EXAMEN FINAL 9 DE DICIEMBRE DE 2020 (TEMA 3)

EJERCICIO 1

1. (Backtracking) Debemos llenar con dominós una fila de n casilleros con n par. Cada ficha ocupa 2 casilleros. Contamos con infinitas fichas de todos los tipos. Cada ficha tiene dos números i,j (de 0 a 6) y tiene un puntaje $P_{i,j}$ ($= P_{j,i}$). Como siempre en el dominó, al poner dos fichas juntas los números de los casilleros adyacentes deben coincidir. El último número de la última ficha debe ser un 6. Además, cada casillero tiene un número prohibido $c_1, \ldots, c_n$. Al colocar una ficha en dos casilleros, ésta debe respetar los números prohibidos.

   Escribir un algoritmo que utilice la técnica de backtracking para obtener el máximo puntaje posible colocando las fichas de manera que se repeten todas las restricciones. 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.

   Ejemplo: Con n = 6 debemos usar tres fichas. Si los números prohibidos para los casilleros son

3	2	6	5	0	5

   una posible solución es:

4	3	3	1	1	6

Veamos primero el enunciado:

- Tenemos n casilleros, con n par.
- Cada ficha ocupa 2 casilleros.
- Cada ficha tiene dos números i, j (de 0 a 6) y tiene un puntaje $P_{i,j}$
- El número derecho de cada ficha debe ser igual al número izquierdo de la ficha de al lado
- Cada casillero tiene un número prohibido c_1, …, c_n.
- El último número de la última ficha debe ser un 6.

Se desea obtener el máximo puntaje posible colocando las fichas de manera que se respeten todas las restricciones.

FUNCIÓN RECURSIVA

domino(k, r) = “máximo puntaje posible que se puede obtener para los casilleros 1,..,k, respetando los números prohibidos y colocando en el casillero k la ficha cuyo lado derecho es r”

LLAMADA PRINCIPAL

```
domino(n,6)
```

FUNCIÓN MATEMÁTICA

```
domino(k,r) =  {- CASO BASE: estoy en el casillero 0. -}
               | k = 0 ------> 0

               {- CASO RECURSIVO: notar que solo estoy considerando los
               casilleros pares. -}
               | k > 0 ------> max{m ∈ {1,2,3,4,5,6} tq m ≠ c_{k-1} ∧ m ≠ c_{k-2} } domino(k-2, m)
+ P_{r,m}
```

(definimos artificialmente $c_0 = -1$ para indicar que cuando estoy en el casillero 1 el único número que no puedo poner ahí es el del casillero 1 (c_1), pues el casillero 0 no existe)

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.

```
{- P[i,j] representa el puntaje de una determinada ficha -}
{- prohibidos[i] representa el número prohibido del casillero i -}
fun domino(P: array[0..6, 0..6] of nat,
           prohibidos: array[0..n] of nat) ret max_puntaje: nat
     var tabla: array[0..n, 0..6]

     {- CASO BASE -}
     for i:=0 to 6 do
          tabla[0, i] := 0
     od

     {- CASO RECURSIVO -}
     for i := 2 to n do
         if i % 2 = 0 then {- solo considero los casilleros pares -}
             for j := 0 to 6 do
               mimax := -infinito
               for k := 0 to 6 do
                   if k ≠ prohibidos[i-1] && k ≠ prohibidos[i-2] then
                       mimax := mimax ‘max’ (domino[i-2, k] + P[k, j])
                    fi
               od
               tabla[i,j] := mimax
             od
         fi
     od

     {- LLAMADA PRINCIPAL -}
     max_puntaje := tabla[n,6]
end fun
```

a) DIMENSIONES DE LA TABLA

La tabla es un arreglo de dimensiones (n+1) x 7:

```
var tabla: array[0..n, 0..6]
```

b) ORDEN EN QUE SE LLENA LA TABLA

La tabla se llena desde arriba hacia abajo en las filas y de izquierda a derecha en las columnas. Además, no está de más notar que solamente se llenan las filas pares.

c) ¿SE PUEDE LLENAR DE OTRA FORMA?

En cuanto a las filas, la tabla debe llenarse sí o sí desde arriba hacia abajo, pues para calcular la fila i me tengo que fijar en la fila i-2. Sin embargo, las columnas pueden llenarse en cualquier orden.

EJERCICIO 3

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

   (a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para haga eso?

