcula de la manera más exacta y simple posible el número de asignaciones a la variable t de los siguientes algoritmos. Las ecuaciones que se encuentran al final del práctico pueden ayudarte.

a.

```
t := 0
for i := 1 to n do
for j := 1 to n^2 do
for k := 1 to n^3 do
t := t + 1
od
od
od
```

ops (t := 0 for i := 1 to n do for j := 1 to n^2 do for k := 1 to n^3 do t := t + 1 od od od)
t := 0 + ops (for i := 1 to n do for j := 1 to n^2 do for k := 1 to n^3 do t := t + 1 od od od)
1 + ops (for i := 1 to n do for j := 1 to n^2 do for k := 1 to n^3 do t := t + 1 od od od)
1 + ∑(1,n) ops (do for j := 1 to n^2 do for k := 1 to n^3 do t := t + 1 od od)))
1 + (∑ (1,n) ∑(1,n^2) ops (for k := 1 to n^3 do t := t + 1 od))
1 + (∑ (1,n) ∑(1,n^2) ∑(1,n^3) ops (t := t + 1))
1 + (∑(1,n) ∑(1,n^2) ∑(1,n^3) 1
1 + (∑(1,n) + ∑(1,n^2) n^3
1 + ∑(1,n) (n^2 * n^3)
1 + n * n^2 * n^3
1 + n^6

b.

```
t := 0
for i := 1 to n do
for j := 1 to i do
for k := j to j + 3 do
t := t + 1
od
od
od
```

ops (t := 0 for i := n to n do for j := 1 to i do for k := j to j + 3 t := t + 1 od od od)
t := 0 + ops (for i := n to n do for j := 1 to i do for k := j to j + 3 t := t + 1 od od od)
1 + ∑(1,n) ops (for j := 1 to i do for k := j to j + 3 t := t + 1 od od)
1 + ∑(1,n) ∑(1,i) ops (for k := j to j + 3 t := t + 1 od)
1 + ∑(1,n) ∑(1,i) ∑(j,j + 3) ops (t := t + 1)
1 + ∑(1,n) ∑(1,i) ∑(j,j + 3) 1
1 + ∑(1,n) ∑(1,i) 4
1 + ∑(1,n) 4 * i
1 + 4 * n * (n + 1) / 2
1 + 2 * n * (n + 1)

6. Descifra que hacen los siguientes algoritmos, explicar como lo hacen y reescribirlos asignando nombres adecuados a todos los identificadores.

a.

```
proc p (in/out a: array[1..n] of T)
var x: nat
```

```
for i:= n downto 2 do
    x:= f(a,i)
    swap(a,i,x)
od
end proc
```

b.

```
fun f (a: array[1..n] of T, i: nat) ret x: nat
x:= 1
for j:= 2 to i do
   if a[j] > a[x] then
   x:= j
   fi
od
end fun
```

a. El procedimiento "p" ordena el arreglo. Itera desde el último elemento del arreglo hasta el primero, colocando los mayores al último.

```
proc reverse_Sorter (in/out a: array[1..n] of T)
var majorElem_position: nat
for i:= n downto 2 do
    majorElem_position:= get_major_pos(a,i)
    swap(a,i,x)
od
end proc
```

b. La función "f" obtiene la posición del mayor entre dos elementos de un arreglo. Lo hace mediante la ejecución de un ciclo y comparando los elementos a partir de la segunda posición con la del primero, y actualizando la variable para guardarlo.

```
fun get_major_pos (a: array[1..n] of T, i: nat) ret x: nat
major pos:= 1
for j:= 2 to i do
    if a[j] > a[x] then
      major_pos:= j
    fi
od
end fun
```

7. Ordena los arreglos del ejercicio 4 utilizando el algoritmo de ordenación por inserción. Mostrar en cada paso de iteración las comparaciones e intercambios realizados hasta ubicar el elemento en su posición.

a. [5, 4, 3, 2, 1]

[5, 4, 3, 2, 1] → invariante (parte ordenada [ ]), menor elemento = a[5]
[1, 4, 3, 2, 5] → invariante (parte ordenada [1]), menor elemento = a[4]
[1, 2, 3, 4, 5] → invariante (parte ordenada [1,2]), menor elemento = a[4]
[1, 2, 3, 4, 5] → invariante (parte ordenada [1,2,3]), menor elemento = a[3]
[1, 2, 3, 4, 5] → invariante (parte ordenada [1,2,3,4]), menor elemento = a[4]

[1, 2, 3, 4, 5] → invariante (parte ordenada [1,2,3,4,5]), menor elemento = a[5]

8. Calcula el orden del número de asignaciones a la variable t de los siguientes algoritmos.
   a. t := 1
      do t < n
         t := t ∗ 2
      od

   b. t := n
      do t > 0
         t := t div 2
      od

   c. for i := 1 to n do
         t := i
         do t > 0
            t := t div 2
         od
      od

   d. **for** i := 1 to n do
         t:= i
         do t > 0
         t := t − 2
         od
      od

   a. Vemos que crece muy lentamente, es de orden logarítmico.

n = 1 → 1
n = 2 →2
n = 3 → 3
n = 4 → 3
n = 5 → 4
n = 10 → 4

   b. Análogo al anterior, es logarítmico.

9. Calcula el orden del número de comparaciones del algoritmo del ejercicio 3.

- En el peor caso, hace "n" comparaciones, y en el mejor caso hace una sola comparación.

Practico 1 - Parte 3

1. Calcula el orden de complejidad de los siguientes algoritmos:

a. **proc** f1(**in** n : nat)
if n ≤ 1 **then** skip
else
for i := 1 to 8 **do** f1(n div 2) **od**
for i := 1 to n^3 **do** t := 1 **od**
fi
end proc

Comencemos viendo que efectivamente es un algoritmo de la forma divide y vencerás :

- El tamaño del problema es el input “n”
- El algoritmo realiza un “skip” en el caso simple.
- El algoritmo se llama recursivamente a sí mismo con la mitad del tamaño.
- En el caso complejo además de la recursión realiza una asignación a t.
- Contaremos el número de asignaciones a la variable t.

Veamos que las recurrencias denotadas por **“a”** son 8, ya que la llamada recursiva está en un bucle el cual se itera 8 veces.
Por otro lado, la recursión se realiza con la mitad del tamaño original del problema, por ende **“b”** es 2.
Finalmente, **“k”** que es el orden de la operación que realiza además de la recursión en el caso complejo es 3 ya que el bucle se itera hasta n^3.

Una vez identificado esto, planteemos la función t(n) que calculara el costo computacional del algoritmo en cuestión.

- ☐ **t(n)** = **0** si n = 0,1
  = **8 * t(n/2) + n^3** si n > 1

Finalmente veamos el orden de t(n). Como a es 8 y b^k es también 8, tenemos que **a = b^k**. Por el teórico sabemos que al cumplirse esto, tenemos que el orden de t(n) es n^k * log(n).

- ☐ La respuesta entonces es que el algoritmo tiene un orden de **n^3 * log(n).**

b. **proc** f2(in n : nat)
for i := 1 to n **do**
for j := 1 to i **do** t := 1 **od**
od
if n > 0 **then**
for i := 1 **to** 4 **do** f2(n div 2) **od**
fi
end proc

2. Dado un arreglo a : array[1..n] of nat se define una cima de a como un valor k en el intervalo 1, . . . , n tal que a[1..k] está ordenado crecientemente y a[k..n] está ordenado decrecientemente.

a. Escribí un algoritmo que determine si un arreglo dado tiene cima.

```
fun existe_cima (in/out a: array[1..n] of nat) ret h_cima : Bool
h_cima := true
var posible_cima : nat
var i : nat
i := 1
while i < n && a[i] < a[i + 1] do
   posible_cima := i + 1
   i := i + 1
od
while posible_cima < n && h_cima do
 h_cima := h_cima && a[posible_cima] > a[posible_cima + 1]
od
end fun
```

b. Escribí un algoritmo que encuentre la cima de un arreglo dado (asumiendo que efectivamente tiene una cima) utilizando una búsqueda secuencial, desde el comienzo del arreglo hacia el final.

```
fun devuelve_cima (in/out a: array[1..n] of nat) cima : Nat
var i : nat
i := 1
cima := 1
while i < n && a[i] < a[i + 1] do
  cima := cima + 1
od
end fun
```

c. Escribí un algoritmo que resuelva el mismo problema del inciso anterior utilizando la idea de búsqueda binaria.

```
fun binary_search_cima_rec (in/out a: array[1..n] of nat, izq, der : nat) ret cima : Nat
izq := 1
der := n
mid := (izq + der) / 2
if a[mid] < a[mid + 1] then
  cima := binary_search_cima (a, m + 1, der)
else
  cima := binary_search_cima (a, izq, mid)
fi
end fun

fun binary_search_cima (in/out a: array[1..n] of nat) ret cima : Nat
  cima := binary_search_cima_rec (a, 1, n)
end fun
```

3. El siguiente algoritmo calcula el mínimo elemento de un arreglo a : array[1..n] of nat mediante la técnica de programación divide y vencerás. Analiza la eficiencia de minimo(1, n).

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

Primero analizemos el algoritmo, para obtener los datos “a”, “b” y “g(n)” :

- El tamaño del problema es el tamaño del arreglo.
- Contamos las asignaciones a la variable m.
- En el caso simple, es decir cuando “i” = “k” (array de un solo elemento) el algoritmo hace una asignación.
- En el caso complejo, hace una asignación a “j” y se llama recursivamente a sí misma con la mitad del arreglo en cuestión.
- La otra operación además de la recursión es la asignación a “j”.

Entonces “**a**” = 2 (dos llamadas recursivas a mínimo), luego “**b**” = 2 (se divide a la mitad el tamaño original del problema) y “**g(n)**” = 1 (la asignación a j vale 1 por teoría). Ahora definamos t(n).

- ☐ **t(n)** = **1** si n = 1
  = **2 * t(n/2) + 1** si n > 1

Ahora comparemos “a” con “b^k” para determinar el orden de la función t(n). Tenemos que **a** = 2 y **b^k** = 2^0 por ende **a > b^k**, por resultado teórico sabemos que el orden de t(n) es **n^log_b(a).**

- ☐ La respuesta entonces es que t(n) es de orden n^log_2(2).

4. Ordena utilizando $\subset$ e $\approx$ los órdenes de las siguientes funciones. No calculles límites, utiliza las propiedades algebraicas.

a. n * log (2^n), 2^n * log(n), n! * log(n), 2^n
   - $n \subset n * \log(n) \rightarrow 2^n < 2^n * \log(n)$
   - $2^n \subset n! \rightarrow 2^n * \log(n) < n! * \log(n)$
   - $\log(n) \subset n \rightarrow \log(n^2) < n^2 \rightarrow n * \log(n^2) < n^2$ (constantes multiplicativas se desprecian)

Respuesta final : $n * \log (2^n) \subset 2^n \subset 2^n * \log(n) \subset n! * \log(n)$

b. n^4 + 2 * log(n), log(n^n^4), 2^(4 * log(n)), 4^n, n^3 * log(n)
   - $4^n \subset n^4 + 2 * \log(n)$
   - $n^3 * \log(n) \subset 4^n$
   - $2^{(4 * \log(n))} \subset 4^n$

Respuesta final : $2^{(4 * \log(n))} \subset \log(n^{n^4}) \subset n^3 * \log(n) \subset n^4 + 2 * \log(n) \subset 4^n$

c. log (n!), n *log(n), log(n^n)

Respuesta final : $n * \log(n) \approx \log(n^n) \subset \log (n!)$

5. Sean K y L constantes, y f el siguiente procedimiento:

proc f (**in** n : **nat**)
if n ≤ 1 **then**
skip
else
for i := 1 to K **do** f(n div L) **od**
for i := 1 to n^4 **do** operacíon de O(1) **od**
fi
end proc

Determina posibles valores de K y L de manera que el procedimiento tenga orden:

a. n^4 * log(n)

Para este inciso, necesitamos que a = b^k siendo k = 4. Escogemos a = 16 y b = 2 tal que cumpla la propiedad anterior, por ende tenemos :

proc f (**in** n : **nat**)
if n ≤ 1 **then**
skip
else
for i := 1 to 16 **do** f(n div 2) **od**
for i := 1 to n^4 **do** operacíon de O(1) **od**
fi
end proc

b. n^4

Necesitamos que a < b^k, siendo k = 4. Por tanto elijo a = 2 y b = 2. Termina quedando :

proc f (**in** n : **nat**)
if n ≤ 1 **then**
skip
else
for i := 1 to 2 **do** f(n div 2) **od**
for i := 1 to n^4 **do** operacíon de O(1) **od**
fi
end proc

c. n^5

Elijo a = 32 y b = 2 ya que log_2(32) es igual a 5 por ende necesito que se cumpla n^log_b(a) y esto se da cuando a > b^k → 32 > 2^4.

proc f (**in** n : **nat**)
if n ≤ 1 **then**
skip
else
for i := 1 to 32 **do** f(n div 2) **od**
for i := 1 to n^4 **do** operacíon de O(1) **od**
fi
end proc

