Ejercicios introductorios de Elección de Estructuras

Román Gorojovsky

30 de Octubre de 2024

Enunciados

Los dos ejercicios que iba a presentar hoy no explican demasiado si se los presento directamente resueltos, por lo que les propongo que si todavía no los intentaron se tomen un rato para pensarlos antes de seguir leyendo. No hace falta que escriban todo para entender la explicación, pero ayuda que los hayan pensado y se hayan encontrado con las dificultades que tienen. Son los ejercicios 5 y 6 de la práctica 8, a saber:

Ejercicio 5. El TAD *Matriz infinita de booleanos* tiene las siguientes operaciones:

- *Crear*, que crea una matriz donde todos los valores son falsos.
- *Asignar*, que toma una matriz, dos naturales (fila y columna) y un booleano, y asigna a este último en esa coordenada. (Como la matriz es infinita, no hay restricciones sobre la magnitud de fila y columna.)
- *Ver*, que dadas una matriz, una fila y una columna devuelve el valor de esa coordenada. (Idem.)
- *Complementar*, que invierte todos los valores de la matriz.

Ejemplo de uso del módulo:

```
MatrizInfinita M := Crear()
bool b1 := Ver(M, 0, 0)
Asignar(M, 1, 3, False)
Asignar(M, 100, 5000, True)
bool b2 := M.Ver(100, 5000)
Complementar(M)
bool b3 := Ver(M, 394, 788)
bool b4 := Ver(M, 100, 5000)
```

Tras lo que deberíamos tener

```
b1 = False
b2 = True
b3 = True
b4 = False
```

Elija la estructura y escriba los algoritmos de modo que las operaciones *Crear*, *Ver* y *Complementar* tomen $O(1)$ tiempo en peor caso.

Ejercicio 6. Una matriz finita posee las siguientes operaciones:

- *Crear*, con la cantidad de filas y columnas que albergará la matriz.
- *Definir*, que permite definir el valor para una posición válida.
- *#Filas*, que retorna la cantidad de filas de la matriz.
- *#Columnas*, que retorna la cantidad de columnas de la matriz.

- *Obtener*, que devuelve el valor de una posición válida de la matriz (si nunca se definió la matriz en la posición solicitada devuelve cero).
- *SumarMatrices*, que permite sumar dos matrices de iguales dimensiones.

Dado n y m son la cantidad de elementos no nulos de A y B, respectivamente, diseñe un módulo (elegir una estructura y escribir los algoritmos) para el TAD *MatrizFinita* de modo tal que dadas dos matrices finitas A y B,

(a) *Definir* y *Obtener* aplicadas a A se realicen cada una en $\Theta(n)$ en peor caso, y

(b) *SumarMatrices* aplicada a A y B se realice en $\Theta(n+m)$ en peor caso,

Soluciones

Matriz Infinita

Cuando nos piden hacer una matriz y nos piden que la operación *Ver* sea $O(1)$ lo primero que debería ocurrírsenos es usar arreglos para acceder en esa complejidad a la íesima posición de cada dimensión. Considerando que las matrices que vamos a usar son de tamaño arbitrario, tiene sentido usar vectores, es decir, arreglos redimensionables, en vez de usar directamente arreglos, ya que la complejidad de agregar nuevos elementos cuando superamos la capacidad inicial es asintóticamente $O(1)$[1]

