EXAMEN FINAL 9 DE DICIEMBRE DE 2020

EJERCICIO 1

1. (Backtracking) No es posible correr una carrera de 800 vueltas sin reemplazar cada tanto las cubiertas (las ruedas) del auto. Como los mecánicos trabajan en equipo, cuando se cambian las cubiertas se reemplazan simultáneamente las cuatro. Reemplazar el set de cuatro cubiertas insume un tiempo T fijo, totalmente independiente de cuál sea la calidad de las cubiertas involucradas. Hay diferentes sets de cubiertas: algunas permiten mayor velocidad que otras, y algunas tienen mayor vida útil que otras, es decir, permiten realizar un mayor número de vueltas. Sabiendo que se cuenta con n sets de cubiertas, que $t_1, t_2, \ldots, t_n$ son los **tiempos por vuelta** que pueden obtenerse con cada uno de ellos, y que $v_1, v_2, \ldots, v_n$ es la vida útil medida en **cantidad de vueltas** de cada uno de ellos, se pide obtener un algoritmo que devuelva el tiempo de carrera mínimo cuando la carrera consta de m vueltas.

   Antes de dar la solución, especificá con tus palabras qué calcula la función recursiva que resolverá el problema, detallando el rol de los argumentos y la llamada principal.

Primero, analizo el enunciado:

- Reemplazar el set de cuatro cubiertas insume un tiempo T fijo, independientemente de cuál sea la calidad de las cubiertas involucradas.
- Tenemos n sets de cubiertas, donde:
  - $t_1, t_2, \ldots, t_n$ son los tiempos por vuelta que puede obtenerse con cada set
  - $v_1, v_2, \ldots, v_n$ es la cantidad de vueltas que dura cada set de cubiertas

  O sea, $v_i * t_i$ es el tiempo total que se tarda en hacer v_i vueltas con el set de cubiertas i.

Se desea dar el tiempo de carrera mínimo cuando la carrera consta de m vueltas.

FUNCIÓN RECURSIVA

carrera(i, j) = "mínimo tiempo de carrera posible que se puede obtener usando algunos de los sets de cubiertas (o todos) desde el 1 hasta el i, de manera tal que se completen las j vueltas restantes"

Rol de los argumentos:

- El argumento i indica hasta qué set de cubiertas se está considerando.
- El argumento j indica la cantidad de vueltas restantes.

LLAMADA PRINCIPAL:

carrera(n, m)

IMPLEMENTACIÓN

```
carrera(i, j) =
     {- CASOS BASE -}
     {- Caso base 1: aún me quedan vueltas por recorrer (j > 0) pero no me
     quedan más sets de cubiertas disponibles (i=0). El problema no tiene
     solución. -}
     |i = 0 && j > 0 --------> infinito

     {- Caso base 2: ya no quedan vueltas por recorrer. -}
     |j = 0 -------------------> 0
```

```
{- CASOS RECURSIVOS -}
{- Caso recursivo 1: con el set de cubiertas i, puedo hacer más vueltas
de las que me faltan por recorrer (v_i > j), entonces con ese set solo
hago las vueltas que faltan. -}
|i > 1 && j > 0 && v_i > j  --------> min(j * t_i  + T, carrera(i-1, j))

{- Caso recursivo 2: pruebo haciendo las vueltas correspondientes con el
set i, y pruebo hacerlas sin el set i. -}
|i > 1 && j > 0 && v_i ≤ j  --------> min(v_i * t_i  + T + carrera(i-1, j-
v_i),
                                        carrera(i-1, j))

{- OPCIÓN PARA JUNTAR LOS DOS CASOS RECURSIVOS -}
|i > 1 && j > 0 -----> min(t_i * (v_i 'min' j)  + T + carrera(i-1, j-v_i
'max' 0),
                          carrera(i-1, j))
```

EJERCICIO 2

2. (Programación dinámica) Escribí un algoritmo que utilice Programación Dinámica para resolver el ejercicio del punto anterior.
   (a) ¿Qué dimensiones tiene la tabla que el algoritmo debe llenar?
   (b) ¿En qué orden se llena la misma?
   (c) ¿Se podría llenar de otra forma? En caso afirmativo indique cuál.