Práctico 2 - Parte 1 - Tipos de datos

1. Escribir un algoritmo que dada una matriz a: array[1..n,1..m] of int calcule el elemento mínimo. Escribir otro algoritmo que devuelva un arreglo array[1..n] con el mínimo de cada fila de la matriz a.

```
fun (a: array[1..N, 1..M] of Int) ret minimo : Int
minimo := a[1,1]
for i := 1 to N do
    for j := 1 to M do
        if a[i,j] < minimo then
          minimo := a[i,j]
        else
          skip
        fi
    od
od
end fun

fun (a : array[1..N, 1..M] of Int) ret min_a : array[1..N]
var minimo : Int
var columna_n : Nat
columna_n := 1
minimo := a[1,1]
while columna_n < M do
for i := 1 to N do
    if a[i,1] < minimo then
      mínimo := a[i,1]
    else
      skip
    fi
min_a[i]:= minimo
od
columna_n := columna_n + 1
od
end fun
```

2. Dados los tipos enumerados :

```
type mes = enumerate
                enero
                febrero
                …
                diciembre
           end enumerate

type clima = enumerate
                Temp
```

TempMax
TempMin
Pres
Hum
Prec
end enumerate

El arreglo med:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat es un arreglo multidimensional que contiene todas las mediciones estadísticas del clima para la ciudad de Córdoba desde el 1/1/1980 hasta el 28/12/2016. Por ejemplo, med[2014,febrero,3,Pres] indica la presión atmosférica que se registró el día 3 de febrero de 2014. Todas las mediciones están expresadas con números enteros. Por simplicidad asumimos que todos los meses tienen 28 días.

a. Dar un algoritmo que obtenga la menor temperatura mínima (TempMin) histórica registrada en la ciudad de Córdoba según los datos del arreglo.
b. Dar un algoritmo que devuelva un arreglo que registre para cada año entre 1980 y 2016 la mayor temperatura máxima (TempMax) registrada durante ese año.
c. Dar un algoritmo que devuelva un arreglo que registre para cada año entre 1980 y 2016 el mes de ese año en que se registró la mayor cantidad mensual de precipitaciones (Prec).
d. Dar un algoritmo que utilice el arreglo devuelto en el inciso anterior (además de med) para obtener el año en que ese máximo mensual de precipitaciones fue mínimo (comparado con los de otros años).
e. Dar un algoritmo que obtenga el mismo resultado sin utilizar el del inciso (c).

a.

```
fun temp_min_historica (med : array[1980..2016,enero..diciembre,1..28,Temp..Prec] of Nat
ret TempMin : Nat
var temp_min : Nat
temp_min := med[1980,enero,1,Temp]
for  i := 1980 to 2016 do
    for j := enero to diciembre do
        for k := 1 to 28 do
            if med[i,j,k,Temp] < temp_min then
              TempMin := temp_min
            else
              skip
            fi
        od
    od
od
```

b.

```
fun temp_max_historica (med : array[1980..2016,enero..diciembre,1..28,Temp..Prec] of Nat
ret max_temp_per_year : [1980..2016] of Nat
var max_temp_y : Nat
max_temp_y := med[1980,enero,1,Temp]
for i := 1980 to 2016 do
    for j := enero to diciembre do
        for k := 1 to 28 do
            if med[i,j,k,Temp] > max_temp_y then
              max_temp_y := med[i,j,k,Temp]
            else
              skip
            fi
        od
    od
max_temp_per_year[i] := max_temp_y
od
end fun
```

c.

```
fun max_prec_mes (med : array[1980..2016,enero..diciembre,1..28,Temp..Prec] of Nat ret
max_prec_mes_a : [1980..2016] of Nat
var prec_dia : Nat
var prec_mensual : Nat
var max_mes_prec_ano : Nat
max_prec_aual := 0
for año := 1980 to 2016 do
    for mes := enero to diciembre do
        prec_mensual := 0
        for dia := 1 to 28 do
            prec_dia := prec_dia + med[año,mes,dia,Prec]
            prec_mensual := prec_mensual + prec_dia
        od
        if prec_mensual >  max_prec_anual  then
             max_prec_anual:= prec_mensual
             max_prec_mes_a[año] = mes
        else
             skip
        fi
     od
od
end fun
```

d.

e.

3. Dado el tipo :

```
type person = tuple
        name: String
        age: Nat
        weight: Nat
end tuple
```

a. Escribí un algoritmo que calcule la edad y peso promedio de un arreglo a : array[1..n] of person.
b. Escribí un algoritmo que ordene alfabéticamente dicho arreglo.

a.

```
type edad_peso_prom = tuple
        pesoProm : Nat
        edadProm : Nat
end tuple

fun edad_peso_promedio (a : array [1..N] of person) ret e_y_p_prom : edad_peso_prom
var peso_total : Nat
var edad_total : Nat
peso_total := 0
edad_total := 0
for i := 1 to N do
    peso_total := peso_total + a[i].weight
    edad_total := edad_total + a[i].age
od
e_y_p_prom.pesoProm := peso_total div N
e_y_p_prom.edadProm := edad_total div N
end fun

proc (in/out a : array [1..N] of person)
var i : nat
i := 0
while i < N do
    if a[i].name > a[i + 1].name then
      swap(a,i,i+1)
    else
      skip
    fi
i := i + 1
od
end fun
```

4. Dados dos punteros p, q : **pointer to int**

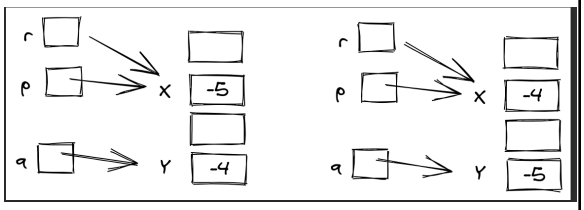
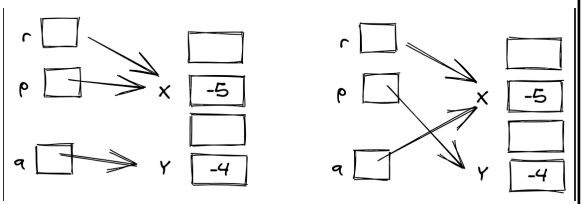
a. Escribir un algoritmo que intercambie los valores referidos sin modificar los valores de p y q.

```
proc swap_values_p (in/out p, q : pointer to int)
var int aux;
aux := *p
*p := *q
*q := aux
end proc
```

b. Escribir otro algoritmo que intercambie los valores de los punteros.

```
proc swap_mem_dir (in/out p, q : pointer to int)
var aux: pointer to int
aux := p
p := q
q := aux
end proc
```

Sea un tercer puntero r : pointer to int que inicialmente es igual a p, y asumiendo que inicialmente ∗p = 5 y ∗q = -4 ¿Cuales serian los valores de ∗p, ∗q y ∗r luego de ejecutar el algoritmo en cada uno de los dos casos?

Procedimiento a	Procedimiento b
p = Dirección (x) = 5 q = Dirección (y) = -4 r = Dirección (x) = 5 aux := 5 *p := -4 *q := 5 -- Valor de r luego de la ejecución -- *r = -4 	p = Dirección (x) = 5 q = Dirección (y) = -4 r = Dirección (x) = 5 *p = 5 *q = -4 aux := Dirección (x) p := Dirección (y) q := Dirección (x) -- Valor de r luego de la ejecución -- r = Dirección (x) = 5 

5. Dados dos arreglos a, b : array[1..n] of nat se dice que a es “lexicográficamente menor” que b si existe k ∈ {1 . . . n} tal que a[k] < b[k], y para todo i ∈ {1 . . . k − 1} se cumple a[i] = b[i]. En otras palabras, si en la primera posición en que a y b difieren, el valor de a es menor que el de b. También se dice que a es “lexicográficamente menor o igual” a b si a es lexicográficamente menor que b o a es igual a b.

a. Escribir un algoritmo lex less que recibe ambos arreglos y determina si a es lexicográficamente menor que b.

```
fun lex_less (a, b : array[1..N] of nat) ret is_lex_less : Bool
var i : nat
i := 1
is_lex_less := False
while i < N do
    if  a[i] != b[i] then
        if a[i] < b[i] then
            is_lex_less := True
            i : = N
        else
            i := N
        fi
    else
        i := i + 1
    fi
od
return is_lex_less
end fun
```

b. Escribir un algoritmo lex less or equal que recibe ambos arreglos y determina si a es lexicográficamente menor o igual a b.

```
fun lex_less (a, b : array[1..N] of nat) ret is_lex_less : Bool
var i : nat
i := 1
is_lex_less := True
while i < N do
    if  a[i] != b[i] then
        if a[i] < b[i] then
            i := N
        else
            is_lex_less := False
            i := N
        fi
    else
        i := i + 1
    fi
od
return is_lex_less
end fun
```

c. Dado el tipo enumerado
type Ord = enumerate
igual
menor
mayor
end enumerate
Escribir un algoritmo lex compare que recibe ambos arreglos y devuelve valores en el tipo ord. ¿Cuál es el interés de escribir este algoritmo?

```
fun lex_compare (a, b : array[1..N] of nat) ret ord_res : Ord
var i : Nat
i := 0
ord_res.igual = 0
ord_res.menor = 0
ord_res.mayor = 0
while i < N do
if a[i] > b[i] then
  ord_res.mayor := ord_res.mayor + 1
fi
if a[i] < b[i] then
  ord_res.menor := ord_res.menor + 1
fi
if a[i] == b[i] then
  ord_res.igual := ord_res.igual + 1
fi
i := i + 1
od
end fun
```

6. Escribir un algoritmo que dadas dos matrices a, b: array[1..n,1..m] of nat devuelva su suma.

```
fun sum_matrices (a, b: array[1..N,1..M] of nat) ret sum : Nat
sum := 0
for i := 1 to N do
    for j := 1 to M do
        sum := sum + (a[i,j] + b[i,j])
    od
od
end fun
```

7. Escribir un algoritmo que dadas dos matrices a: array[1..n,1..m] of nat y b: array[1..m,1..p] of nat devuelva su producto.

```
fun prod_matrices (a : array[1..N,1..M] of nat, b : array[1..M,1..P] of Nat) ret prod : Nat
prod := 0
for i := 1 to N do
    for k := 1 to P
        for j := 1 to M do
        prod := prod + (a[i,j] * b[j,k])
        od
     od
od
end fun
```

Práctico 2 - Parte 2

1. Completa la implementación de listas dada en el teórico usando punteros.

```
spec List of T where
        constructors
                fun empty( ) ret l : List of T
                proc addl (in e : T, in/out l : List of T)
        destroy
                 proc destroy (in/out l : List of T)
        operations
                fun is_empty (l : List of T) ret b : bool
                fun head (l : List of T) ret e : T
                proc tail (in/out l : List of T)
                proc addr (in/out l : List of T,in e : T)
                fun length (l : List of T) ret n : nat
                proc concat (in/out l : List of T,in l0 : List of T)
                fun index (l : List of T,n : nat) ret e : T
                proc take (in/out l : List of T,in n : nat)
                proc drop (in/out l : List of T,in n : nat)
                fun copy_list (l1 : List of T) ret l2 : List of T

implement List of T where
        type Node of T = tuple
                                elem : T
                                next : pointer to (Node of T)
                        end tuple

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

fun empty( ) ret l : List of T
        l := null
end fun
```

```
proc addl (in e : T, in/out l : List of T)
        var p : pointer to (Node of T)
        alloc(p)
        p->elem := e
        p->next := l
        l := p
end proc

fun is_empty(l : List of T) ret b : bool
        b := ( l = null )
end fun

{- PRE: not is_empty(l) -}
fun head(l : List of T) ret e : T
        e := l-> elem
end fun

{- PRE: not is_empty(l) -}
proc tail(in/out l : List of T)
        var p : pointer to (Node of T)
        p := l
        l := l->next
        free(p)
end proc

proc addr (in/out l : List of T,in e : T)
        var p,q : pointer to (Node of T)
        alloc(q)
        q->elem := e
        q->next := null
        if (not is_empty(l)) then
                p := l
                while (p->next != null) do
                        p := p->next
                od
                p->next := q
        else
                l := q
        fi
end proc

fun length(l : List of T) ret n : nat
        var p : pointer to (Node of T)
        n := 0
        p := l
        while (p != null) do
                n := n+1
                p := p->next
```

```
        od
end fun

proc concat(in/out l : List of T,in l0 : List of T)
        var p : pointer to (Node of T)
        p := l
        while (p ->next != null) do
                p := p->next
        od
        p->next := l0
end fun

{- PRE: length(l) > n -}
fun index(l : List of T,n : nat) ret e : T
        var counter : nat
        var p : pointer to (Node of T)
        p := l
        counter := 0
        while (p->next != null && counter != n) do
                p := p->next
                counter := counter + 1
        od
        e := p->elem
end fun

proc take(in/out l : List of T,in n : nat)
        var p : pointer to (Node of T)
        var q : pointer to (Node of T)
        p := l
        for counter := 0 to n -1 do
                if (p->next != null) then
                        p := p->next
                fi
        od
        p := p->next
        while (p != null) do
                q := p
                p := p->next
                free(q)
        od
end fun

proc drop(in/out l : List of T,in n : nat)
        var p : pointer to (Node of T)
        var q : pointer to (Node of T)
        var counter : nat
        counter := 0
        p := l
```

```
        while (p != null && counter != n) do
                q := p
                p := p->next
                free(q)
        od
end fun

fun copy_list(l1 : List of T) ret l2 : List of T
        var p,q : pointer to (Node of T)
        var saved_elem : T
        p := l1
        q := l2
        while (p->next != null && q->next != null) do
                saved_elem := p->elem
                p := p->next
                q->elem := saved_elem
                q := q->next
        od
end fun

proc destroy (in/out l : List of T)
        var p,q : pointer to (Node of T)
        p := l
        while (p->next != null) do
        q := p
        p := p->next
        free(q)
        od
end fun
```