Entonces proponemos esta primer estructura:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <
    var matriz: Vector<Vector<bool>>
>
```

Veamos informalmente si con esta estructura podríamos cumplir las complejidades que nos piden

- `proc Ver`(in $M : MIB$, in $f : \mathbb{Z}$, in $c : \mathbb{Z}$) : *bool*
  Es $O(1)$ porque, en principio, son dos accesos a vector.
- `proc Crear`() : MIB
  ¿Es $O(1)$? ¿Qué debería hacer el algoritmo de esta operación? El estado inicial de una matriz de booleanos es que todos los valores estén en *False*, es decir que `ver(M, f, c)` debería devolver **False** para cualquier valor. Entonces, para crear en $O(1)$, podemos inicializar con un vector vacío y en el algoritmo de `Ver()` devolver false para cualquier posición mientras no se asigne **`true`** en algún lado. De hecho, vamos a ver en el algoritmo completo que esto es cierto para cualquier f y c que sean mayores que las dimensiones mayores que asignadas hasta este momento.

  Con este cambio, el proc `ver` pasa a ser:

  - Si f y c son menores que los tamaños de los arreglos, devolver el valor de la posición
  - Si no, devolver `False`
- `proc Asignar`(inout $M : MIB$, in $f : \mathbb{Z}$, in $c\mathbb{Z}$, in *bbool*)
  No nos piden nada sobre la complejidad de esta operación, pero de todos modos veamos qué complejidad estaría teniendo. El peor caso es cuando tanto f como c sean mayores a las dimensiones asignadas previamente, en cuyo caso hay que recorrer todas las filas existentes, pedir memoria para extender los vectores y luego pedir las filas que falten y crear los vectores de tamaño c en cada posición. Esta operación es $O(f \times c)$, aunque como estamos trabajando con vectores en vez de arreglos no hay que copiar todos los datos al extender los que ya existen.

[1] Los detalles de esto están en las clases teóricas y también en el apunte de módulos básicos

- `proc Complementar`(inout $MMIB$)
  Acá tenemos un problema: con la estructura que tenemos la única forma de implementar esta operación es recorriendo todos nuestros vectores de vectores y cambiando el valor en cada posición, lo que claramente no es $O(1)$.

¿Cómo resolvemos esto? Lo que podemos hacer es tener dos vectores de vectores, uno con los valores como entran y otro con los valores invertidos, más un booleano para definir de cuál leer. Entonces el proc `complementar` todo lo que hace es invertir ese booleano:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <
    var matriz: Vector<Vector<bool>>
    var matrizComplementada: Vector<Vector<bool>>
    var devolverComplementada: bool
>
```

Veamos de nuevo cómo serían informalmente los algoritmos y si cumple las complejidades

- `proc Ver`(in $M : MIB$, in $f : \mathbb{Z}$, in $c : \mathbb{Z}$) : *bool*
  Ahora es:
  - Si f y c son menores que los tamaños de los arreglos, devolver el valor de la posición en la matriz determinada por `devolverComplementada`
  - Si no, devolver `False` o `True` según corresponda

  Todas operaciones en $O(1)$
- `proc Crear`() : MIB
  Sigue prácticamente igual, $O(1)$.
- `proc asignar`(inout $M : MIB$, in $f : \mathbb{Z}$, in $c\mathbb{Z}$, in *bbool*)
  Ahora hay que cargar los datos en dos matrices separadas, que es dos veces la operación anterior así que el orden sigue siendo el mismo.
- `proc Complementar`(inout $MMIB$)
  Ahora esto es simplemente invertir un booleano, que es claramente $O(1)$

Con esto tenemos una solución correcta del ejercicio, pero no es la ideal. Las dos matrices están duplicando información, lo que deberíamos controlar en el Invariante de Representación, pero además en este caso duplicar la información es redundante. Podríamos guardar todo en una sola matriz y según el booleano decidir si complementar el valor que hay en la matriz antes de devolverlo en `ver` o el que entra en `asignar`. El módulo quedaría:

```
Módulo MIB implementa MatrizInfinitaDeBooleanos <
    var matriz: Vector<Vector<bool>>
    var devolverComplementada: bool
>
```

Y ahora sólo falta escribir los algoritmos:

```
function Crear : MIB
    bool res := new MIB
    res.matriz = new Vector<Vector<bool>>
    # El booleano se inicializa solo en False
    return res
end function
```

```
function VER(inout M: MIB, in f: int, in c: int) : bool
    bool res := False
    if f < longitud(M.matriz) ∧ c < longitud(M.matriz[0]) then
        res := M.matriz[f][c]
    end if
    if M.devolverComplementada then
        res := ¬ res
    end if
    return res
end function

function ASIGNAR(inout M: MIB, in f: int, in c: int, in b: bool)
    bool b_a_guardar = b
    if M.devolverComplementada then
        b_a_guardar := ¬ b_a_guardar
    end if
    if b_a_guardar = True ∧ (f ≥ longitud(M.matriz) ∧ c ≥ longitud(M.matriz[0])) then
        # Pedir memoria para los vectores según sea necesario
    end if
    M.matriz[f][c] := b_a_guardar
end function

function COMPLEMENTAR(inout M: MIB)
    M.devolverComplementada := ¬ M.devolverComplementada
end function
```

Matriz Finita

En este ejercicio lo más importante de entender es que la complejidad está expresada no en términos de la dimensión de la matriz, sino de sus elementos no nulos (que como estamos hablando de matrices quiere decir “distintos de cero”, no “`!= null`”).