   (b) ¿Cómo lo hace?

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

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

```
fun s(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[i] < p[y] then y:= i fi
    od
end fun

fun t(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[y] < p[i] then y:= i fi
    od
end fun
```

```
proc r(p: array[1..n] of nat)
    for i := 1 to n div 2 do
        swap(p, i, s(p, i, n-i+1));
        swap(p, n-i+1, t(p, i+1, n-i+1));
    od
end fun
```

a) ¿QUE HACEN? ¿CUALES SON LAS PRECONDICIONES NECESARIAS PARA QUE HAGA ESO?

- ➔ Algoritmo s

  Dado un arreglo p[1..n] de naturales y dos números naturales v y w, devuelve la posición del elemento mínimo en el segmento [v..w] del arreglo p.

  Precondiciones necesarias para que haga esto: $1 \leq v \leq n$, $v \leq w \leq n$.

- ➔ Algoritmo t

  Dado un arreglo p[1..n] de naturales y dos números naturales v y w, devuelve la posición del elemento máximo en el segmento [v..w] del arreglo p.

  Precondiciones necesarias para que haga esto: $1 \leq v \leq n$, $v \leq w \leq n$.

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

b) ¿COMO LO HACEN?

➔ Algoritmo s
Comienza suponiendo que el elemento mínimo del segmento p[v..w] se encuentra en la posición v (y:=v). Luego, recorre el segmento p[v+1..w] de izquierda a derecha y si encuentra un elemento que sea menor al mínimo inicial, le asigna a la variable y la posición de este nuevo mínimo. Si nuevamente encuentra un elemento que sea menor a este mínimo, le asigna su posición a la variable y, y así sucesivamente hasta recorrer todo el segmento.

➔ Algoritmo t
Hace lo mismo que el algoritmo s pero buscando el máximo.

➔ Algoritmo r
Llama a los dos algoritmos anteriores, ordenando en cada iteración el arreglo “desde afuera hacia adentro”.
En la primera iteración, encuentra el elemento mínimo del arreglo (es decir, del segmento p[1..n]) y lo pone en la posición 1; y encuentra el elemento máximo del arreglo y lo pone en la posición n.
En la segunda iteración, encuentra el mínimo de los restantes (es decir, del segmento p[2..n-1]) y lo pone en la posición 2; y encuentra el máximo de los restantes y lo pone en la posición n-1.
En cada iteración va ordenando dos elementos hasta que el arreglo quede completamente ordenado.

c) ORDEN DE LOS ALGORITMOS

➔ Algoritmo s
Es claro que en este algoritmo, la operación que más se repite (representativa) es la comparación entre elementos del arreglo de entrada. Además, no se pueden distinguir distintos casos, puesto que estas comparaciones se hacen siempre.
Veamos que se hace exactamente una comparación para $i \in \{v+1, v+2, ..., w-1, w\}$. Luego, es claro que se hacen en total
$w - (v+1) + 1 = w - v - 1 + 1 = w - v$
comparaciones entre elementos del arreglo p.
Es decir, el algortimo s es de orden O(w-v).

➔ Algoritmo t
Siguiendo un razonamiento análogo que para el algoritmo anterior, es fácil de ver que este algoritmo también es de orden O(w-v).

➔ Algoritmo r
Es claro que el orden de este algoritmo está relacionado con los órdenes de los anteriores, puesto que en cada iteración se los llama. Luego, también es claro que la operación representativa de este algoritmo vuelve a ser la comparación entre elementos del arreglo de entrada.

Ya vimos que s(p, i, n-i+1) es de orden (n-i+1) - i = -2i + n + 1
Ya vimos que t(p, i+1, n-i+1) es de orden (n-i+1) - (i+1) = (n-i+1) - i-1 = -2i + n
Es decir, cada iteración es del orden de (-2i + n + 1) + (-2i + n) = -4i + 2n + 1
Ahora bien, se hace esa cantidad de comparaciones para i ∈ {1, 2, 3, ..., (n div 2) - 1, n div 2}. Luego, en total tenemos:

$$\sum_{i=1}^{n/2}(-4i+2n+1)=\sum_{i=1}^{n/2}-4i+\sum_{i=1}^{n/2}2n+\sum_{i=1}^{n/2}1=-4\sum_{i=1}^{n/2}i+(n/2)*2n+n/2=\frac{-4*(n/2)*(n/2+1)}{2}+n^2+n/2=¿¿$$

$$¿-\frac{2*n^2+2n}{4}+n^2+n/2=\frac{-n^2+2n}{2}+n^2+n/2=-n^2/2-n+n^2+n/2=n^2/2-n/2=O(n^2)$$