2. Dada una constante natural N, implementa el TAD Lista de elementos de tipo T, usando un arreglo de tamaño N y un natural que indica cuántos elementos del arreglo son ocupados por elementos de la lista. ¿Esta implementación impone nuevas restricciones? ¿En qué función o procedimiento tenemos una nueva precondición?

```
spec List of T where
        constructors
                fun empty( ) ret l : List of T
                proc addl (in e : T, in/out l : array[1..n] of T)

        operations
                fun is_empty (l : List of T) ret b : bool
                proc addl (in e : T , in/out l : List of T)
                fun head (l : List of T) ret e : T
                proc tail (in/out l : List of T)
                proc addr (in/out l : List of T,in e : T)
                fun length (l : List of T) ret n : nat
```

```
        proc concat (in/out l0 : List of T,in l1 : List of T)
        fun index (l : List of T,n : nat) ret e : T
        proc take (in/out l : List of T,in n : nat)
        proc drop (in/out l : List of T,in n : nat)
        fun copy_list (l1 : List of T) ret l2 : List of T
        proc destroy (in/out l : List of T)

implement List of T where

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

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

{- PRE: l.size < N -}
proc addl (in e : T , in/out l : List of T)
    if l.size = 0 then
        a[0] := e
    else
        for i := 1 to l.size do
            l.elems[i + 1] l.elems[i]
        od
        a[1] := e
        l.size := l.size + 1
    fi
end fun

fun is_empty (l : List of T) ret b : bool
    b := (l.size = 0)
end fun



fun head (l : List of T) ret e : T
    e := l.elems[1]
end fun

proc tail (in/out l : List of T)
    for i := 1 to l.size - 1 do
        l.elems[i] :=l.elems[i + 1]
    od
    l.size := l.size - 1
end fun
```

```
proc addr (in/out l : List of T, in e : T)
        if l.size = 0 then
                a[1] := e
        else
                l.size := l.size + 1
                l.elems[l.size] := e
        fi
end fun

fun length (l : List of T) ret n : nat
        n := l.size
end fun

{- PRE: l0.size + l1.size < N -}
proc concat (in/out l0 : List of T, in l1 : List of T)
var total_size : nat
total_size := l0.size + l1.size
        for l0.size to total_size  do
                l0.elems[l0.size + 1] := l1.elems[l1.size]
        od
        l0.size := total_size
end fun

fun index (l : List of T,n : nat) ret e : T
        e := l.elems[n]
end fun

proc take (in/out l : List of T,in n : nat)
        l.size := n
end fun

proc drop (in/out l : List of T,in n : nat)
        for i := 1 to l.size - n do
                l.elems[i] := l.elems[n + i]
        od
        l.size := l.size - n
end fun
fun copy_list (l1 : List of T) ret l2 : List of T
        for i := to l1.size do
                l2.elems[i] := l1.elems[i]
        od
        l2.size := l1.size
end fun

proc destroy (in/out l : List of T)
        l.size := 0
end proc
```

3. Implementa el procedimiento add at que toma una lista de tipo T, un natural n, un elemento e de tipo T, y agrega el elemento e en la posición n, quedando todos los elementos siguientes a continuación. Esta operación tiene como precondición que n sea menor a lo largo de la lista. AYUDA: Puede ayudarte usar las operaciones copy, take y drop.

```
proc add_at (in/out l : List of T, in n : nat, in e : T)
        var aux : List of T
        copy(l,aux)
        drop(aux,n - 1)
        take(l, n - 1)
        addr(l,e)
        concat(l, aux)
end proc
```

4.

a. Especifica un TAD tablero para mantener el tanteador en contiendas deportivas entre dos equipos (equipo A y equipo B). Debería tener un constructor para el comienzo del partido (tanteador inicial), un constructor para registrar un nuevo tanto del equipo A y uno para registrar un nuevo tanto del equipo B. El tablero solo registra el estado actual del tanteador, por lo tanto el orden en que se fueron anotando los tantos es irrelevante. Además se requiere operaciones para comprobar si el tanteador está en cero, si el equipo A ha anotado algún tanto, si el equipo B ha anotado algún tanto, una que devuelva verdadero si y s´olo si el equipo A va ganando, otra que devuelva verdadero si y s´olo si el equipo B va ganando, y una que devuelva verdadero si y sólo si se registra un empate. Finalmente habrá una operación que permita anotarle un número n de tantos a un equipo y otra que permita “castigarlo” restando un número n de tantos. En este último caso, si se le restan más tantos de los acumulados equivaldría a no haber anotado ninguno desde el comienzo del partido.
b. Implementa el TAD Tablero utilizando una tupla con dos contadores: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B.
c. Implementa el TAD Tablero utilizando una tupla con dos naturales: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B. ¿Hay alguna diferencia con la implementación del inciso anterior? ¿Alguna operación puede resolverse más eficientemente?

b.

```
spec tablero where
        constructors
                fun tab_inic () ret res : tablero
                proc add_point__A (in/out res : tablero)
                proc add_point__B (in/out res : tablero)

        operations
                fun is_tab_inic (res : tablero) ret b : bool
                fun are_any_point_A (res : tablero) ret b : bool
                fun are_any_point_B (res : tablero) ret b : bool
                fun A_is_winning (res : tablero) ret b : Bool
```

```
                fun B_is_winning (res : tablero) ret b : Bool
                fun is_tie (res: tablero) ret b : Bool
end fun

implement tablero where
type tablero = tuple
                counter_A = Counter
                counter_B = Counter
        end tuple

fun tab_inic () ret res : tablero
        init(res.counter_A)
        init(res.counter_B)
end fun

proc add_point_A (in/out res : tablero)
        res.counter_A := inc(res.counter_A)
end fun

proc add_point_B (in/out res : tablero)
        res.counter_B := inc(res.counter_B)
end fun

fun is_tab_inic (res : tablero) ret b : bool
        b := is_init(res.counter_A)  && is_init(res.counter_B)
end fun

fun are_any_point_A (res : tablero) ret b : bool
        b := !is_init(res.counter_A)
end fun

fun are_any_point_B (res : tablero) ret b : bool
        b := !is_init(res.counter_B)
end fun

fun A_is_winning (res : tablero) ret b : Bool
        b := res.counter_A > res.counter_B
end fun
fun B_is_winning (res : tablero) ret b : Bool
        b := res.counter_B > res.counter_A
end fun

fun is_tie (res: tablero) ret b : Bool
        b := res.counter_A = res.counter_B
end fun
```

c.

```
implement tablero where
```

```
type tablero = tuple
                    counter_A = nat
                    counter_B = nat
               end tuple

fun tab_inic () ret res : tablero
        res.counter_A := 0
        res.counter_B := 0
end fun

proc add_point_A (in/out res : tablero)
        res.counter_A := res.counter_A + 1
end fun

proc add_point_B (in/out res : tablero)
        res.counter_B := res.counter_B + 1
end fun

fun is_tab_inic (res : tablero) ret b : bool
        b := res.counter_A = 0  && res.counter_B = 0
end fun

fun are_any_point_A (res : tablero) ret b : bool
        b := res.counter_A) != 0
end fun

fun are_any_point_B (res : tablero) ret b : bool
        b := res.counter_B != 0
end fun

fun A_is_winning (res : tablero) ret b : Bool
        b := res.counter_A > res.counter_B
end fun

fun B_is_winning (res : tablero) ret b : Bool
        b := res.counter_B > res.counter_A
end fun

fun is_tie (res: tablero) ret b : Bool
        b := res.counter_A = res.counter_B
end fun
```

5. Especifica el TAD Conjunto finito de elementos de tipo T. Como constructores consideran el conjunto vacío y el que agrega un elemento a un conjunto. Como operaciones: una que chequee si un elemento pertenece a un conjunto c, una que chequee si un conjunto es vaćıo, la operación de unir un conjunto a otro , insersectar un conjunto con otro y obtener la diferencia. Estas últimas tres operaciones deberán especificarse como procedimientos que toman dos conjuntos y modifican el primero de ellos.

```
spec finit_set of T where
        constructors
                fun empty_set () ret set : finit_set of T
                proc add_element_to_set  (in/out set : finit_set of T, in e : T)
        operations
                fun bellows_to_set (set : finit_set of T, e : T) ret b : Bool
                fun is_empty_set (set : finit_set of T) ret b : Bool
                proc union_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
                proc intersect_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
                proc diff_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
end spec
```

6. Implementar el TAD Conjunto finito de elementos de tipo T utilizando:
   a. Una lista de elementos de tipo T, donde el constructor para agregar elementos al conjunto se implementa directamente con el constructor addl de las listas.
   b. Una lista de elementos de tipo T, donde se asegure siempre que la lista está ordenada crecientemente y no tiene elementos repetidos. Debes tener cuidado especialmente con el constructor de agregar elementos y las operaciones de unión, intersección y diferencia. A la propiedad de mantener siempre la lista ordenada y sin repeticiones le llamamos invariante de representación. Ayuda: Para implementar el constructor de agregar elemento puede serte muy útil la operación add at implementada en el punto 3.

a.

```
implement finit_set of T where

type finit_set of T = List of T

fun empty_set () ret set : finit_set of T
        set := empty_list()
end fun

proc add_element_to_set  (in/out set : finit_set of T, in e : T)
        set := addl(e,set)
end proc

fun bellows_to_set (set : finit_set of T, e : T) ret b : Bool
        var size_set : Nat
        var actual_elem : T
        var i : nat
        b := False
        i := 0
        size_set := length(set)
        while i < size_set do
                actual_elem := index(set,i)
```

```
                if actual_elem = e then
                        b := True
                        i := size_set
                fi
        od
end fun

fun is_empty_set (set : finit_set of T) ret b : Bool
        b := is_empty(set)
end fun



proc union_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
        concat(set1,set2)
end fun

{-PRE : length(set1) = length(set2) -}
proc intersect_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
        var size_set1, size_set2, i, j : Nat
        var actual_elem1, actual_elem2 : T
        i, j := 0
        size_set1 := length(set1)
        while i < size_set1 do
                actual_elem1 := index(set1, i)
                while j < size_set2 do
                        actual_elem2 := index(set2, i)
                        if actual_elem1 = actual_elem2 then
                                addr(set1, actual_elem1)
                        else
                                j := j + 1
                od
                i := i + 1
        od
        drop(set1,length(size_set1))
        end fun

proc diff_set (in/out set1 : finit_set of T, in set2 : finit_set of T)
{-PRE : length(set1) = length(set2) -}
end fun
```

Práctico 2 - Parte 3

1. Implementa el TAD Pila utilizando la siguiente representación:

```
implement Stack of T where

type Stack of T = List of T

fun empty_stack() ret s : Stack of T
        s := empty()
end fun

proc push (in e : T, in/out s : Stack of T)
        addl(e, s)
end fun

fun is_empty_stack(s : Stack of T) ret b : Bool
        b := is_empty(s)
end fun

fun top(s : Stack of T) ret e : T
        e := head(s)
end fun
```

```
proc pop (in/out s : Stack of T)
        tail(s)
end fun
```

2. Implementar el TAD Pila utilizando la siguiente representación:

```
implement Stack of T where

type Node of T = tuple
                elem : T
                next : pointer to (Node of T)
        end tuple

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

fun empty_stack() ret s : Stack of T
        s := Null
end fun
```

```
proc push (in e : T, in/out s : Stack of T)
        var p : pointer to (Node of T)
        alloc(p)
        p->elem := e
        p->next := s
        s := p
end fun

fun is_empty_stack(s : Stack of T) ret b : Bool
        b := (s = Null)
end fun

fun top(s : Stack of T) ret e : T
        e := s-> elem
end fun

{- PRE: not is_empty_stack(s) -}
proc pop (in/out s : Stack of T)
        var p : pointer to (Node of T)
        p := s
        s := s->next
        free(p)
end proc
```

3.

a. Implementar el TAD Cola utilizando la siguiente representación, donde N es una constante de tipo nat:

```
implement Queue of T where
        type Queue of T = tuple
                elems : array[0..N-1] of T
                size : Nat
        end tuple
```

b. Implementar el TAD Cola utilizando un arreglo como en el inciso anterior, pero asegurando que todas las operaciones estén implementadas en orden constante.

