Elección de estructuras

Recomendaciones varias para estos ejercicios:

Lo que más conviene es: leer y entender bien el enunciado. Repetir recursivamente hasta entenderlo bien.

Siempre, o casi siempre, nos van a dar datos clave en él. Hay que ponerse en el rol de detective. La mayoría de las veces nos vamos a encontrar con un dato (o más de uno) que está acotado: debemos reconocerlos. Que un dato esté acotado nos va a permitir elegir casi cualquier casi estructura para representarlo.

Los pasos serían:
1. Leer bien el enunciado.
2. Mirar el apunte de Módulos.
3. Ir de los procs de menor complejidad a mayor complejidad.
4. Elegir estructuras que nos permitan cumplir las complejidades del **3** e ir ajustándola para que funcionen con procs de mayor complejidad.

Otra, y última cosa, a veces vamos a tener que elegir si la eficiencia la vamos a tener al escribir o al leer. Pocas veces pasa que la lectura y escritura son eficientes en simultáneo, así que, en muchos ejercicios nos piden que cumplamos ciertas complejidades para leer (o escribir), pero no para escribir (o leer). Hay veces que sólo nos van a pedir leer, o sólo nos van a pedir escribir. Leamos bien, muy bien, el enunciado. No hace falta pensar la estructura perfecta, solo la mínima posible para que cumpla con lo pedido.

Ejercicio 4

En esta sección nos concentraremos en la elección de estructuras y los algoritmos asociados. Cuando el enunciado diga "diseñe este módulo", salvo que indique lo contrario, sólo nos interesa:

1. La estructura
2. Su invariante de representación y su función de abstracción
3. Las operaciones con sus parámetros y su complejidad
4. Los algoritmos correspondientes

Ejercicio 4. Se desea diseñar un sistema para registrar las notas de los alumnos en una facultad. Al igual que en Exactas, los alumnos se identifican con un número de LU. A su vez, las materias tienen un nombre, y puede haber una cantidad no acotada de materias. En cada materia, las notas están entre 0 y 10, y se aprueban si la nota es mayor o igual a 7.

```
TAD Sistema {
obs notas: dict⟨materia, dict⟨alumno,ℤ⟩⟩

proc nuevoSistema() : Sistema
proc registrarMateria(inout s : Sistema, in m : materia)
proc registrarNota(inout s : Sistema, in m : materia, in a : alumno, in n : nota)
proc notaDeAlumno(in s : Sistema, in a : alumno, m : materia) : nota
proc cantAlumnosConNota(in s : Sistema, in m : materia, n : nota) : ℤ
proc cantAlumnosAprobados(in s : Sistema, in m : materia) : ℤ
}
```

Dados $m = cantmaterias$ y $n = cantalumnos$ se desea diseñar un módulo con los siguientes requerimientos de complejidad temporal:

- nuevoSistema O(1)
- registrarMateria en O(log m)
- registrarNota en O(log n + log m)
- notaDeAlumno en O(log n + log m)
- cantAlumnosConNota y CantAlumnosAprobados en O(log m)

Figura 1: Enunciado Problema 4

Hay dos formas de resolver este ejercicio (posiblemente más). La primera es más intuitiva así que vamos con esa. Luego de leer bien el enunciado nos damos cuenta de que las **notas** están acotadas. Solo hay 11 notas posibles, y lo vamos a usar a nuestro favor.

Alumno = *string*
Materia = *string*
Nota = *int*

Modulo SistemaImpl implements Sistema {
var **notas**: DiccionarioLog<Materia, Array<ConjuntoLog<Alumno>>>

proc nuevoSistema () : SistemaImpl

```
res.notas:= diccionarioVacío<>() // O(1)

return res // O(1)
/**
* Complejidad final
* O(1)
*/
```

proc registrarMateria (inout s: SistemaImpl, in m: Materia)

```
// En algún lado debería haber un requiere que materia aún no está registrada.
var valor: Array<ConjuntoLog<Alumno>>
    valor:= new Array(11) // O(11) == O(1)

var j: int
    j:= 0

while (j <= 11) do // O(11) == O(1)
  valor[j]:= conjuntoVacío<Alumno>() // O(1)
  j:= j + 1
endwhile

s.notas.definirRápido(m, valor) // O(log(m))
/**
 * Complejidad final
 * O(log(m))
 */
```

