0.1 EJERCICIOS DE ALGORITMOS

0.1.1 Ejercicio 1

Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

(a) ¿Que hace?

(b) ¿Cómo lo hace?

(c) El orden del algoritmo, analizando los distintos casos posibles.

(d) Proponer nombres mas adecuados para los identificadores (de variables, funciones y procedimientos).

```
proc p(in/out a: array[1..n] of nat)
    var d: nat
    for i := 1 to n do
        d := i
        for j:= i+1 to n do
            if a[j] < a[d] then
                d := j
            fi
        od
        swap(a,i,d)
    od
end proc
```

```
fun f(a: array[1..n] of nat) ret b: array[1..n] of nat
    var d: nat
    for i := 1 to n do
        b[i] := i
    od
    for i := 1 to n do
        d := i
        for j:= i+1 to n do
            if a[b[j]] < a[b[d]] then
                d := j
            fi
        od
        swap(b,i,d)
    od
end fun
```

0.1.1.0 Punto a

- El procedimiento **p** recibe un arreglo de números naturales y lo ordena de menor a mayor.
- EL procedimiento **f** recibe un arreglo de números naturales y devuelve un arreglo con los índices de los elementos del arreglo original ordenados de menor a mayor.

0.1.1.0 Punto b

- Recorre el arreglo de izquierda a derecha, en cada iteración busca el índice del menor elemento del arreglo que se encuentra a la derecha del índice actual y lo intercambia con el elemento en la posición actual.
- Primero crea un arreglo con los índices de los elementos del arreglo original, luego recorre el arreglo de índices de izquierda a derecha, en cada iteración busca el índice del menor elemento del arreglo original que se encuentra a la derecha del índice actual y lo intercambia con el índice actual.

0.1.1.0 Punto c

- El procedimiento **p** tiene un orden de $O(n^2)$, ya que recorre el arreglo de tamaño n y en cada iteración recorre el arreglo de tamaño n.
- El procedimiento **f** tiene un orden de $O(n^2)$, ya que recorre el arreglo de tamaño n y en cada iteración recorre el arreglo de tamaño n.

0.1.1.0 Punto d

- **p sort_array**
- **f index_sorted_array**

0.1.2 Ejercicio 2

Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

(a) ¿Que hace?

(b) ¿Cómo lo hace?

(c) El orden del algoritmo.

```
proc p(in/out a: array[0..n] of nat)
    for i := 0 to n/2 do
        swap(a,i,n−1)
    od
end proc
```

```
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 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
```

0.1.2.0 Punto a

- El procedimiento **p** recibe un arreglo de números naturales y lo invierte.
- La función **f** recibe un arreglo de números naturales y devuelve verdadero si el arreglo esta ordenado de menor a mayor y falso en caso contrario.
- El procedimiento **q** recibe una matriz de números naturales cuadrada, e invierte cada fila de la matriz.

0.1.2.0 Punto b

- Recorre el arreglo de izquierda a derecha, en cada iteración intercambia el elemento en la posición actual con el elemento en la posición $n-1$.
- Recorre el arreglo de izquierda a derecha, en cada iteración compara el elemento en la posición actual con el elemento en la posición siguiente, si el elemento actual es menor o igual al siguiente, avanza al siguiente elemento, en caso contrario devuelve falso.
- Recorre la matriz de izquierda a derecha y de arriba a abajo, en cada iteración intercambia el elemento en la posición actual con el elemento simétrico respecto al eje vertical.

0.1.2.0 Punto c

- El procedimiento **p** tiene un orden de $O(n)$, ya que recorre el arreglo de tamaño n y en cada iteración realiza una cantidad constante de operaciones.
- La función **f** tiene un orden de $O(n)$, ya que recorre el arreglo de tamaño n y en cada iteración realiza una cantidad constante de operaciones.
- El procedimiento **q** tiene un orden de $O(n^2)$, ya que recorre la matriz de tamaño $n \times n$ y en cada iteración realiza una cantidad constante de operaciones.

0.1.3 Ejercicio 3

Calcula el orden de cada uno de los siguientes algoritmos:

```
proc P(in/out a: array[1..n] of nat)
    for i:= to n do
        isort(a)
        swap(a,1,n)
    od
end proc
```

donde isort es el algoritmo de ordenación por inserción.

```
proc Q(in/out a: array[1..n] of nat, in izq, der: nat)
    var med : nati
    if izq < der then
        ssort(a,izq,der)
        med := (izq + der) div 2
        Q(a,izq,med)
        Q(a,med+1,der)
    fi
end proc

proc main(in/out a:array[1..n] of nat)
    Q(a,1,n)
end proc
```

donde ssort(a,izq,der) ordena el arreglo a entre las posiciones izq y der utilizando el algoritmo de ordenación por selección.