Ayuda 1: Quizás convenga agregar algún campo más a la tupla. ¿Estamos obligados a que el primer elemento de la cola esté representado con el primer elemento del arreglo?
Ayuda2: Buscar en Google aritmética modular.

a.

```
implement Queue of T where
type Queue of T = tuple
                elems : array[0..N-1] of T
                size : Nat
        end tuple

fun empty_queue() ret q : Queue of T
        q.size := 0
end fun
proc enqueue (in/out q : Queue of T, in e : T)
        q.elems[q.size] := e
        q.size := q.size + 1
end fun

fun is_empty_queue(q : Queue of T) ret b : Bool
        b := q.size = 0
end fun

{- PRE: not is_empty_queue(q) -}
fun first(q : Queue of T) ret e : T
        e := q.elems[0]
end fun

{- PRE: not is_empty_queue(q) -}
proc dequeue (in/out q : Queue of T)
        for i := 0 to q.size - 2 do
                q.elems[i] := e.elems[i + 1]
        od
        q.size := q.size - 1
end fun
```

b.

```
implement Queue of T where
type Queue of T = tuple
                    elems : array[0..N-1] of T
                    size : Nat
                    first : Nat
          end tuple

fun empty_queue() ret q : Queue of T
          q.size := 0
end fun
proc enqueue (in/out q : Queue of T, in e : T)
          q.elems[q.size] := e
          q.size := q.size + 1
end fun

fun is_empty_queue(q : Queue of T) ret b : Bool
          b := q.size = 0
end fun

{- PRE: not is_empty_queue(q) -}
fun first(q : Queue of T) ret e : T
          e := q.elems[q.first]
end fun

{- PRE: not is_empty_queue(q) -}
proc dequeue (in/out q : Queue of T)
          q.size := q.size - 1
          q.first := (q.first + 1) % N
end fun
```

4. Completa la implementación del tipo Árbol Binario dada en el teórico, donde utilizamos la siguiente representación:

```
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 empty_tree () ret t : Tree of T
          t := null
end fun

fun node (tl : Tree of T, e : T, tr : Tree of T) ret t : Tree of T
          alloc(t)
          t->value := e
          t->left := tl
```

```
        t->right := tr
end fun

fun is_empty_tree () ret b : Bool
        b := (t = null)
end fun

fun root (t : Tree of T) ret e : T
        e := t->value
end fun

{- PRE: not is_empty_tree (t) -}
fun left (t : Tree of T) ret tl : Tree of T
        tl := t->left
end fun

{- PRE: not is_empty_tree (t) -}
fun right (t : Tree of T) ret tr : Tree of T
        tr := t->right
end fun



fun height (t : Tree of T) ret n : Nat
        if is_empty_tree (t) then
                n := 0
        else
                n := (height(t->right) max height (t->left)) + 1
        fi
end fun

fun is_path(t : Tree of T, p : Path) ret b : Bool
        var p : pointer to (Node of T)
        p := t
        if is_empty_tree(t) or is_empty_list(p) then
                b := False
        else
                for i := 0 to length(p) - 1 do
                        if p = Null then
                                b := False
                        else if  index(p,i) = Left then
                                p := p->left
                        else
                                p := p->right
                        fi
                od
        b := True
```

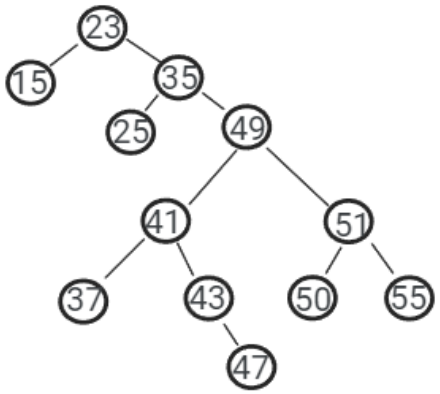
```
        fi
end fun

fun subtree_at (t : Tree of T, p : Path) ret res : Tree of T
        var p_aux : Path
        if is_empty_tree(t) or is_empty(p) then
                res := t
        else
                p_aux := copy_list(p)
                tail(p_aux)
        if head(p) = Left then
                res:= subtree_at(t->left, p_aux)
        else
                res:= subtree_at(t->right, p_aux)
        fi
        destroy(p_aux)
        fi
end fun
```

```
{- PRE: is_path(t,p) -}
fun elem_at(t : Tree of T, p : Path) ret e : T
        var p : pointer to (Node of T)
        p := t
        for i := 0 to length(p) - 1 do
                index(p,i) = Left then
                        p := p->left
                else
                        p := p->right
        od
        e := p->value
end fun
```

6. En un ABB cuyos nodos poseen valores entre 1 y 1000, interesa encontrar el número 363. ¿Cuáles de las siguientes secuencias no puede ser una secuencia de nodos examinados según el algoritmo de búsqueda? ¿Por qué?

a. 2, 252, 401, 398, 330, 344, 397, 363. Es válido
b. 924, 220, 911, 244, 898, 258, 362, 363 Es válido
c. 925, 202, 911, 240, 912, 245, 363 No es válido
d. 2, 399, 387, 219, 266, 382, 381, 278, 363. Es válido
e. 935, 278, 347, 621, 299, 392, 358, 363. No es válido

7. Dada la secuencia de números 23, 35, 49, 51, 41, 25, 50, 43, 55, 15, 47 y 37, determinar el ABB que resulta al insertarlos exactamente en ese orden a partir del ABB vacío.



8. Determinar al menos dos secuencias de inserciones que den lugar al siguiente ABB:

a. 22, 10, 6, 4, 19, 15, 11, 18, 35, 25, 33, 52, 49, 53
b. 22, 35, 52, 53, 49, 25, 33, 10, 19, 15, 18, 11, 5, 4

Repaso primer parcial

- Se conoce como cocktail sort a una variación del algoritmo por selección, que consiste en obtener en cada pasada del arreglo, el elemento máximo y el elemento mínimo, intercambiándolos luego por las posiciones primera y última respectivamente. Implementa de manera precisa el algoritmo, usando el lenguaje de la materia :

```
proc cocktail_sort (in/out a: array[1..n] of T)
        var minp, maxp, actual_n: nat
        actual_n := n
        for i:= 1 to n do
                minp := i
                maxp := i
                for j:= i+1 to n do
                        if a[j] < a[minp] then
                                minp:= j
                        fi
                        if a[j] > a[maxp] then
                                maxp := j
                        fi
                od
        swap(a,i,minp)
        swap(a,actual_n,maxp)
        actual_n := actual_n - 1
        od
end proc
```

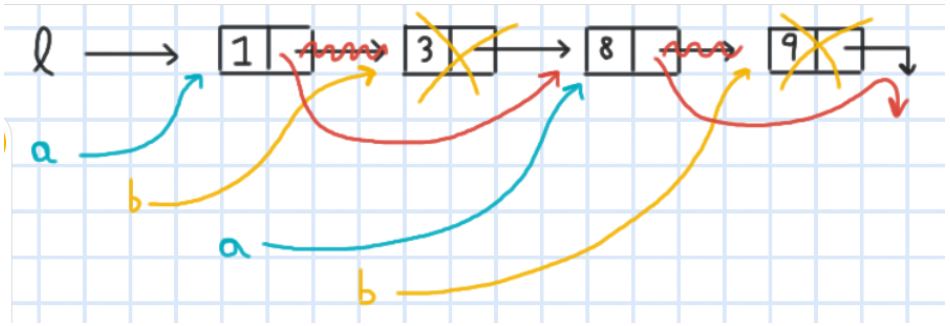
- Dado el siguiente algoritmo :

```
proc p (in/out l : list of T)
        var a, b : pointer to (Node of T)
        a := l
        while a != null do
                b := a->next
                if b != null then
                        a->next := b->next
                        free(b)
                fi
                a := a->next
        od
end proc

type node = tuple
                value : T
                next : pointer to (Node of T)
        end tuple
```

type List **of** T = **pointer to** (Node **of** T)

a. Explica que hace el procedimiento p.
b. Indicar el orden el del procedimiento p.
c. Si se llama al procedimiento p con una lista que tiene los valores 1,2,3,4,5,6 y 7, qué valores tendrá la lista luego de la ejecución del mismo?

a. Para entender qué hace, demos un ejemplo de corrida con la lista 1, 3, 8 y 9.



Luego, el algoritmo elimina los elementos en las posiciones impares de la lista.

b. Si analizamos un poco el procedimiento, vemos que contiene un bucle while el cual tiene la guarda a != null. Podemos darnos cuenta que el bucle se itera dependiendo la cantidad de elementos de la lista, osea seria de orden lineal. Puesto que a será null cuando lleguemos al último nodo de la lista, el cual su campo next será null, provocando que la guarda sea falsa y no entre en una nueva iteración. Por tanto el orden de p es n.
c. Con la lista [1,2,3,4,5,6,7], luego de ejecutar el procedimiento tendríamos algo de la pinta :

[1,2,3,4,5,6,7] = [1,3,5,7]

- Dada la siguiente función :

```
fun f (n : Nat) ret m : Nat
        if n <= 1 then
                m := 2 * n
        else
                m := 1
                for i := n downto 1 do
                        m := n * m
                od
                m : =3 * f (n div 2)
        fi
end fun
```

a. ¿Cuántas llamadas recursivas a f(n div 2) se realizan durante la ejecución de f(n)?
b. Expresa la ecuación de recurrencia en función a la cantidad de asignaciones a la variable m.
c. Célula el orden de asignaciones a la variable m.

a. El número de llamadas recursivas con f(n div 2) es 1.
b. t (n) = 1 para n <= 1
1 * t(n div 2) + (n + 1) para n > 1

Esto, debido a que la función presenta todas las características de un divide y vencerás, haciendo algo simple para una entrada pequeña (m := 2 * n) y en el caso más complejo, hace una llamada recursiva con el problema reducido (f(n div 2)) y ejecuta un for “n” veces el cual contiene una asignación (m := n * m).

c. Sea a = 1, b = 2, y k = 1 (sale de g(n) = n). Llegamos a que :
a < b^k → 1 < 2 → es de orden n^k → n

- Considere la especificación del tipo Listas de algún tipo T, vista en el teórico :

Implementa una función que reciba dos listas de enteros y devuelva otra lista que contenga en las posiciones pares todos los elementos de la primera lista y en las posiciones impares todos los elementos de la segunda lista.

```
fun fstpar_sndimpar_list (l1 : List of Int, l2 : List of Int) ret res : List of Int
    res := empty()
    for i = 0 to (length(l1) - 1) do
        addr(res, index(l1, i))
        addr(res, index(l2, i))
    od
end fun
```

- El juevo “Martín Pescador Extreme” consiste en una serie de "rondas" y comienza con una cola de N jugadores que inicialmente tendrán h vidas cada uno. En cada ronda se re-encola (se manda al fondo) k veces a las personas que se encuentren al principio de la cola y, al terminar la ronda, se le quita una vida al jugador que haya quedado al principio de la cola y se evalúa lo siguiente sobre este jugador:
  - ☐ 1. Si tiene al menos una vida, sigue en el juego y queda en el mismo lugar.
  - ☐ 2. Si no tiene más vidas se lo elimina de la cola (pierde) Al finalizar esta evaluación comienza una nueva ronda. El juego termina cuando queda una sola persona en la cola, a quien consideramos ganador/a.

  Nota: El número k es fijo durante todo el juego, es decir, es el mismo en cada ronda.

Completá la implementación del algoritmo ExtremeKingfisherGame en la cual deberás utilizar los TADs Player y Queue of Player sin romper la abstracción. Sólo se pueden utilizar operaciones que estén especificadas en los respectivos TADs.

```
fun ExtremeKingfisherGame(playersNames: array[1..N] of String, lives: Nat, k: Nat) ret winner: String
    var q: Queue of Player
    {- completar -}
end fun
```

La especificación del TAD Player es como sigue:

spec Player **where**
constructors

```
        fun createPlayer(name: String, lives: Nat) ret Player
operations
        fun getName(p: Player) ret n: String
        {- Devuelve el nombre del jugador p -}

        fun isAlive(p: Player) ret b: Bool
        {- Devuelve True si el jugador p tiene al menos 1 vida -}

        proc loseLife(in/out p: Player)
        {- Resta una vida al jugador p -}
        {- PRE: getLives(p) > 0 -}
end spec

fun ExtremeKingfisherGame(playersNames: array[1..N] of String, lives: Nat, k: Nat) ret
winner: String
        var q, aux: Queue of Player
        var amount_of_players : Nat
        var actual_player : Player
        amount_of_players := N
        for i := 1 to N do
                actual_player := createPlayer (playersNames[i], lives)
                enqueue(q, actual_player)
        od
        copy(q, aux)
        while (amount_of_players > 1) do
                for j := 1 to k do
                        re_enqueue(aux, first(aux))
                od
                loseLife(first(aux))
                if !isAlive(first(aux)) then
                        dequeue(aux)
                        amounth_of_players := amount_of_players - 1
                fi
        od
        winner := getName(first(aux))
end fun
```

- Escriba un algoritmo que, dado un arreglo y un elemento e que no pertenece al mismo, ubique los menores a e al comienzo del arreglo y devuelva la cantidad de elementos menores a e. Utilice para ello el siguiente encabezado:

  proc p (**in/out** a : array[1..n] **of** int,**in** e : int, **out** k: nat)

```
proc p (in/out a : array[1..N] of int,in e : int, out k: nat)
        var actual_seg : Nat
```

```
        actual_seg_init := 1
        k := 0
        for i := 1 to N do
                if a[i] < e then
                        k := k + 1
                        swap(a, actual_seg, i)
                        actual_seg := actual_seg + 1
                fi
        od
end fun
```

- Para cada una de los siguientes algoritmos determinar por separado cada uno de los siguientes incisos
    1. ¿Qué hace?
    2. ¿Cómo lo hace?
    3. El orden del algoritmo.

```
fun f(a: array[1..n] of nat) ret b: bool
        var i : nat
        i := 1
        while (i < n) ∧ (a[i] ≤ a[i+1]) do
                i:= i+1
        od
        b:= (i = n)
end fun

proc p(in/out a: array[0..n] of nat)
        for i:= 0 to n/2 do
                swap(a,i,n-i)
        od
end proc

proc q(in/out a: array[0..n,0..n] of nat)
        var tmp: nat
        for i:= 0 to n do
                for j:= 0 to n/2 do
                        tmp:= a[i,j]
                        a[i,j]:= a[i,n-j]
                        a[i,n-j]:= tmp
                od
        od
end proc
```

Un programa de computación simbólica permite a sus usuarios manipular polinomios con coeficientes enteros.