proc registrarNota (inout s: SistemaImpl, in m: Materia, in a: Alumno, in nota: Nota)

```
// En algún lado debería haber un requiere que ese alumno no tiene una nota registrada.
// para esa materia. Y que nota está en rango.
var notasDeMateria: Array<ConjuntoLog<Alumno>>
    notasDeMateria:= s.notas.obtener(m) // O(log(m))

// O(log(n)). Peor caso: Todos los alumnos tienen la misma nota en la misma materia.
notasDeMateria[nota].agregarRapido(a) // O(log(n))
notasDeMateria.definir(m, notasDeMateria) // O(log(m))

/**
 * Complejidad final
 * O(log(m) + log(n))
 */
```

proc notaDelAlumno (in s: SistemaImpl, in m: Materia, in a: Alumno) : Nota

```
// En algún lado debería haber un requiere que ese alumno tiene una nota
// (convenientemente única) en esa materia.
var notasDeMateria: Array<ConjuntoLog<Alumno>>
    notasDeMateria:= s.notas.obtener(m) // O(log(m))

var j: int
var nota: int
    j:= 0
    nota:= 0

while (j <= 11) do // O(11 * log(n)) == O(log(n))
  if notasDeMateria[j].pertenece(alumno) then // O(log(n))
    nota:= j
  endif

  j:= j + 1
endwhile

return nota
/**
 * Complejidad final
 * O(log(m) + log(n))
 */
```

proc cantAlumnosConNota (in s: SistemaImpl, in m: Materia, in nota: Nota) : int

```
var notasDeMateria: Array<ConjuntoLog<Alumno>>
    notasDeMateria:= s.notas.obtener(m) // O(log(m))

return notasDeMateria[nota].tamaño()
/**
 * Complejidad final
 * O(log(m))
 */
```

proc cantAlumnosAprobados (in s: SistemaImpl, in m: Materia) : int

```
var notasDeMateria: Array<ConjuntoLog<Alumno>>
    notasDeMateria:= s.notas.obtener(m) // O(log(m))

var j: int
var suma: int
    j:= 7
    suma:= 0

while (j <= 11) do // O(1) == O(1)
  suma:= suma + cantAlumnosConNota(s, m, j) // O(1)
  j:= j + 1
endwhile
/**
 * Complejidad final
 * O(log(m))
 */
```

}

Ejercicio 5

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.

Figura 2: Enunciado Problema 5

Nos tenemos que poder dar cuenta después de **dos** cosas después de nuestro intensivo estudio del enunciado.

La primera es que los valores están acotados, solo pueden haber dos casos, true o false. Eso es clave.

La segunda es que no nos piden ninguna complejidad para escribir (el ej. busca lectura eficiente), así que: Sacrifiquemos complejidad en ese proc.

Con el primer dato, podemos inteligentemente, armar una estructura que nos permita Complementar en O(1).

Si recorremos cada uno de los Módulos del apunte, nos vamos a dar cuenta que ninguna estructura nos sirve para leer en O(1), salvo Vector.

Así que... tendremos que usar Vector: **necesitamos** leer un par i,j en O(1).

Colunna = *int*
Fila = *int*

```
Modulo MatrizInfinitaDeBooleanosImpl implements MatrizInfinitaDeBooleanos {
var datos: Vector<Vector<bool>>
var estaComplementada: bool
```

proc agrandarFila (inout m: MatrizInfinitaDeBooleanosImpl, in f: Fila, in c: Columna)

```
1  if f > m.datos.obtener(f).longitud() then
2    int j: int
3        j:= 0
4
5    while (j <= c) do // O(m)
6      // Insertamos false: Valor por default.
7      m.datos.obtener(f).agregarAtras(false) // O(f(m)). Amortizado: O(2).
8      j:= j + 1 // O(1)
9    endwhile
10 endif
11 /**
12  * Complejidad final
13  * O(m)
14  */
```

proc agrandarFilas (inout m: MatrizInfinitaDeBooleanosImpl, in f: Fila, in c: Columna)

```
1  if f > m.datos.longitud() then
2    int j: int
3        j:= 0
4
5    while (j <= f) do          // O(n * m)
6      m.agrandarFila(m, j, c) // O(m)
7      j:= j + 1 // O(1)
8    endwhile
9  endif
10 /**
11  * Complejidad final
12  * O(n * m)
13  */
```

```
% Asignar %
```

proc Asignar (inout m: MatrizInfinitaDeBooleanosImpl, in f: Fila, in c: Columna, in v: bool)

```
1  m.agrandarVector(m, f, c)                    // O(n * m)
2  m.datos.obtener(f).modificarPosicion(c, v) // O(1)
3  /**
4   * Complejidad final
5   * O(n * m)
6   */
```

```
% Procs O(1) %
```