Veamos cómo quedarían los algoritmos i intentamos guardar todas las posiciones en un vector de vectores y calculemos las complejidades para una matriz de dimensiones $f \times c$

- *Crear* Hay que pedir memoria para todas las posiciones y ponerlas en 0 (porque *Obtener* tiene que devolver 0 cuando algo no está definido). Eso va a ser $\Theta(f \times c)$
- *Definir* y *Obtener* Hay que acceder a una posición en un vector dentro otro al que también accedo por posición. Eso va a tomar $\Theta(1)$
- *SumarMatrices* Hay que recorrer las dos matrices posición a posición y guardar el resultado de las sumas en algún lado. Esto va a ser $\Theta(f \times c)$

Esto último no cumple la complejidad pedida. Por un lado, la cantidad de elementos no nulos debería ser siempre menor o igual a la cantidad de posiciones de la matriz, es decir $n \leq f \times c$. Pero además las dimensiones de la matriz crecen independientemente de la cantidad de posiciones nulas, y la complejidad que nos piden está en función de eso. Es razonable pensar que, dadas las cotas de complejidad que se piden, este módulo se va a usar para guardar matrices que tienen proporcionalmente pocos elementos no nulos.

Por lo tanto tendríamos que buscar una opción donde sólo guardemos estos elementos. Les propongo el siguiente módulo

```
ElemMatriz es Struct<f: int, c: int, valor: int>

  Módulo MatrizFinitaLista implementa MatrizFinita <
        var elementosNoNulos: ListaEnlazada<ElemMatriz>
        var filas: int
        var columnas: int
  >
```

¿Cómo represento una matriz con una lista enlazada? ¿Por qué una lista enlazada en vez de otra estructura?

- Cada nodo va a tener la posición (f y c) y el valor.
- Ningún elemento de la lista debería tener f y c dentro de las dimensiones definidas en filas y columnas
- Voy a pedir que la lista esté ordenada por f y después por c. Esto va a hacer que para definir una posición nueva tenga que buscar dónde insertarla en la lista, pero como vamos a ver las complejidades que nos piden lo permiten. También va a ser útil para la operación de sumar.

Todo esto debería estar especificado en el invariante de representación, que queda como ejercicio para ustedes.

Veamos las complejidades de estas operaciones

- *Crear* va a ser $\Theta(1)$, ya que hay que crear una lista vacía
- *Definir* y *Obtener* van a tomar $\Theta(n)$ ya que en peor caso hay que recorrer la lista entera para encontrar la posición que buscamos (o descubrir que no está)
- *SumarMatrices* Para hacer esta suma podemos recorrer las dos listas, que pedimos que estén ordenadas nodo a nodo y, suponiendo que guardamos la suma en A [2] tenemos dos casos:
  - encontramos una posición no nula en B que no es nula en A: agregamos esa posición en $O(1)$ porque como vamos recorriendo ambas listas ordenadas estamos en la posición dónde debería insertarse el siguiente nodo
  - encontramos una posición no nula en B que tampoco es nula en A: sumamos los valores de ambas en $O(1)$

Pero momento. El módulo básico ListaEnlazada<T> del apunte y su iterador no permiten hacer este tipo de inserciones que estamos planteando. Para poder implementar esto como quereos necesitamos poder trabajar directamente con los nodos de la lista. Entonces tenemos que redefinir el módulo usando el NodoLista que está en el apunte y nos queda

```
  Módulo MatrizFinitaLista implementa MatrizFinita <
        var primerElementoNoNulo: NodoLista<ElemMatriz>
        var filas: int
        var columnas: int
  >
```

Ahora vamos a tener que agregar al invariante del módulo el predicado *esLista*? visto en otras clases, para asegurar que tengamos una lista válida, cosa que antes no necesitábamos.