Para ello se utilizará un TAD que entre sus operaciones incluye evaluar en la indeterminada x, devolver el
Coeficiente de grado k, obtener el grado del polinomio y sumar dos polinomios. Especifica el TAD. Se puede
definir como constructores uno que devuelve el polinomio nulo y un segundo que suma un monomio (de la
forma ax^k) a un polinomio.

```
spec math_polinomios where
        constructors
                fun null_pol (   ) ret p : math_polinomios
                proc (in/out p : math_polinomios, in q: math_polinomios)
        operations
                fun evaluar (p : math_polinimios , x : int) ret res : int
                fun grado_k (p : math_polinomios, g : grado) ret res : nat
                fun grado_polinomio (p : math_polinomios) ret res : nat
                fun sum_polinomios (p,q : math_polinomios) ret res : int
end fun
```

Escribí una variante del algoritmo de ordenación por selección que vaya ordenando desde la última celda del arreglo hacia la primera. El resultado debe ser el mismo, es decir, el arreglo resultante debe estar ordenado en forma creciente, pero el modo de hacerlo debe ser diferente: en cada paso se debe seleccionar el máximo elemento aún no ordenado y colocarlo en la posición que corresponda desde el extremo nal del arreglo.

```
proc reverse_selection_sort (in/out a : array[1..N] of Int)
        var maxp : Nat
        for i = N downto 1 do
                maxp := i
                for j := i - 1 downto 1 do
                        if a[maxp] < a[j] then
                                maxp := j
                        fi
                od
                swap(a,maxp,i)
        od
end fun
```

Escribí una variante del algoritmo de ordenacion por inserción que vaya ordenando desde la última celda del arreglo hacia la primera. El resultado debe ser el mismo, es decir, el arreglo resultante debe estar ordenado en forma creciente, pero el modo de hacerlo debe ser diferente: en cada paso se debe seleccionar el máximo elemento aún no ordenado y colocarlo en la posición que corresponda desde el extremo nal del arreglo.

```
proc reverse_insertion_sort (in/out a : array[1..N] of Int)
        for i := N downto 2 do
                j := i
                while (j > 1) && (a[j] < a[j - 1]) do
                        swap(a,j,j - 1)
```

```
                    j : = j - 1
                od
        od
end proc
```

Práctico 3.1 - Técnicas de Programación

1. Demostrar que el algoritmo voraz para el problema de la mochila sin fragmentación no siempre obtiene la solución óptima. Para ello puede modificar el algoritmo visto en clase de manera de que no permita fragmentación y encontrar un ejemplo para el cual no halla la solución óptima.

```
type Objeto = tuple
        id : Nat
        value: Float
        weight: Float
end tuple

type  Obj_Mochila = tuple
        obj : Objeto
end tuple

fun mochila(W: Float, C: Set of Objeto) ret L : List of Obj_Mochila
        var o_m : Obj_Mochila
        var resto : Nat
        var C_aux : Set of Objeto
        L := empty_list()
        C_aux:= set_copy(C)
        resto := W
        do (resto > 0)
                o_m.obj := select_obj(C_aux)
                if o_m.obj.weight <= resto then
                        resto := resto - o_m.obj.weight
                        addl(L, o_m)
                        elim(C_aux, o_m.obj)
                fi
        od
        set_destroy(C_aux)
end fun

fun select_obj(C: Set of Objeto) ret r : Objeto
        var C_aux : Set of Objeto
        var o : Objeto
        var m : Float
        m := -∞
        C_aux := set_copy(C)
        do (not is_empty_set(C_aux))
                o := get(C_aux)
                if (o.value div o.weight > m) then
                        m := o.value div o.weight
                        r := o
                fi
                elim(C_aux,o)
```

```
        od
        set_destroy(C_aux)
end fun
```

Un contra-ejemplo que demuestra que este problema no admite solución voraz es el siguiente :

- Mochila con capacidad de 10 kg.
- Set of Objeto = ((Iphone, 12, 6), (Samsung, 11, 5), (Xiaomi, 8, 4))

Luego el algoritmo seleccionara el objeto “Samsung” ya que su valor relativo es de 2,2 y posteriormente el objeto “Xiaomi” ya que no puede seleccionar “Iphone” porque sobrepasa la capacidad. Es fácil ver que esta no es la solución óptima puesto que suma un total de 19 de valor, mientras que si elegía “Iphone” y “Xiaomi” sumaba un total de 20 de valor.

2. Considere el problema de dar cambio. Pruebe o de 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 :

- set of Moneda = {1, 5, 11}
- m (Monto a pagar) = 15

Es claro que la solución arrojada por el algoritmo es pagar 15 con una moneda de 11 y cuatro de 1, usando un total de 5 monedas, cuando es más factible pagar utilizando tres de 5.

3. Se desea realizar un viaje en un automóvil con autonomía A (en kilómetros), desde la localidad l0 hasta la localidad ln pasando por las localidades l1, . . . , ln−1 en ese orden. Se conoce cada distancia d ≤ A entre la localidad li−1 y la localidad li (para 1 ≤ i ≤ n), y se sabe que existe una estación de combustible en cada una de las localidades. Escribir un algoritmo que compute el menor número de veces que es necesario cargar combustible para realizar el viaje, y las localidades donde se realizaría la carga. Suponer que inicialmente el tanque de combustible se encuentra vacío y que todas las estaciones de servicio cuentan con suficiente combustible.

    - Criterio de búsqueda : di > Autonomía restante
    - Pensemos las localidades como una lista, y la autonomía A como un Nat. Las distancias entre localidades serán dadas por un array[1..N].
    - El algoritmo primero registrara la autonomía actual, y dependiendo de ella computará cual sera la localidad l_i en donde se realizará la carga. Luego, se agregara esa localidad a la lista resultado, y sumará 1 a una variable en caso de realizar una carga.

```
type res = tuple
        number_of_rechanges : Nat
        localities : List of Nat
end tuple
```

```
fun where_to_recharge_o (a_localities : array[1..N] of Nat, autonomy : Nat) ret
greedy_car_problem_res : res
        var actual_a : Nat
        var aux_res : res
        actual_a := autonomy
        aux_res.number_of_rechanges := 1
        addr(aux_res.localities, 0)
        for i := 1 to N - 1 do
                if a_localities[i + 1]  > actual_a then
                        aux_res.number_of_rechanges := aux_res.number_of_rechanges + 1
                        addr(aux_res.localities, i)
                        actual_a := autonomy - a_localities[i + 1]
                else
                        actual_a := actual_a - a_localities[i + 1]
                fi
        od
        greedy_car_problem_res := aux_res
end fun
```

4. En numerosas oportunidades se ha observado que cientos de ballenas nadan juntas hacia la costa y quedan varadas en la playa sin poder moverse. Algunos sostienen que se debe a una pérdida de orientación posiblemente causada por la contaminación sonora de los océanos que interferiría con su capacidad de intercomunicación. En estos casos los equipos de rescate realizan enormes esfuerzos para regresarlas al interior del mar y salvar sus vidas. Se encuentran n ballenas varadas en una playa y se conocen los tiempos s1, s2, . . . , sn que cada ballena sea capaz de sobrevivir hasta que la asista un equipo de rescate. Dar un algoritmo voraz que determine el orden en que deben ser rescatadas para salvar el mayor número posible de ellas, asumiendo que llevar una ballena mar adentro toma tiempo constante t, que hay un único equipo de rescate y que una ballena no muere mientras está siendo regresada mar adentro.

    - Criterio de búsqueda : decido salvar a la que tenga menor tiempo de vida.
    - Tenemos N ballenas, cada una con un tiempo de vida diferente.
    - El algoritmo debe optimizar la cantidad de ballenas salvadas.
    - Rescatar a la ballena lleva un tiempo “t”. (es el mismo para todas)
    - Tipo de dato elegido : tupla para cada ballena con un campo id, y un tiempo de vida.

```
type Wale = tuple
        id : Nat
        life_time : Nat
end tuple
```

```
fun wales_rescuer (w : Set of Wale, t : Nat) ret res : List of Wale
        var w_aux : Set of Wale
        var time : Nat
        var actual_wale : Wale
        w_aux := copy_set(w)
        time := 0
        res := empty_list()
        while !is_empty_Set(w_aux) do
                actual_wale := rescue_w(w_aux)
                addr(res, ballena)
                elim_set(w_aux, actual_wale)
                time := time + t
                delete_dead_wales(w_aux, time)
        od
        destroy(w_aux)
end fun

fun rescue_w (w: Set of Wale) ret res : Wale
        var min_time_left := Nat
        var w_aux : Set of Wale
        var actual_wale : Wale
        w_aux := copy_set(w)
        min_time_left := inf
        while !is_empty_set(w_aux) do
                actual_wale := get(w_aux)
                if actual_wale.life_time < min_time_left then
                        min_time_left := actual_wale.life_time
                        res := actual_wale
                fi
                elim_set(w_aux, actual_wale)
        od
        destroy_set(w_aux)
end fun

proc delete_dead_wales (in/out w : Set of Wale, t : Nat)
        var w_aux : Set of Wale
        var pos_dead_wale : Wale
        w_aux := copy_set(w)
        while !is_empty_set(w_aux) do
                pos_dead_wale := get(w_aux)
                if pos_dead_wale.life_time < t then
                        elim_set(w, pos_dead_wale)
                fi
                elim_set(w_aux, pos_dead_wale)
        od
        destroy_set(w_aux)
end proc
```

5. Sos el flamante dueño de un teléfono satelital, y se lo ofreces a tus n amigos para que lo lleven con ellos cuando salgan de vacaciones el próximo verano. Lamentablemente cada uno de ellos irá a un lugar diferente y en algunos casos, los períodos de viaje se superponen. Por lo tanto es imposible prestarle el teléfono a todos, pero quisieras prestarlo al mayor número de amigos posible. Suponiendo que conoces los días de partida y regreso (pi y ri respectivamente) de cada uno de tus amigos, ¿cuál es el criterio para determinar, en un momento dado, a quien conviene prestarle el equipo? Tener en cuenta que cuando alguien lo devuelve, recién a partir del día siguiente puede usarlo otro. Escribir un algoritmo voraz que solucione el problema.
    - Criterio de selección: se lo presto a quienes más pronto viajen y que menor tiempo va a estar de viaje.
    - Tengo “n” amigos.
    - Pi (dia de partida) - Ri (dia de regreso).
    - Si lo devuelve el dia “i”, se puede usar recien el dia “i + 1”
    - Tipos de datos : una tupla Friend, un Set of Friend y como resultado un List of Friend

```
type Friend = tuple
        name : String
        start_day_h : Nat
        end_day_h : Nat
end tuple

fun solidarity_phone (friends : Set of Friend) ret res : List of Friend
        var aux_friends : Set of Friend
        var actual_friend : Friend
        var last_day : Nat              /*dia que va a estar disponible para alquilar de nuevo*/
        aux_friends : copy_Set(friends)
        last_day := 0
        while !is_empty_set do
                actual_friend := who_lend_phone(aux_friends, last_day)
                last_day := actual_friend.end_day_h          /*actualizamos ese dia*/
                addr(res, actual_friend)
                elim _set(aux_friends)
        od
        destroy_set(aux_friends)
end fun

fun who_lend_phone (friends : Set of Friend, last_day : Nat) ret res : Friend
        var aux_friends : Set of Friends
        var pos_lend_friend : Friend
        var min_start_day : Int                         /*minimo dia en el que alguien viaja*/
        var is_available : Bool                         /*¿Está disponible para alquilar?*/
        aux_friends := copy_set(friends)
        min_start_day := inf
        while !is_empty_set(aux_friends) do
                pos_lend_friend := get(aux_friends)
```

```
            is_available := last_day < pos_lend_friend.start_h
            elim_set(aux_friends, pos_lend_friend)
            if pos_lend_friend.start_day_h < min_start_day && is_available then
                min_start_day := pos_lend_friend.start_day_h
                res := pos_lend_friend
            fi
        od
        destroy(aux_friends)
end fun
```

6. Para obtener las mejores facturas y medialunas, es fundamental abrir el horno el menor número de veces posible. Por supuesto que no siempre es fácil ya que no hay que sacar nada del horno demasiado temprano, porque queda cruda la masa, ni demasiado tarde, porque se quema. En el horno se encuentran n piezas de panaderia (facturas, medialunas, etc). Cada pieza i que se encuentra en el horno tiene un tiempo mínimo necesario de cocción ti y un tiempo máximo admisible de cocción Ti . Si se la extrae del horno antes de ti quedaría cruda y si se la extrae después de Ti se quemará.