proc obtenerValor (in m: MatrizInfinitaDeBooleanosImpl, in f: Fila, in c: Columna) : bool

```
1  if f >= m.datos.lontitud() || c >= m.datos.obtener(f).longitud() then // O(1)
2    // Devolvemos: Valor por default.
3    return false // O(1)
4  else
5    return m.datos.obtener(f).obtener(c) // O(1)
6  endif
7  /**
8   * Complejidad final
9   * O(1)
10  */
```

proc crear () : MatrizInfinitaDeBooleanosImpl

```
1  res.datos:= vectorVacío<>() // O(1)
2  res.estaComplementada:= false // O(1)
3  return res // O(1)
4  /**
5   * Complejidad final
6   * O(1)
7   */
```

proc Ver (inout m: MatrizInfinitaDeBooleanosImpl, in f: Fila, in c: Columna) : bool

```
1  var v: bool
2      v:= obtenerValor(m, f, c) // O(1)
3
4  return estaComplementada ? !v : v // O(1)
5  /**
6   * Complejidad final
7   * O(1)
8   */
```

proc Complementar (inout m: MatrizInfinitaDeBooleanosImpl)

```
1  m.estaComplementada = !m.estaComplementada
2  /**
3   * Complejidad final
4   * O(1)
5   */
```

```
}
```

Ejercicio 6

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,

Figura 3: Enunciado Problema 6

Nos hablan de $\Theta(n)$ en peor caso: Qué sugiere?

Cuando se habla de $\Theta(n)$ en el peor caso, significa que el comportamiento del algoritmo está acotado tanto superior como inferiormente por n (Tpeor $\in O(n)$ y Tpeor $\in \Omega(n)$).

En cambio $O(n)$ solo describe una cota superior, lo que permite que el algoritmo sea más eficiente en ciertos casos, incluso $O(1)$ y aún así cumpliría con $O(n)$.

Dicho esto: ¿Por qué nos piden Θ y no O? ¿Cambia en algo?

Como se mencionó, si nos pidieran que el peor caso es $O(n)$, tranquilamente podríamos diseñar un algoritmo que sea $O(1)$ y estaríamos cumpliendo con lo pedido. Pero con $\Theta(n)$ no podemos hacer eso.

Podríamos usar un Vector de que almacene cosas de tipo Dato (que contenga fila, columna, dato), forzar una búsqueda lineal y siempre mantenernos en $\Theta(n)$. Esto funcionaría realmente bien para **Definir** y **Obtener**.

Pero, hay un gran problema... la suma de matrices. Nunca, pero nunca (con las estructuras que podemos usar), vamos a poder cumplir con la complejidad $\Theta(n+m)$, siempre nos vamos a "pasar", ya que, para cada elemento de la matriz B tendremos que buscar en A (Búsqueda Lineal) si existe un **Dato** con ese par i,j para realizar la suma y eso nos aumentaría la complejidad a $\Theta(n*m)$.

Pero, ¿Qué pasa si mantenemos ordenado el Vector? ¿Cuál es el costo de mantener ordenado un Vector a medida que vamos insertando?

Podemos hacer una búsqueda lineal, encontrar la posición ideal, y luego: mover a la derecha todos los elementos del vector a partir de esa posición (Búsqueda Binaria también estaría OK) con un costo total de $\Theta(n)$.

Mantenerlo ordenado nos a permitir cumplir con la complejidad de la suma de matrices, puesto que, al estar ordenada, por medio de dos punteros podremos recorrer los Vectores de A y B de manera lineal y exacta: $\Theta(n+m)$

Dato $= f : int, \texttt{Struct}\langle \texttt{c: } int, \texttt{ valor: } int\rangle$
Columna $= int$
Fila $= int$

```
Modulo MatrizFinitaImpl implements MatrizFinita {
    var datos: Vector<Dato>
    var nroFilas: int
    var nroColumnas: int
```

proc esMasPrioritario (in d1: Dato, in d2: Dato) : bool

```
return d1.f < d2.f || (d1.f == d2.f && d1.c < d2.c)
```

proc actualizarNumeros (inout m: MatrizFinitaImpl, d: Dato) : bool

```
m.nroFilas = m.nroFilas <= d.f ? m.nroFilas : d.f
m.nroColumnas = m.nroColumnas <= d.c ? m.nroColumnas : d.c
```