Por lo tanto, el algoritmo r es de orden $O(n^2)$.

d) NOMBRES MÁS ADECUADOS

```
fun local_minimum(p:array[1..n] of nat, start: nat, end: nat) ret local_min_pos: nat
     local_min_pos := start
     for i := start + 1 to end do
          if p[i] < p[local_min_pos] then local_min_pos := i fi
     od
end fun

fun local_maximum(p:array[1..n] of nat, start: nat, end: nat) ret local_max_pos: nat
     local_max_pos := start
     for i := start + 1 to end do
          if p[local_max_pos] < p[i] then local_max_pos := i fi
     od
end fun

proc sort_array(p: array[1..n] of nat)
     for i := 1 to n div 2 do
          swap(p, i, local_minimum(p, i, n-i+1))
          swap(p, n-i+1, local_maximum(p, i+1, n-i+1))
     od
end proc
```

EJERCICIO 4

4. Completar la especificación del tipo Conjunto de elementos de tipo T, agregando las operaciones

   - **elim**, que elimina de un conjunto un elemento dado.
   - **union**, que agrega a un conjunto todos los elementos que pertenecen a otro conjunto dado.
   - **dif**, que elimina de un conjunto todos los elementos que pertenecen a otro conjunto dado.

```
spec Set of T where

constructors
(...)

destroy
(...)

operations
(...)

     proc elim(in/out s: Set of T, in e: T)
     {- Elimina al elemento e del conjunto s -}

     proc union(in/out s: Set of T, in s0: Set of T)
     {- Agrega al conjunto s todos los elementos del conjunto s0 -}

     proc dif(in/out s: Set of T, in s0: Set of T)
     {- Elimina del conjunto s todos los elementos de s que pertenecen al
     conjunto s0 -}

end spec
```

EJERCICIO 5

5. A partir de la siguiente implementación de conjuntos utilizando listas ordenadas, implemente el constructor **add**, y las operaciones **member**, **inters** y **cardinal**. La implementación debe mantener el invariante de representación por el cual todo conjunto está representado por una lista ordenada crecientemente. Puede utilizar todas las operaciones especificadas para el tipo lista vistas en el teórico. Para cada operación que utilice, especifique su encabezado, es decir: si es función o procedimiento, cómo se llama, qué argumentos toma y devuelve.

```
implement Set of T where

type Set of T = List of T

fun empty_set() ret s : Set of T
    s := empty_list()
end fun
```

Constructor add

```
proc add (in e: T, in/out s: Set of T)
{- Recordar que una vez que agregue el elemento e, se tiene que seguir preservando el invariante de representación, es decir que la lista que representa al conjunto debe estar ordenada crecientemente. Además, recordar que en los conjuntos no hay elementos repetidos. -}
    var s_aux: List of T
    var i: nat

    s_aux := copy_list(s)
    i := 0

    while not is_empty_list(s_aux) && e > head(s_aux) do
        tail(s_aux)
        i := i + 1
    od
    if is_empty_list(s_aux) V head(s_aux) > e then
        add_at(s, i, e) {- Agrega el elemento e en la posición i de la lista s -}
    fi

    destroy(s_aux)
end proc
```

Operación member

```
fun member(e: T, s: Set of T) ret b: bool
    var s_aux: List of T
    var elem: T
```

```
     s_aux := copy_list(s)
     b := false

     while not is_empty_list(s_aux) && not b do
          b := head(s_aux) = e
          tail(s_aux)
     od

     destroy(s_aux)
end fun
```

Operación inters

```
proc inters(in/out s: Set of T, in s0: Set of T)
     var s_aux: List of T

     s_aux := copy_list(s)

     while not is_empty_list(s_aux) do
          if not member(head(s_aux), s0)
               elim(s, head(s_aux))
          fi
          tail(s_aux)
     od
     destroy(s_aux)
end proc
```

Operación cardinal

```
fun cardinal(s : Set of T) ret n: nat
     var s_aux: List of T

     n := 0
     s_aux := copy_list(s)

     while not is_empty_list(s_aux) do
          n := n + 1
          tail(s_aux)
     od
end fun
```