   Asumiendo que abrir el horno y extraer piezas de el no insume tiempo, y que ti ≤ Ti para todo i ∈ {1, . . . , n}, ¿qué criterio utilizará un algoritmo voraz para extraer todas las piezas del horno en perfecto estado (ni crudas ni quemadas), abriendo el horno el menor número de veces posible? Implementarlo.

   - Optimizar apertura del horno
   - N piezas de panadería
   - Cada pieza “i” tiene un tiempo de cocción mínimo “ti”
   - Cada pieza “i” tiene un tiempo de cocción máximo “Ti”
   - Criterio de selección : seleccionar las piezas más cercanas a su máximo tiempo admisible de cocción, sin que salgan quemadas.
   - Tipos de datos, para cada pieza usamos una tupla, un conjunto de piezas y para retornar una lista de piezas.

```
type Bakery_piece = tuple
            id : Nat
            min_time_c : Nat
            max_time_c : Nat
end tuple
fun order_to_cook_bakery_pieces (bakery_pieces:Set of Bakery_piece) ret res : List of Bakery_piece
        var bakery_pieces_aux : Set of Bakery_piece
        var actual_bakery_piece : Bakery_piece
        var actual_time : Nat
        bakery_pieces_aux := copy_set(bakery_pieces)
        while !is_empty_set(bakery_pieces_aux) do
            actual_bakery_piece := get_out_of_oven(bakery_pieces_aux, actual_time)
            actual_time := actual_bakery_piece.max_time_c
            addr(res, actual_bakery_piece)
```

```
            elim_set(actual_bakery_piece)
        od
        destroy_set(bakery_pieces_aux)
end fun

fun get_out_of_oven (bakery_pieces : Set of Bakery_piece, actual_time) ret res : Bakery_piece
```

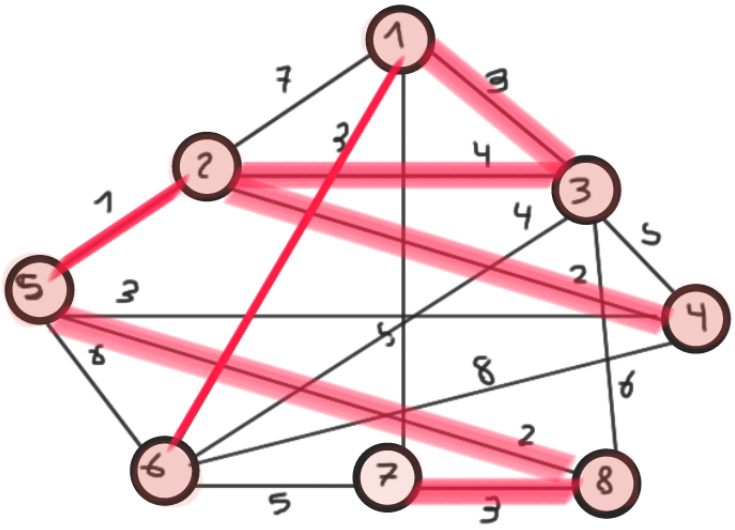
7. Usted vive en la montaña, es invierno, y hace mucho frío. Son las 10 de la noche. Tiene una voraz estufa a leña y troncos de distintas clases de madera. Todos los troncos son del mismo tamaño y en la estufa entra solo uno por vez. Cada tronco i es capaz de irradiar una temperatura ki mientras se quema, y dura una cantidad ti de minutos encendido dentro de la estufa. Se requiere encontrar el orden en que se utilizará la menor cantidad posible de troncos a quemar entre las 22 y las 12 hs del dia siguiente, asegurando que entre las 22 y las 6 la estufa irradie constantemente una temperatura no menor a K1; y entre las 6 y las 12 am, una temperatura no menor a K2.
    - Criterio de selección : el que mayor tiempo dure y que su temperatura sea mayor o igual a K1/K2.
    - Tipos de datos : un conjunto de troncos y dos naturales para K1 y K2

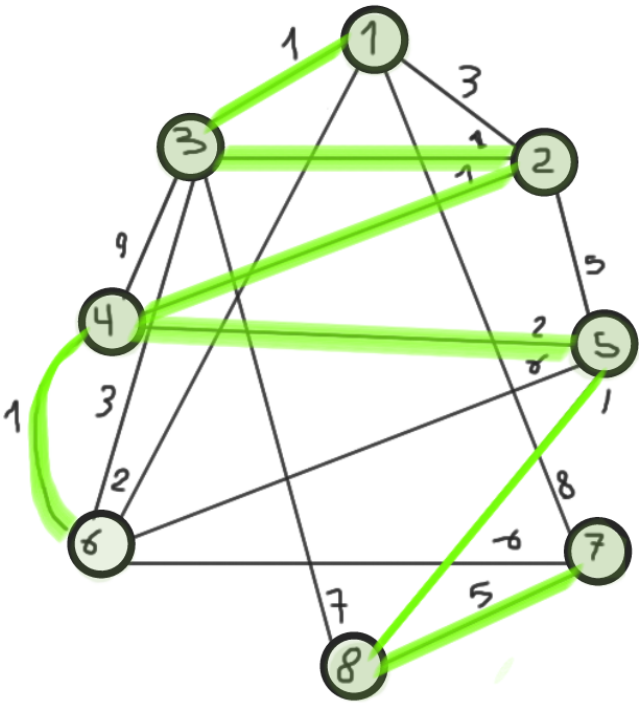
```
type Tronco = tuple
        id : Nat
        k : Nat
        t : Nat
end tuple

fun estufa (troncos : Set of Tronco, K1 : Nat, K2 : Nat) ret res : List of Tronco
        var copy_troncos : Set of Tronco
        var tronco_actual : Tronco
        var hora : Nat
        copy_troncos := copy_set(troncos, copy_troncos)
        hora := 0                   /*la hora hora desde cero hasta hasta 13*/
        while hora < 14 do
                if h < 8 then
                        tronco := seleccionar_mejor_tronco(copy_troncos, K1)
                else
                        tronco := seleccionar_mejor_tronco(copy_troncos, K2)
                fi
                addr(res, tronco)
                elim_set(copy_troncos, tronco)
                hora := hora + 1
        od
        destroy(copy_troncos)
end fun
```

```
fun seleccionar_mejor_tronco (troncos : Set of Tronco, K : nat) ret res : Tronco
        var copy_troncos : Set of Tronco
        var tronco_actual : Tronco
        var max_tiempo := -inf
        copy_troncos := copy_set(troncos, copy_troncos)
        while !is_empty_set(copy_troncos) do
                tronco_actual := get(copy_troncos)
                elim_set(copy_troncos, tronco_actual)
                if tronco_actual.k >= K && tronco_actual.t > max_tiempo then
                        res := tronco_Actual
                        max_tiempo := tronco_actual.t
                fi
        od
        destroy_set(copy_troncos)
end fun

                        i
```

Práctico 3 - Parte 2: Algoritmos voraces sobre grafos

1. Ejecutar paso a paso, graficando las soluciones parciales, el algoritmo de Prim que computa el árbol generador mínimo sobre los grafos con nodos {1, 2, . . . , 8} y costos dados por una función w:
   a. w((1, 2)) = 7 w((1, 6)) = 3 w((1, 7)) = 5 w((1, 3)) = 3 w((2, 3)) = 4 w((2, 4)) = 2 w((2, 5)) = 1 w((3, 4)) = 5 w((3, 6)) = 4 w((3, 8)) = 6 w((4, 6)) = 8 w((5, 4)) = 3 w((5, 6)) = 6 w((6, 7)) = 5 w((8, 5)) = 2 w((8, 7)) = 3
   b. w((1, 2)) = 3 w((1, 6)) = 2 w((1, 7)) = 8 w((1, 3)) = 1 w((2, 3)) = 1 w((2, 4)) = 1 w((2, 5)) = 5 w((3, 4)) = 9 w((3, 6)) = 3 w((3, 8)) = 7 w((4, 6)) = 1 w((5, 4)) = 2 w((5, 6)) = 6 w((6, 7)) = 6 w((8, 5)) = 1 w((8, 7)) = 5





2. Ejecutar paso a paso el algoritmo de Dijkstra que computa el camino de costo mínimo entre un nodo dado y los restantes nodos de un grafo, sobre los dos grafos especificados en el ejercicio anterior. Considerar 1 como el nodo inicial. Explicitar en cada paso el conjunto de nodos para los cuales ya se ha computado el costo mínimo y el arreglo con tales costos.

- Subir corrida a mano.

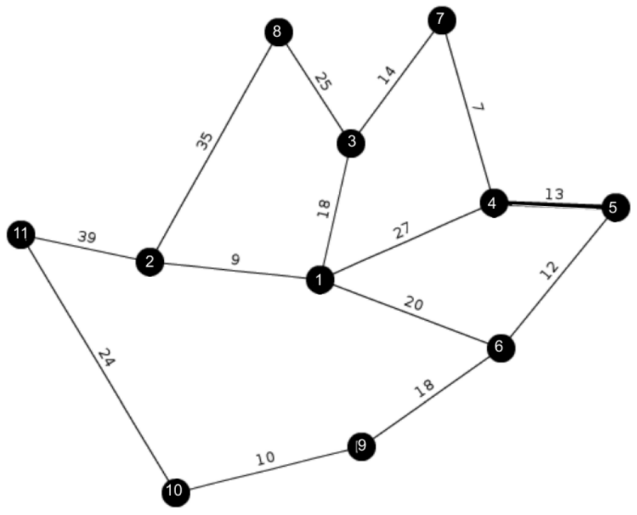
3. Usted quiere irse de vacaciones y debe elegir una ciudad entre K posibles que le interesan. Como no dispone de mucho dinero, desea que el viaje de ida hacia la ciudad pueda realizarse con a lo sumo L litros de nafta.

a. De un algoritmo que, dado un grafo representado por una matriz E : array[1..n,1..n] of Nat, donde el elemento E[i,j] indica el costo en litros de nafta necesario para ir desde la ciudad i hasta la ciudad j; un conjunto C de vértices entre 1 y n, representando las ciudades que quieren visitar; un vértice v, representando la ciudad de origen del viaje; y un natural L, indicando la cantidad de litros de nafta total que puede gastar; devuelva un conjunto D de aquellos vértices de C que puede visitar con los L litros.
b. Ejecuta el algoritmo implementado en el inciso anterior para el grafo descrito en el siguiente gráfico, con vértices 1,2,...11, tomando C = {11,5,10,7,8} como las ciudades de interés, disponiendo de L = 40 litros de nafta. ¿Cuáles son los posibles destinos de acuerdo a su presupuesto?
Ayuda: Utilice el algoritmo de Dijkstra

```
fun min_oil_liters (E : array[1..n, 1..n] of Nat, vertex_set : Set of Nat, v : Nat, l : Nat) ret res : Set of Nat
    var aux : Set of Nat
    var oil_cost : array[1..n] of Nat
    var actual_vertex_cost : Nat
    aux := copy_set(vertex_set)
    res := empty_set()
    oil_cost := Dijkstra(E, v)              --Obtengo los costos desde la ciudad inicial--
    while !is_empty_set(aux) do
        actual_vertex_cost := get(aux)
        if oil_cost[actual_vertex_cost] < l then
            add(res, actual_vertex_cost)
        fi
        elim(aux, actual_vertex_cost)
    od
    destroy(aux)
end fun
```

Ejecución del ejemplo :

- E = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
- l = 40
- C = {11, 5, 10, 7, 8}



Luego de ejecutar el algoritmo, obtenemos el arreglo con los costos a los vértices, partiendo desde v = 1 teniendo en cuenta que las ciudades de interés son C = {5, 7, 8, 10, 11}.

- oil_cost = [32, 32, 43, 48, 48]
- res = {5} o {7}

- cambio(i, j) = | j < 0
  |-inf j>0 && i<0
  |0 j=0 && i<0
  |max(cambio(i-1, j - di),cambio(i-1, j)) j>0 && i>0

Práctico 3 - Parte 3: Backtracking

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

- cambio(i, j) = “menor número de monedas necesarias para pagar exactamente el monto j con denominaciones d1, d2, . . . , di .”
- cambio(i, j) | 0 j = 0
  | inf j > 0 ^ i = 0
  | cambio(i - 1, j) di > j > 0 ^ i >0
  | min(cambio(i - 1, j) , di + cambio(i, j - di)) di <= j ^ j > 0 ^ i > 0

2. En un extraño país las denominaciones de la moneda son 15, 23 y 29, un turista quiere comprar un recuerdo pero también quiere conservar el mayor número de monedas posibles. Los recuerdos cuestan 68, 74, 75, 83, 88 y 89. Asumiendo que tiene suficientes monedas para comprar cualquiera de ellas, ¿cuál de ellas elegiría? ¿Qué monedas utilizará para pagarlo? Justificar claramente y mencionar el método utilizado.
   - Podemos utilizar la técnica de backtracking para evaluar todas las posibles combinaciones y quedarnos con el recuerdo que consuma una menor cantidad de monedas. Es decir, seleccionar las monedas que sumen el monto de un recuerdo, y luego quedarnos con el recuerdo que consuma una menor cantidad de las mismas.
   - recuerdo_ideal(j) será mi función, siendo “mj” los montos de los recuerdos :

La función quedaría :

- recuerdo_ideal(i, j) | 0 j = 0
  | min(cambio(i, dj) + recuerdo_ideal(i, j -1)) j > 0
- siendo recuerdo_ideal(3, 6) la llamada principal.