Con esto definido, una vez más sólo queda escribir los algoritmos

[2] Podríamos guardarla en una matriz nueva, pero esto no afectaría la complejidad

```
function CREAR(in f: int, in c: int) : MatrizFinitaLista
    MatrizInfinitaLista res := new MatrizInfinitaLista        ▷ Pido memoria para el módulo
    res.primerElementoNulo := nil
    res.filas = f
    res.columnas = c return res
end function

function #FILAS(in M: MatrizFinitaLista)
    return M.filas
end function

function #FILAS(in M: MatrizFinitaLista)
    return M.columnas
end function

function DEFINIR(inout M: MatrizFinitaLista, in f: int, in c: int, in v: int)
    NodoLista<ElemMatriz> actual := M.primerElementoNoNulo
    while actual.siguiente ≠ nil ∧ actual.val.f < f do
        actual := actual.siguiente
    end while
    if actual.siguiente = nil ∨ actual.val.f < f then        ▷ No está la fila, agrego nodo nuevo
        AgregarNodoNuevoAtras(actual, new ElemMatriz(f, c, v))
    else        ▷ Encontré la fila, hay que encontrar la columna
        while actual.siguiente ≠ nil ∧ actual.f = f ∧ actual.c < c do
            actual := actual.siguiente
        end while
        if actual.siguiente = nil ∨ actual.val.f > f ∨ ∨ actual.val.c > c then        ▷ No está la columna
            AgregarNodoNuevoAtras(actual, new ElemMatriz(f, c, v))
        else        ▷ Encontré f y c, hay que cambiar el valor nomás
            actual.valor = v
        end if
    end if
end function

function OBTENER(in M: MatrizFinitaLista, in f: int, in c: int) : int
    NodoLista<ElemMatriz> actual := M.primerElementoNoNulo
    int res := 0
    while actual.siguiente ≠ nil ∧ actual.val.f < f do
        actual := actual.siguiente
    end while
    if actual.val.f = f then        ▷ Encontré la fila, busco la columna
        while actual.siguiente ≠ nil ∧ actual.val.f = f ∧ actual.val.c ¡c do
            actual := actual.siguiente
        end while
    end if
    if actual.val.f = f ∧ actual.val.c then        ▷ ¡Encontré el elemento!
        res := actual.val.v
    end if
    return res
end function
```

```
function SumarMatrices(inout A: MatrizFinitaLista, in B: MatrizFinitaLista)
    NodoLista<ElemMatriz> actualA := A.primerElementoNoNulo
    NodoLista<ElemMatriz> actualB := B.primerElementoNoNulo
    while actualA.siguiente ≠ nil do
        while actualB.siguiente ≠ nil ∧ EsAnterior(actualB, actualA) do
            AgregarNodoNuevoAtras(actualA, actualB.val)
            actualB := actualB.siguiente
        end while
        if actualA.val.f = actualB.val.f ∧ actualA.val.c = actualB.val.c then
            actualA.val.v := actualA.val.v + actualB.val.v
        end if
        actualA := actualA.siguiente
    end while
    if actualA.siguiente = nil ∧ actualB.siguiente ≠ nil then        ▷ Todavía hay que copiar de B
        while actualB≠ nil do                                        ▷ Agrego adelante en cada iteración
            NodoLista<ElemMatriz> n := new NodoLista<ElemMatriz>
            n.val.f := actualB.val.f
            n.val.c := actualB.val.c
            n.val.valor := actualB.val.v
            n.anterior := actualA
            actualA.siguiente := n
            actualA := actualA.siguiente
        end while
    end if
end function

function AgregarNodoNuevoAtras(inout actual: NodoLista<ElemMatriz>, in elem: ElemMatriz)
    NodoLista<ElemMatriz> n := new NodoLista<ElemMatriz>        ▷ Pido memoria para el nodo
    n.val.f := elem.f
    n.val.c := elem.c
    n.val.valor := elem.v
    n.siguiente := actual
    actual.anterior := n
end function

function EsAnterior(in nodo1: NodoLista<ElemMatriz>, in nodo2: NodoLista<ElemMatriz>)bool
    return nodo1.val.f ¡nodo2.val.f ∨ (nodo1.val.f = nodo2.val.f ∧ nodo1.val.c ¡nodo2.val.c)
end function
```