Representaré a cada set de cubiertas mediante la siguiente tupla:

```
type Cubierta = tuple
                    vueltas: nat
                tiempo: nat
            end tuple

fun carrera(cubiertas: array[1..n] of Cubierta, vueltas: nat, T: nat)
                                                    ret min_tiempo: nat
    var tabla: array[0..n, 0..vueltas] {- tabla[i,j] = carrera(i,j) -}

    for i := 0 to n do
      tabla[i, 0] := 0
    od

    for j := 1 to vueltas do
      tabla[0, j] := infinito
    od
```

```
    for i := 1 to n do
      for j:= 1 to vueltas do
        if cubiertas[i].vueltas > vueltas then
          tabla[i, j] := min(j * cubiertas[i].tiempo + T, tabla[i-1,j]
        else if cubiertas[i].vueltas ≤ vueltas then
          tabla[i, j] := min(cubiertas[i].vueltas * cubiertas[i].tiempo + T
           +
                                tabla[i-1, j - cubiertas[i].vueltas],
                                tabla[i-1,j])
        fi
      od
    od

    min_tiempo := tabla[n, vueltas]
end fun
```

a) DIMENSIONES DE LA TABLA

Las dimensiones de la tabla que el algoritmo debe llenar son (n+1) x (vueltas+1):

```
    var tabla: array[0..n, 0..vueltas]
```

b) ORDEN EN QUE SE LLENA LA TABLA

La tabla se llena de izquierda a derecha y de arriba a abajo.

c) ¿SE PUEDE LLENAR DE OTRA FORMA?

La tabla se tiene que llenar sí o sí de arriba hacia abajo verticalmente, pero horizontalmente puede llenarse en cualquier orden.

EJERCICIO 3

3. (Comprensión de algoritmos) Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

   (a) ¿Qué hace?
   (b) ¿Cómo lo hace?
   (c) El orden del algoritmo, analizando los distintos casos posibles.
   (d) Proponer nombres más adecuados para los identificadores (de variables y procedimientos).

```
proc p(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
```

a) ¿QUÉ HACE?

➔ Algoritmo p
Dado un arreglo a[1..n] de naturales, ordena el arreglo de forma creciente.

➔ Algoritmo f
Dado un arreglo a[1..n] de naturales, devuelve un arreglo b[1..n] cuyos elementos son índices de los elementos de a (es decir b[i] representa al elemento a[b[i]]) que indican cómo deberían estar ordenados los elementos de a para que a esté ordenado crecientemente.
Para clarificar, pongo un ejemplo:
Supongamos que a es un arreglo de tamaño [1..5] y el resultado de ejecutar el algoritmo f es el siguiente arreglo b: [5,3,2,1,4]. El arreglo b indica que para que a esté ordenado, el elemento a[5] debería estar primero, a[3] segundo, a[2] tercero, a[1] cuarto y a[4] último.

b) ¿COMO LO HACE?

➔ Algoritmo p
Es un selection_sort:

> Encuentra el elemento mínimo del arreglo y lo intercambia con el elemento que está en la primera posición.
> Encuentra el elemento mínimo de los restantes (es decir sin considerar el primer elemento, que ya está en su posición definitiva) y lo intercambia con el elemento que se encuentra en la segunda posición.
> En cada paso, ordena un elemento hasta que el arreglo quede ordenado completamente.

➔ Algoritmo f

Es una especie de selection_sort, explicado recién, con una modificación:

En lugar de hacer los intercambios entre los elementos de a, lo que hace es hacer los intercambios en el arreglo b, que contiene los índices de los elementos de a, y al arreglo a lo deja intacto.

c) ORDEN DE LOS ALGORITMOS

➔ Algoritmo p

La operación representativa de este algoritmo es la comparación entre elementos del arreglo de entrada.