3. Una panadería recibe n pedidos por importes m1, . . . , mn, pero solo queda en depósito una cantidad H de harina en buen estado. Sabiendo que los pedidos requieren una cantidad h1, . . . , hn de harina (respectivamente), determinar el máximo importe que es posible obtener con la harina disponible.

   - n pedidos cada uno con un importe determinado.
   - tenemos una cantidad H de harina disponible.
   - necesitamos obtener el máximo de ganancia con la harina que tenemos.
   - bakery_orders(i, k) siendo di los pedidos y k la harina.
   - siendo bakery_orders(n, H) la llamada principal.

bakery_orders(i, k)	| -inf	i > 0, k = 0
	| 0	i = 0
	| bakery_orders(i - 1, k)	hi > k
	| max(bakery_orders(i - 1, k), mi + bakery_orders(i - 1, k - hi))	c.c

4. Usted se encuentra en un globo aerostático sobrevolando el océano cuando descubre que empieza a perder altura porque la lona está levemente dañada. Tiene consigo n objetos cuyos pesos p1, . . . , pn y valores v1, . . . , vn conoce. Si se desprende de al menos P kilogramos logrará recuperar altura y llegar a tierra firme, y afortunadamente la suma de los pesos de los objetos supera holgadamente P. ¿Cuál es el menor valor total de los objetos que necesita arrojar para llegar sano y salvo a la costa?
    - save_balloon(j, k) = “mínimo valor total que necesito arrojar para llegar sano y salvo”.
    - “j” es el objeto que necesito soltar, “j” va desde 1,...,n y “k el peso que necesito igualar.
    - la llamada principal sera save_the_balloon(n, P).

save_the_balloon(j, k)	| 0	k = 0
	| inf	k > 0 ^ j = 0
	| min(save_balloon(j - 1, k), vj + save_balloon(j - 1, k - pj))	k > 0 ^ j > 0

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

    - rent_phone(i) = “máximo valor alcanzable alquilando el teléfono a partir del dia i hasta el ultimo dia”.
    - la llamada principal será rent_phone(0)

rent_phone(i)	| 0	pi < i
	| max(rent_phone(i + 1), mi * (ri - pi) + rent_phone(ri +1))	pi >= i

6. Un artesano utiliza materia prima de dos tipos: A y B. Dispone de una cantidad MA y MB de cada una de ellas. Tiene a su vez pedidos de fabricar n productos p1, . . . , pn (uno de cada uno). Cada uno de ellos tiene un valor de venta v1, . . . , vn y requiere para su elaboración cantidades a1, . . . , an de materia prima de tipo A y b1, . . . , bn de materia prima de tipo B. ¿Cuál es el mayor valor alcanzable con las cantidades de materia prima disponible?
    - craft_man(k, MA, MB) = “mayor valor alcanzable con las cantidades de materia prima disponible”.
    - siendo “k” el número de pedidos.
    - craft_man(n, MA, MB) será la llamada principal.

craft_man(k, a, b)	| 0	k = 0
	| -inf	k > 0 ^ (a < 0 || b< 0)
	| craft_man(k - 1, a, b)	t > 0 ^ (ak > a || bk > a)
	| max (craft_man(k - 1, a, b), vk + craft_man(k - 1, a - ak, b - bk))	c.c

7. Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene n estaciones de trabajo, S1,1, . . . , S1,n para la primera y S2,1, . . . , S2,n para la segunda. Dos estaciones S1,i y S2,i (para i = 1, . . . , n), hacen el mismo trabajo, pero lo hacen con costos a1,i y a2,i respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por n estaciones de trabajo Si1,1, Si2,2 . . . , Sin,n no necesariamente todas de la misma línea de montaje (ik = 1, 2). Si el automóvil está en la estación Si,j , transferirlo a la otra línea de montaje (es decir continuar en Si 0 ,j + 1 con i 0 6= i) cuesta ti,j . Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.
    - Tenemos n estaciones por cada línea.
    - Cada estación tiene un costo distinto.
    - Mover el coche de una línea a otra produce un costo ti, j.
    - Para fabricar un auto no es necesario pasar por las dos líneas.
    - Debemos encontrar un mínimo.

Práctico 3 - Parte 4: Programación dinámica

1. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de la siguiente definición recursiva (backtracking):
cambio(i, j) = “n” denominaciones, monto “k”.

- La tabla que daremos es una matriz, de (n) filas, y (k) columnas.
- Luego, la función quedaría de la forma:

```
fun cambio_dinamica (c_denominaciones : array[1..N] of Nat , k : Nat) ret res :  Nat
        var tab : array[1..N, 1..K] of Nat
        var min_d := ∞
        for i := 1 to N do
                tab[i, 1] := 0
        od
        for j := 1 to K do
                tab[1, j] := ∞
        od
        for i := 2 to N do
                for j : 2 to K do
                        for q := 1 to (j / c_denominaciones[i]) do
                                min_d := min_d min (q + tab[i - 1, j - q * c_denominaciones[i]])
                        od
                        tab[i, j] := min_d
                od
        od
        res := tab[N, K]
end fun
```

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.

- La podemos llenar de derecha a izquierda, o viceversa. Pero si o si debe llenarse de arriba a abajo puesto que la llamada recursiva (q + cambio(i − 1, j − q ∗ di)) se fija en la fila “i - 1”, osea en la anterior.

3. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de cada una de las siguientes definiciones recursivas (backtracking):

a. cambio(i, j) =

0	j = 0
∞	j > 0 ^ i = n
cambio(i + 1, j)	di > j > 0 ^ i < n
min(cambio(i + 1, j), 1 + cambio(i, j − di))	j ≥ di > 0 ∧ i < n

- Notemos que, las llamadas recursivas se hacen siempre con i + 1, por lo tanto la tabla se llenará de arriba hacia abajo, sin importar el orden de izquierda a derecha o viceversa.

```
fun cambio_V2 (c_den : array[1..N] of Nat , k : Nat) ret res :  Nat
        var tab : array[0..N,0..K] of Nat
        for i := 0 to N do
                tab[i,0] := 0
        od
        for j := 1 to K do
                tab[N, j] := ∞
        od
        for i := N - 1 to 1 do
                for j := 2 to K do
                        if j > 0 && c_den[i] && j >= c_den[i] && i < N then
                                tab[i, j] := tab[i + 1, j] min (1 + tab[i, j - c_den[i])
                        fi
                        if c_den[i] > 0 && j > 0 && c_den[i] > j && i < N then
                                tab[i, j] := tab[i + 1, j]
                        fi
                od
        od
        res := tab[0, k]
end fun
```

4. Para cada una de las soluciones que propuso a los ejercicios del 3 al 9 del práctico de backtracking, dar una definición alternativa que utilice la técnica de programación dinámica. En los casos de los ejercicios 3, 5 y 7 modificar luego el algoritmo para que no solo calcule el valor óptimo sino que devuelva la solución que tiene dicho valor (por ejemplo, en el caso del ejercicio 3, cuales serian los pedidos que debería atenderse para alcanzar el máximo valor).

```
bakery_orders(i, k)   | 0                                                    i = 0
                      | -inf                                                 i > 0, k = 0
                      | bakery_orders(i - 1, k)                             hi > k
                      | max(bakery_orders(i - 1, k), mi + bakery_orders(i - 1, k - hi))   c.c

save_the_balloon(j, k)      | 0                                              k = 0
                            | inf                                            k > 0 ^ j = 0
                            | save_the_balloon(j - 1, k)                    k = P ^ j > 0
                            | min(save_balloon(j - 1, k), vj + save_balloon(j - 1, k - pk))  k > 0 ^ j > 0

rent_phone(i)         | 0                                                    pi < i
                      | max(rent_phone(i + 1), mi * (ri - pi) + rent_phone(ri +1))   pi >= i


craft_man(k, a, b)    | 0                                                    k = 0
                      | -inf                                                 k > 0 ^ (a < 0 || b< 0)
                      | craft_man(k - 1, a, b)                              t > 0 ^ (ak > a || bk > a)
                      | max (craft_man(k - 1, a, b), vk + craft_man(k - 1, a - ak, b - bk ))   c.c
```

```
fun bakery_orders(orders:array[1..N] of Nat, flour: Nat) ret max_profit : Nat
        var tab: array[0..N, 0..H] of Nat
        var orders_res : array[1..K]
        for j := 0 to H do
                tab[0, j] := 0
        od
        for i := 1 to N do
                tab[i, 0] := -∞
        od
        for i := 1 to N do
                for j := 1 to H do
                        if j > flour then
                                tab[i, j] := tab[i - 1, j]
                        else
                                tab[i, j] := tab[i - 1, j] max (orders[i].m + tab[i - 1, j - orders[i].h])
                        fi
                od
        od
        max_profit := tab[N, H]
end fun

fun save_the_balloon (objects:array[1..N] of Nat, weight: Nat) ret min_loss : Nat
        var tab: array[0..N, 0..P] of Nat
        for k := 0 to N  do
                tab[i, 0] := 0
        od
        for j := 1 to P  do
                tab[1, j] := ∞
        od
        for j := 1 to N do
                tab[j, k] :=tab[j - 1, k]
        od
        for j := 1 to N do
                for k := 1 to P do
                        if k > 0 && j > 0 then
                                tab[j, k] := tab[j - 1, k] min objects[i] + save_balloon[j - 1, k - objects[i]]
                        fi
                od
        od
        min_loss := tab[N,P]
end fun
```

REPASO PARA EL 2DO PARCIAL

1. Dada la siguiente definición recursiva, diseñar un algoritmo en programación dinámica.

gunthonacci (i, j) =	| 1	i = 0 ^ j = 0
	| 1	i = 0 ^ j = 1
	| 1	i = 1 ^ j = 0
	| gunthonacci(i, j - 2) + gunthonacci(i, j - 1)	i = 0 ^ j > 1
	| gunthonacci(i - 2, j) + gunthonacci(i - 1, j)	i > 1 ^ j = 0
	| gunthonacci(i , j - 1) + gunthonacci(i - 1, j)	i > 0 ^ j > 0

Analicemos un poco esta definición recursiva, en donde gunthonacci(n, n) es la llamada principal.

- Tendremos una matriz cuadrada, de (n + 1) filas y columnas.
- Notemos que para ambos índices siempre se completan en razón a sus anteriores, por lo que la deberemos de llenar de arriba hacia abajo y desde izquierda a derecha.
- Luego la función queda de la forma:

```
fun gunthonacci(n : Nat) ret res : Nat
        var tab_g : array[0..n,0..n] of Nat
        array[0, 0] := 1
        array[0, 1] := 1
        array[1, 0] := 1
        for j = 2 to n do
                tabla[0, j] := tabla[0, j - 2] + tabla[0, j - 1]
        od

        for i = 2 to n do
                tabla[i, 0] := tabla[i - 2, 0] + tabla[i - 1, 0]
        od

        for i = 1 to n do
                for j = 1 to n do
                        tabla[i, j] :=  tabla[i, j - 1] + tabla[i - 1, j]
                od
        od
        res := tab_g[n, n]
end fun
```

2. Sea T un árbol (no necesariamente binario) y supongamos que deseamos encontrar la hoja que se encuentra más cerca de la raíz. ¿Cuáles son las distintas maneras de recorrer T? ¿Cuál de ellas elegirías para encontrar esa hoja y porque?

- Bien, comencemos enumerando las formas de recorrer un árbol, que principalmente se dividen en dos ramas. La primer rama es DFS, llamada así porque procesa los vértices en “profundidad”, luego en esta rama encontramos 3 variantes (cabe resaltar que existen permutaciones de las mismas) :
    - Pre-order : procesa primero la raíz, y luego todo subárbol izquierdo, finalmente concluye procesando el sub árbol derecho.
    - In-order : procesa primero el subárbol izquierdo, luego la raíz para concluir procesando el subárbol derecho.
    - Pos-order : procesa primero el subárbol izquierdo y luego el subárbol derecho para concluir procesando la raíz.

- La segunda rama se llama BFS, llamada así porque recorre el árbol a lo ancho, es decir, por niveles. Esta última forma de procesar los vértices es más compleja.
- Finalmente, suponiendo que queremos conocer la hoja más cercana a la raíz, elegiría un algoritmo BFS, ya que recorreremos todo el segundo nivel (hojas siguientes a la raíz) procesando esos vértices. Esto es mucho más simple que utilizar alguna variante de DFS.