0.1.3.0 Solución

- Insertion sort tiene una complejidad de $O(n^2)$, entonces el procedimiento p tiene un orden de n^3.
- El procedimiento tiene primero una complejidad de (n^2) al hacer la llamada al algoritmo de ordenación por selección, luego se divide el arreglo en dos partes y se llama recursivamente a la función Q con cada una de las partes, por lo que la complejidad total es de $O(n^2 \log n)$.

0.1.4 Ejercicio 4

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 de 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 final del arreglo.

0.1.4.0 Solución

```
proc p(in/out a: array[1..n] of nat)
    var d: nat
    for i := n downto 1 do
        d := i
        for j:= 1 to n-1 do
            if a[j] > a[d] then
                d := j
            fi
        od
        swap(a,i,d)
    od
end proc
```

0.1.5 Ejercicio 5

Se conoce con el nombre de *cocktail sort* a una variación del algoritmo de ordenación *selection sort* que consiste en obtener en cada pasada del arreglo no sólo el elemento mínimo del segmento correspondiente, sino el mínimo y el máximo, intercambiándolos luego por las posiciones primera y última del segmento, respectivamente. Por ejemplo si tenemos el arreglo [8,5,18,3,4] (con posiciones de 1 a 5), el algoritmo en su primera pasada encontraría que el elemento mínimo está en la posición 4 y el máximo en la posición 3, que al intercambiarlos el arreglo resultante sería [3,5,18,8,4]. Se pide:

(a) Explicá con palabras claras cómo implementarás el algoritmo.

(b) Implementá de manera precisa el *cocktail sort* en el lenguaje de la materia.

0.1.5.0 Punto a

Recorrer el arreglo de izquierda a derecha, en cada iteración buscar el mínimo y el máximo del segmento correspondiente y luego intercambiarlos con los elementos en las posiciones primera y última del segmento, respectivamente. Luego se llama recursivamente al algoritmo con el segmento que va desde la posición siguiente a la primera hasta la posición anterior a la última.

0.1.5.0 Punto b

```
proc cocktail_sort(a :array[1..n] of nat)
    var d: nat
    for i:= 1 to n do
        d := i
        k := i
        for j:= i to n do
            {- busca el minimo -}
            if a[j] < a[d] then
                d := j
            fi
            {- busca el maximo -}
            if a[j] > a[k] then
                k := j
            fi
        od
        {- posiciona -}
        swap(a,i,d)
        swap(a,n-i+1,k)
    od
end proc
```

0.2 EJERCICIOS DE DIVIDE Y VENCERÁS

0.2.1 Ejercicio 1

(a) Dado el siguiente algoritmo, planteá la recurrencia que indica la cantidad de asignaciones realizadas en función de la entrada n:

```
fun f(n: nat) ret m: nat
    if n ≤ 2 then
        m := n
    else
        m := 3*f(n/2) + n
    fi
end fun
```

(b) Resolvé la siguiente recurrencia: $t(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ 4t(n/2) + n^2 & \text{si } n > 2 \end{cases}$.

(c) La recurrencia que obtuviste en el item a) ¿es la misma que resolviste en el item b)?. En caso contrario, resolvela también.

0.2.1.0 Punto a

- Recibe una entrada del tamaño n.
- La operación a contar es la asignación de la variable m.

$$t(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ 3t(n/2) & \text{si } n > 2 \end{cases}$$

- $a = 1$,
- $b = 2$,
- $k = 0$.

Entonces $a = 1 = b^k = 2^0 \rightarrow t(n) = \log_2 n$

0.2.1.0 Punto b

$$t(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ 4t(n/2) + n^2 & \text{si } n > 2 \end{cases}$$

- $a = 4$,
- $b = 2$,
- $k = 2$.

Entonces $a = 4 = b^k = 2^2 = 4 \rightarrow t(n) = n^2 \log_2 n$.