Ahora bien, veamos que para cada valor del ciclo principal (i), se hacen

n - j + 1 = n - (i+1) + 1 = n - i - 1 + 1 = n - i

comparaciones entre elementos del arreglo de a.

Luego, se hacen n-i comparaciones para $i \in \{1,2,3, \dots , n\}$. Por lo tanto, en total:

(n-1) + (n-2) + (n-3) + … + (n-(n-1)) + (n-n) = (n-1) + (n-2) + (n-3) + … + 1 + 0 =

$= \sum_{i=1}^{n-1} i = \frac{n*(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$

Por lo tanto, el algoritmo p es de orden n^2.

➔ Algoritmo f

No es difícil de ver que este realiza las mismas operaciones que el algoritmo p. Si bien hay un ciclo más, en este no se realiza la operación representativa del algoritmo, que es nuevamente la comparación entre elementos de a.

Luego, el algoritmo f es del mismo orden que el algoritmo p, es decir n^2.

d) NOMBRES MÁS ADECUADOS

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

fun index_selection_sort(a:array[1..n] of nat) ret sorted_indexes:array[1..n]
of nat
     var min_pos: nat
     for i := 1 to n do sorted_indexes[i] := i od
     for i := 1 to n do
          min_pos := i
          for j := i+1 to n do
```

```
            if a[sorted_indexes[j]] < a[sorted_indexes[min_pos]] then
          min_pos:=j fi
          od
          swap(sorted_indexes, i, min_pos)
     od
end fun
```

EJERCICIO 4

(a) A partir de la siguiente implementación de listas mediante punteros, implemente las operaciones copy_list, tail y concat.

```
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
```

a) IMPLEMENTACIÓN DE OPERACIONES

➔ **Operación copy_list**

```
fun copy_list(l1: List of T) ret l2: List of T
     var l1_aux: List of T

     l1_aux := l1
     l2 := empty()

     while not is_empty_list(l1_aux) do
          addr(l2, l1_aux->elem)
          l1_aux := l1_aux->next
     od
end fun
```

OTRA OPCIÓN:

```
fun copy_list(l1: List of T) ret l2: List of T
     var l1_aux: List of T
     var l2_aux: List of T
     var new_node: pointer to (Node of T)

     l1_aux := l1
     l2 := empty()

     if not is_empty_list(l1) then
          alloc(l2)
          l2->elem := l1->elem
          l2->next := null
          l2_aux := l2

          l1_aux := l1_aux->next

          while not is_empty_list(l1_aux) do
               alloc(new_node)
               new_node->elem := l1_aux->elem
               new_node->next := null
               l2_aux->next := new_node
               l2_aux := l2_aux->next
               l1_aux := l1_aux->next
          od
     fi
end fun
```

➔ **Operación tail**

```
proc tail(in/out l: List of T)
{- Recordar que en esta implementación el primer elemento de la lista es el
primer nodo. -}
     var aux: pointer to (Node of T)
     aux := l
     l := l->next
     free(aux)
end proc
```

➔ **Operación concat**

```
proc concat(in/out l: List of T, in l0: List of T)
     var p: pointer to (Node of T)
     p := l0
```

```
     for i:=1 to length(l0) do
          addr(l, p->elem)
          p := p->next
     od
end proc
```

OTRA OPCIÓN:

```
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 proc
```

b) USO DEL TAD LISTA

(b) Implemente una función que reciba una lista de enteros y decida si está ordenado de menor a mayor. Dicha función debe usar el tipo **abstracto** lista, sin importar cuál es su implementación.

```
fun list_is_sorted (l : List of int) ret is_sorted: bool
     var i: nat

     i := 1
     is_sorted := true

     while i < length(l) && is_sorted do
          is_sorted := index(l, i) ≤ index(l, i+1)
          i := i + 1
     od
end fun
```