proc insertarEnOrden (inout m: MatrizFinitaImpl, in f: Fila, in c: Columna, in valor: int) : MatrizFinitaImpl

```
var nuevoDato: Dato
    nuevoDato:= new Dato(f=f, c=c, valor=valor) // Theta(1)

var j: int                        // Theta(1)
    j:= m.datos.longitud() - 1 // Theta(1)

while (j > 0 && m.esMasPrioritario(m.datos.obtener(j), nuevoDato)) do // Peor caso: Theta(n)
  m.datos.modificarPosicion(j, m.datos.obtener(j))                   // Theta(1)
  j:= j - 1                                                          // Theta(1)
endwhile

if j == -1 then
  m.datos.agregarAtras(nuevoDato)                    // Theta(n)
  m.actualizarNumeros(m, nuevoDato)
  // Peor caso: O(n) dónde n es la cota mas ajustada -> Theta(n).
else
  m.datos.modificarPosicion(j, nuevoDato) // Theta(1)
endif
/**
 * Complejidad final
 * Peor caso: Theta(n)
 */
```

proc Crear (in f: Fila, in c: Columna) : MatrizFinitaImpl

```
res.datos:= vectorVacío<>(0)
res.nroFilas:= 0
res.nroColumnas:= 0
return res
/**
 * Complejidad final
 * O(f * c)
 */
```

proc #Filas (in m: MatrizFinitaImpl, in f: Fila, in c: Columna) : bool

```
return m.nroFilas
/**
 * Complejidad final
 * O(1)
 */
```

proc #Columna (in m: MatrizFinitaImpl, in f: Fila, in c: Columna) : bool

```
return m.nroColumnas
/**
 * Complejidad final
 * O(1)
 */
```

proc Definir (inout m: MatrizFinitaImpl, in f: Fila, in c: Columna, in v: bool)

```
m.insertarEnOrden(m, f, c, v)
/**
 * Complejidad final
 * Theta(n)
 */
```

proc Obtener (inout m: MatrizFinitaImpl, in f: Fila, in c: Columna) : int

```
var v: int
var j: int
    j:= 0

while (j < m.datos.longitud()) do // Búsqueda Lineal en peor caso:  Theta(n)
  if m.datos.obtener(j).f == f && m.datos.obtener(j).c == c then // Theta(1)
    return m.datos.obtener(j).valor                              // Theta(1)
  endif

  j:= j + 1
endwhile

// Devolver valro default: 0.
return 0 // Theta(1)
/**
 * Complejidad final
 * Theta(n)
 */
```

```
% No estoy del todo de acuerdo de acceder directamente al attr interno "datos"de m2 %
% Pero no hay un TAD definido MatrizFinita donde se pueda acceder a un elemento %
% por un índice determinado (j1). %
% Lo mas coherente sería que el TAD MatrizFinita esté bien definido y que se pueda %
% obtener un Dato de una determinada posición. %
```

proc SumarMatrices (inout m1: MatrizFinitaImpl, in m2: MatrizFinitaImpl)

```
var datosNuevos: Vector<Dato>
var j1: int
var j2: int
    datosNuevos:= vectorVacío<>()       // Theta(1)
    j1:= 0                              // Theta(1)
    j2:= 0                              // Theta(1)

while (j1 < m1.datos.longitud() && j2 < m2.datos.longitud()) do // Theta(min(m, n))
  if m1.esMasPrioritario(m1.datos.obtener(j1), m2.datos.obtener(j2)) then
    datosNuevos.agregarAtras(m1.datos.obtener(j1))
    j1:= j1 + 1
  else
    datosNuevos.agregarAtras(m2.datos.obtener(j2))
    j2:= j2 + 1
  endif
endwhile

while (j1 < m1.datos.longitud()) do                 // Theta(n)
  datosNuevos.agregarAtras(m1.datos.obtener(j1))    // Theta(n)
  j1:= j1 + 1
endwhile

while (j2 < m2.datos.longitud()) do                 // Theta(m)
  datosNuevos.agregarAtras(m2.datos.obtener(j2))    // Theta(m)
  j2:= j2 + 1
endwhile

m1.datos = datosNuevos

var j: int
    j:= 0

while (j < datosNuevos.longitud()) do           // Theta(n + m)
  actualizarNumeros(m1, datosNuevos.obtener(j)) // Theta(1)
endwhile
/**
 * Complejidad final
 * Theta(n + m)
 */
```

}