0.3 EJERCICIOS DE TADS

0.3.1 Ejercicio 1

Completar la implementación de listas dada en el teórico usando punteros.

```
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-}
fun head(l: List of T) ret e : T
    e := l->elem
end fun
```

```
{–PRE: not is_empty–}
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
        do p–>next ≠ null
            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)
    p := l
    n := 0
    do p ≠ null
        p := p–>next
        n := n+1
    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
    do p–>next ≠ null
        p := p–>next
    od
    p–>next := l0
end proc

fun index (l : List of T, n : nat) ret e : T
    var p : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p–>next
    end for
    e := p–>elem
end fun
```

```
proc take(in/out l : List of T, in n : nat)
    var p,q : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p->next
    end for
    q := p->next
    p->next := null
    do q != null
        var r : pointer to (Node of T)
        r := q
        q := q->next
        free(r)
    do
end proc

proc drop(in/out l : List of T, in n : nat)
    var p : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p->next
    end for
    l := p
end proc

fun copy_list(l1 : List of T) ret l2 : List of T
    var p : pointer to (Node of T)
    var q : pointer to (Node of T)
    if l1 = null then
        l2 := null
    else
        alloc(q)
        q->elem := l1->elem
        q->next := null
        l2 := q
        p := l1->next
        do p != null
            alloc(q->next)
            q := q->next
            q->elem := p->elem
            q->next := null
            p := p->next
        end while
    end if
end fun

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

0.3.2 Ejercicio 2

Un **multiconjunto** M con elementos en S, es un subconjunto de S donde cada elemento tiene asociado un número natural que indica cuantas veces el elemento ocurre. En otras palabras, un multiconjunto puede pensarse como un conjunto en el cual los elementos pueden estar incluidos más de una vez, indicando cuantas veces lo hace.

(a) Especificar el TAD `Multiset of T` de multiconjuntos con elementos de tipo T incluyendo constructores para crear el multiconjunto vacío y otro para agregar un elemento a un multiconjunto. Y operaciones para saber si un multiconjunto es vacío o no, para saber si un elemento dado existe en el multiconjunto, para obtener cuántas veces un elemento dado ocurre en un multiconjunto, y para eliminar una ocurrencia de un elemento en un multiconjunto.

(b) Implementar el TAD `Multiset of T` especifiado en el punto anterior, utilizando como representación interna una lista donde cada elemento es un par consistente de un elemento de tipo T y un número natural que indica la cantidad de ocurrencias.

(c) Implementar una función que reciba un arreglo de N naturales y devuelva el multiconjunto que contiene todos los elementos pares que ocurren en el arreglo. Todo tipo de datos utilizado en la función debe utilizarse de manera abstracta.

0.3.2.0 Punto a

```
spec Multiset of T where

constructors
    fun empty_m() ret m : Multiset of T
    {–crea un multiconjunto vacio–}

    proc add_elem(in e: T, in/out m: Multiset of T)
    {–agrega un elemento al multiconjunto–}

destroy
    proc destroy_m(in/out m: Multiset of T)
    {–libera memoria en caso de que sea necesario–}

operations
    fun is_empty(m: Multiset of T) ret b : bool
    {–devuelve true si el multiconjunto es vacio–}

    fun exists(m: Multiset of T, e: T) ret b: bool
    {–devuelve true si el elemento existe en el multiconjunto–}

    fun hm_ocu(m: Multiset of T, e: T) ret k: nat
    {–devuelve la cantidad de ocurrencias de un elemento en el multiconjunto–}

    {–PRE: exists(m,e)–}
    proc del_ocu(in/out m: Multiset of T, e: T)
    {–elimina una ocurrencia de un elemento en un multiconjunto–}
```

0.3.2.0 Punto b

```
implement Multiset of T where

type par of T = tuple
                  elem: T
                  ocu: nat
                end tuple

type Multiset of T = List of par

fun empty_m() ret m: Multiset of T
    m := empty()
end fun

proc add_elem(in e: T, in/out m: Multiset of T)
    var p: par of T
    var n: Multiset of T
    p.elem := e
    p.ocu := 1
    if is_empty(m) then
        addl(p, m)
    else
        n := copy_list(m)
        do not is_empty(n) ->
            if (head(n).elem = e) then
                head(n).ocu := head(n)->ocu + 1
            fi
            tail(n)
        od
        addl(p, m)
    fi
end proc

proc destroy_m(in/out m: Multiset of T)
    destroy(m)
end proc

fun exits(m: Multiset of T, e: T) ret b: bool
    var n: Multiset of T
    var tmp: par of T
    b := false
    n := copy_list(m)
    do not is_empty(n) ->
        tmp := head(n)
        if (tmp.elem = e) then
            b := true
        fi
        tail(n)
    od
end fun
```