3. Ejercicio de algoritmos voraces. Un colectivero puede transportar solamente un pasajero, su recorrido va desde la parada 1 hasta la n pasando por las paradas 2,...,n - 1. Hay m pasajeros esperando, para cada pasajero i sabemos en qué parada se quiere subir y en cual parada se quiere bajar. La intención del colectivero es transportar la mayor cantidad de pasajeros en un solo viaje.
    a. Indicar de manera simple y concreta, cual es el criterio de selección voraz para construir la solución?
    b. Indicar que estructura de datos utilizaras para resolver este problema
    c. Explicar en palabras cómo resolverá el problema el algoritmo.
    d. Implementar el algoritmo en el lenguaje de la materia de manera precisa.

Las estructuras a utilizar serán una arreglo desde 1..n con las paradas y una lista con los pasajeros:

- pasajeros : Set of Pasajero

- El resultado estará dado por un natural, que tendrá el máximo de pasajeros que el colectivero puede transportar.
- El criterio de selección será elegir al pasajero que antes baje.
- La tupla “Pasajero” contendrá los campos “id” y las paradas donde subirá y donde bajará.

```
type Pasajero = tuple
        id : Nat
        s_i : Nat
        b_i : Nat
end tuple

fun colectivero (pasajeros : Set of Pasajero) ret max_pasajeros : Nat
        var pasajeros_copy : Set of Pasajero
        var pasajero_a : Pasajero
        var pasajeros_subidos : List of Pasajero
        pasajeros_subidos := empty_list()
        while !is_empty_set(pasajeros_copy) do
                pasajero_a := elegir_pasajero(pasajeros_copy)
                addr(pasajero_a, pasajeros_subidos)
                elim(pasajero_a, pasajeros_copy)
                remover_pasajeros(pasajeros_copy, pasajero_a)
        od
        max_pasajeros := length(pasajeros_subidos)
        destroy(pasajeros_subidos)
        destroy(pasajeros_copy)
end fun

fun elegir_pasajero(pasajeros : Set of Pasajero) ret res : Pasajero
        var pasajeros_copy : Set of Pasajero
        var pasajero_a : Pasajero
        var minima_bajada : Nat
        minima_bajada := infinito
        while !is_empty_set(pasajeros_copy) do
                pasajero_a := get(pasajeros_aux)
                set_elim(pasajero_a, pasajeros_copy)
                if pasajero_a.b_i < minima_bajada then
                        res := pasajero_a
                        minima_bajada := pasajero_a.b_i
                fi
        od
end fun

proc remover_pasajeros(in/out pasajeros : Set of Pasajero, in pasajero_a : Pasajero)
        var pasajero_b : Pasajero
        while !is_empty_set(pasajeros) do
                pasajero_b := get(pasajeros)
                if pasajero_b.s_i <=  pasajero_a.s_i then
                        set_elim(pasajero_b, pasajeros_copy)
                fi
        od
end proc
```

4. El presidente de tu país te acaba de elegir como asesor para tomar una serie de medidas que mejoren la situación económica. En el análisis preliminar se proponen n medidas, donde cada medida i perteneciente a 1,...,n producirá una mejora económica m. También se analizó para cada una el nivel de daño ecológico d. El puntaje de cada medida será dado en relación m/d. Se debe determinar el máximo puntaje obtenido eligiendo K medidas de manera que la suma total de daño ecológico no sea mayor a C.

- La función recursiva calculará un máximo, en este caso el máximo puntaje obtenible sin causar un daño mayor a C.
- La llamada principal será asesor(n, C,K)
- La función recursiva, será de la forma asesor(i,c,k), siendo i la medida que estoy considerando para la propuesta, c el daño ecológico actual y k las propuestas restantes.

asesor(i,c,k) =	| 0	c = 0
	| -inf	i = 0 ^ k > 0
	| asesor(i - 1, c,k)	di > C ^ i > 0 ^ k > 0
	| max(asesor(i - 1, c,k), mi/di + asesor(i - 1, c - di,k - 1)	c < C ^ i > 0 ^ k > 0

5. El profe de algoritmos 2 tiene n medias diferentes, con n numero par (digamos n = 2m). Hay una tabla P[1..n, 1..n] tal que P[i, j] es un número que indica cuán 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.

aparear (S) =	| 0	si S es vacío
	| max{i, j / i != j} P[i, j] + aparear(S - {i, j})	si S no es vacío

aparear (S) =	| 0	si S es vacío
	| max(aparear(S - {i, j}), P[i, j] + aparear(S - {i, j})	si S no es vacío ^ i != j

6. Vos y tus amigos, en total son m personas, se quieren ir de viaje por el fin de semana y tienen a disposición n autos, cada uno con capacidad para llevar una cantidad de personas c1, . . . , cn. ¿Cual es la menor cantidad de autos necesaria para que las m personas puedan viajar?

- n autos para llevar m personas.
- Cada uno con una capacidad distinta de pasajeros.
- Criterio de selección : elegir el auto con mayor capacidad.
- Estructuras de datos y funciones :

type Auto = **tuple**
    id : Nat
    capacidad : Nat
end tuple

```
fun viaje_voraz (autos : Set of Auto, m : Nat) ret cantidad_autos : Nat
        var copy_autos : Set of Auto
        var auto_a : Auto
        var personas_restantes : Nat
        var autos_elegidos : List of Auto
        copy_autos := copy_set(autos, copy_set)
        personas_restantes := 0
        autos_elegidos := empty_list()
        while !is_empty_set(copy_autos) && personas_restantes < m do
                auto_a := elegir_mejor_auto(copy_autos)
                addr(autos_elegidos, auto_a)
                elim_set(auto_a, copy_autos)
                personas_restantes := personas_restantes + auto_a.capacidad
        od
        cantidad_autos := length(autos_elegidos)
        destroy_set(copy_autos)
        destroy_list(autos_elegidos)
end fun

fun elegir_mejor_auto (autos : Set of Auto) ret res : Auto
        var copy_autos : Set of Auto
        var auto_a : Auto
        var max_capacidad : Nat
        max_capacidad := -inf
        copy_autos := copy_set(autos, copy_set)
        while !is_empty_set(copy_autos) do
                auto_a := get(copy_autos)
                elim_set(auto_a)
                if auto_a.capacidad > max_capacidad then
                        res := auto_a
                        max_capacidad := auto_a.capacidad
                fi
        od
        destroy_set(copy_autos)
end fun
```

7. Dados c1, c2, . . . , cn y d1, d2, . . . , dk, la siguiente definición recursiva de la función m, para $1 \leq i \leq n$ y $1 \leq j \leq k$, determinar un programa que utilice la técnica de programación dinámica para calcular el valor de m(1, 1).

- m(i, j) = | ci — si j = k
  | dj — si i = n ∧ j < k
  | m(i, j + 1) + m(i + 1, j) — si j < k ∧ i < n

Anotaciones importantes :

- El algoritmo llenará una tabla de n filas y k columnas.
- La tabla se llenará de abajo hacia arriba y de derecha a izquierda.

```
fun dinámica (c : array[1..n] of Nat, d : array[1..k] of Nat) ret res :  Nat
        var tab : Array[1..n, 1..k] of Nat
        for i := n downto 1 do
                tab[i, k] := c[i]
        od
        for j := k - 1 downto 1 do
                tab[n, j] := d[j]
        od
        for i := n - 1 downto 1 do
                for j := k - 1 downto 1 do
                        tab[i, j] := tab[i, j + 1] + tab[i + 1, j]
                od
        od
        res := tab[1, 1]
end fun
```

8. Es viernes a las 18 y usted tiene ganas de tomar limonada con sus amigos. Hay n bares cerca, donde cada bar i tiene un precio Pi de la pinta de limonada y un horario de happy hour Hi , medido en horas a partir de las 18 (por ejemplo, si el happy hour del bar i es hasta las 19, entonces Hi = 1). Usted toma una cantidad fija de 2 pintas por hora y no se considera el tiempo de moverse de un bar a otro. Escribir un algoritmo que obtenga el menor dinero posible que usted puede gastar para tomar limonada desde las 18 hasta las 02 am (es decir que usted tomará 16 pintas) eligiendo en cada hora el bar que más le convenga.
Anotaciones importantes :

- Existen n bares.
- Cada bar tiene un precio Pi y un horario Hi de happy hour.
- La persona toma dos pintas por horas.
- Criterio de selección : elegir tomar la pinta más barata, en un bar con HH.
- Supongamos que no existe bar en donde su precio inicial sea más barato que en uno con Happy Hour.
- Estructuras de datos y función voraz:

```
type Bar = tuple
        Id : Nat
        HH : Bool
        precio : Nat
        hora : Nat
end tuple

fun pintas(bares :  Set of Bar) ret min_gasto :  Nat
        var copy_bares : Set of Bar
        var bar_actual :  Bar
        var hora : Nat
        hora := 0
        min_gasto := 0
        copy_bares := copy_set(bares, copy_bares)
        while !is_empty_set(copy_bares) && hora < 8 do
                bar_actual := elegir_bar(copy_bares)
                min_gasto  := min_gasto + bar_actual.precio
                hora := hora + 1
                restringir_bares(copy_bares, hora)
        od
```

```
        destroy_set(copy_bares)
end fun

proc restringir_bares(in/out bares : Set of Bar, in hora: nat)
        var bar_actual :  Bar
        while !is_empty_set(bares) do
                bar_actual := get(bares)
                if hora > bar_actual.hora then
                        elim_set(bares, bar_actual)

                fi
end proc

fun elegir_bar(bares : Set of Bar) ret res : Bar
        var copy_bares : Set of Bar
        var bar_actual :  Bar
        var min_precio : Nat
        copy_bares := copy_set(bares, copy_bares)
        min_precio := inf
        while !is_empty_set(copy_bares) do
                bar_actual := get(bares)
                if bar_actual.precio < min_precio then
                        min_precio := bar_actual.precio
                        res := bar_actual
                fi
        od
end fun
```

9. Dados c1, c2, . . . , cn y la siguiente definición recursiva de la función m, para $1 \le i \le j \le n$, determinar un programa que utilice la técnica de programación dinámica para calcular el valor de m(n, n).

- $m(i, j) = \begin{cases} cj & \text{si } i = 1 \\ m(i-1, j-1) + m(i-1, j) & \text{si } j \ge i > 1 \end{cases}$

Ayuda: Antes de comenzar, hace un ejemplo para entender la definición recursiva. Puedes tomar por caso n = 4, c1 = 3, c2 = 1, c3 = 2, c4 = 5, y calcular m(4, 4).

Anotaciones importantes :

- La tabla se llena desde arriba hacia abajo, y de izquierda a derecha.
- La tabla será una matriz cuadrada de n filas y columnas.
- El resultado estará en la celda (n, n).

```
fun dinámica (c : array[1..n] of Nat) ret res :  Nat
        var tab : Array[1..n, 1..n] of Nat
        for j := 1 to n do
                tab[1, j] := c[ j ]
        od
        for i := 2 to n do
                for j := i to n do
                        tab[i, j] := tab[i - 1, j - 1] + tab[i - 1, j]
                od
        od
        res := tab[n, n]
        end fun
```

Ejercicio 2

El presidente de tu país te acaba de elegir como asesor para tomar una serie de medidas de producción que mejoren la situación económica. En el análisis preliminar se proponen n medidas, donde cada medida i ∈ {1, .., n} producirá una mejora económica de mi puntos, con mi > 0. También se analizó para cada una el nivel de daño ecológico di que producirá, donde di > 0. El puntaje que tendrá cada medida i está dado por la relación mi/di.

Se debe determinar cuál es el máximo puntaje obtenible eligiendo K medidas, con K < n, de manera tal que la suma total del daño ecológico no sea mayor a C.

i e {1,....,n}
Asesor(k, i, p) =

|0	k = 0 v i = 0
| asesor(k, i-1, p)	k>0 && i >0 && p < 0
| max(asesor(k-1, i-1, p -d1) + mi/di , asesor(k, i-1, p))	k > 0 ∧ i > 0 ∧p > 0

EJERCICIO 5. El profe de algoritmos 2 tiene n medias diferentes, con n numero par (digamos n = 2m). Hay una tabla P[1..n, 1..n] tal que P[i, j] es un número que indica cuán 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.

6. Vos y tus amigos, en total son m personas, se quieren ir de viaje por el fin de semana y tienen a disposición n autos, cada uno con capacidad para llevar una cantidad de personas c1, . . . , cn. ¿Cual es la menor cantidad de autos necesaria para que las m personas puedan viajar?

```
*criterio de selección: elegimos los autos con mayor capacidad
*type auto : tuple
                id: nat
                cant: nat
             end tuple

fun auto_voraz(s : Set of auto, m : Nat ) ret res : nat
        var s_copy : Set of auto
        var auto_a : auto
        var p_r : nat
        s_copy := copy_set(s, s_copy)
        p_r := 0
        res := 0
        while  (!is_empty_set(copy_set) && p_r < m) do
                auto_a := elegir(s_copy)
                elim_set(auto_a, s_copy)
```

```
                p_r := p_r + auto_a.cant
                res := res +1
        od
        destroy_set(s_copy)
end fun


fun elegir(s : Set of auto) ret res : auto
        var s_copy : Set of auto
        var max: nat
        var auto_a : auto
        s_copy := copy_set(s, s_copy)
        max := -inf
        while(!is_empty(s_copy)) do
                auto_a := get(s_copy)
                if (auto_a.cant > max) then
                        max := auto _a.cant
                        res := auto _a
                fi
                elim_set(auto_a, s_copy)
        od
end fun
```