```
fun hm_ocu(m: Multiset of T, e: T) ret k: nat
    var n: Multiset of T
    var tmp: par of T
    k := 0
    n := copy_list(m)
    do not is_empty(n) ->
        tmp := head(n)
        if (tmp.elem = e) then
            k := k+1
        fi
        tail(n)
    od
end fun

fun del_ocu(in/out m: Multiset of T, e: T)
    var n,o: Multiset of T
    n := copy_list(m)
    o := copy_list(m)
    var index : nat
    var tmp: par of T
    index := 1
    do not is_empty(n) ->
        tmp := head(n)
        if(tmp.elem = e)
            destroy(n)
        fi
        else index := index +1
        tail(n)
    od
    m := concat(take(o,index-1), drop(o,index+1))
end fun
```

0.3.2.0 Punto c

```
fun mul_pares(a : array[1..N] of nat) ret m: Multiset of T
    {-pongo los pares en el multiconjunto-}
    m := empty_m()
    for i := 1 to N do
        if a[i] mod 2 = 0 then
            add_elem(a[i], m)
        fi
    od
end fun
```

0.3.3 Ejercicio 3

Implementá el TAD Pila utilizando la siguiente representación

```
implement Stack of T where

type Stack of T = List of T
```

(la especificación del TAD Stack es la siguiente)

```
spec Stack of T where

constructors
    fun empty_stack() ret s : Stack of T
    {—crea una pila vacia.—}

    proc push (in e : T,in/outs : Stack of T)
    {—agrega el elemento e al tope de la pilas. —}

operations
    fun is_empty_stack(s : Stack of T) ret b : Bool
    {—Devuelve True si la pila es vacia—}

    fun top(s : Stack of T) ret e : T
    {—Devuelve el elemento que se encuentra en el tope des. —}

    {—PRE: not is_empty_stack(s) —}
    proc pop (in/out s : Stack of T)
    {—Elimina el elemento que se encuentra en el tope des. —}

    {—PRE: not is_empty_stack(s) —}
    fun copy_stack (s1 : Stack of T) ret s2 : Stack of T
    {— copia el contenido de la pila s1 en la nueva pila s2 —}

destroy
    proc destroy_stack (in/out s: Stack of T)
    {— elimina la memoria usada por la pila s en caso de ser necesario —}

end spec
```

0.3.3.0 Solución

```
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/outs : Stack of T)
    addl(e,s)
end proc

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 proc

fun copy_stack(s1: Stack of T) ret s2 : Stack of T
    s2 := copy_list(s1)
end fun

proc destroy_stack(in/out s: Stack of T)
    destroy(s)
end proc

end implement
```

0.3.4 Ejercicio 4

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)
```

0.3.4.0 Solución

```
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
    if s = null then
        s := p
    else
        p->next := s
        s := p
    fi
end proc

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

proc pop (in/out s : Stack of T)
    var p: pointer to (Node of T)
    p := s
    s := s->next
    free(p)
end proc

proc copy_stack(s1 : Stack of T) ret s2 : Stack of T
    var s10: pointer to (Node of T)
    var s20: pointer to (Node of T)
    var t : Node of T
    if is_empty_stack(s1) then
        s2 := empty_stack()
    else
        s10 := s1
        s2->elem := s1->elem
        s2->next := null
        s20 := s2

    while s10 != null do
        alloc(t)
        t->elem := s10->elem
        t->next := null
        s10 := s10->next
        s20 := s20->next
    fi
end proc
```

0.3.5 Ejercicio 5

Implementá el TAD Cola utilizando un arreglo, pero asegurando que todas las operaciones estén implementadas en orden constante.

0.3.5.0 Solución

```
implement Queue of T where

type Queue of T = tuple
                    elems : array [0 .. N-1] of T
                    size : nat
                    start : nat
                  end tuple

fun empty_queue() ret q : Queue of T
    q.size := 0
    q.start := 0
end fun

proc enqueue (in/out q : Queue of T,ine : T)
    var pos: nat
    pos := (q.start + q.size) {-N-}
    q.elems[pos] := e
    q.size := q.size + 1
end proc

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

fun first(q : Queue of T) ret e : T
    e := q.elems[0]
end fun

proc dequeue (in/out q : Queue of T)
    var start_pos: nat
    start_pos := (q.start + 1) {-N-}
    q.start := start_pos
    q.size := q.size - 1
end proc

proc destroy_queue (in/out q: Queue of T)
    skip
end proc

end implement
```

0.3.6 **Ejercicio 6**