Soluciones de Ejercicios Seleccionados de la Guía 3

Algoritmos y Estructuras de Datos II, DC, UBA.

Segundo cuatrimestre 2020

Índice

1. Elección de Estructuras

1.1. Ejercicio 1: matriz finita

La estructura de representación elegida es la siguiente:

MatrizFinita **se representa con** estr

donde estr es tupla ⟨ *ancho*: nat,
alto: nat,
celdasNoNulas: lista(tupla⟨ *fila*: nat, *columna*: nat, *valor*: nat ⟩)

La idea es que *ancho* y *alto* tienen las dimensiones de la matriz, y en *celdasNoNulas* están todas las posiciones de las matriz distintas de 0. Toda posición que no esté en celdasNoNulas se considera que vale 0.

celdasNoNulas deberá respetar algunos invariantes:

- Las filas y columnas de cada elemento deberá estar dentro de las dimensiones de la matriz
- No puede haber dos elementos con la misma fila y columna (porque no tiene sentido que en una misma posición de la matriz haya más de un número)
- Además, la lista deberá estar **ordenada** primero por fila y luego por columna. Esto nos permitirá más adelante garantizar la complejidad de *SumarMatrices*.

La idea detrás de esta representación es garantizar la complejidad de *SumarMatrices*. Observar que como no guardamos explícitamente las posiciones nulas, si queremos crear una nueva matriz con X elementos no nulos, eso lo podríamos hacer en $\Theta(X)$.

Los algoritmos son los siguientes:

iCrear(**in** *ancho*: `nat`, **in** *alto*: `nat`) → *res* : matriz
// Asumimos que Crear devuelve una matriz llena de ceros
1: $res \leftarrow \langle ancho, alto, Vacia() \rangle$

iFilas(**in** m: `matriz`) → *res* : nat
1: $res \leftarrow m.alto$

iColumnas(**in** m: `matriz`) → *res* : nat
1: $res \leftarrow m.ancho$

iDefinir(**in/out** m : `matriz`, **in** i : `nat`, **in** j : `nat`, **in** $nuevoValor$: `nat`)

```
1: it ← CrearIt(m.celdasNoNulas)

   // Nos salteamos todas las celdas que están en una posición 'anterior' a la posición que queremos definir.
2: while HaySiguiente(it) ∧ (i < Siguiente(it).fila ∨ (i = Siguiente(it).fila ∧ j < Siguiente(it).columna)) do
3:     Avanzar(it)
4: end while

5: if HaySiguiente(it) ∧ i = Siguiente(it).fila ∧ j = Siguiente(it).columna then
   // Si el siguiente elemento es el mismo que el queremos definir, actualizamos su valor
6:     Siguiente(it).valor = nuevoValor
7: else
   // Si no lo es, como la lista está ordenada, seguro que la posición que queremos definir no está en la lista.
   // Por ende, la insertamos de tal forma que la lista siga estando ordenada.
8:     AgregarComoSiguiente(it, ⟨i, j, nuevoValor⟩)
9: end if
```

Complejidad: En el primer ciclo, en el peor caso recorremos todo *celdasNoNulas*, que contiene las posiciones no nulas de la matriz. Por ende, la complejidad del ciclo es $\Theta(n)$ (donde n es la cantidad de elementos no nulos en la matriz pasada por parámetro). Todo lo demás son copias y comparaciones de nats, ifs, y manipulaciones de iteradores fuera de ciclos, lo que es todo $\Theta(1)$. Por ende, el algoritmo es $\Theta(n)$.

iObtener(**in** m : `matriz`, **in** i : `nat`, **in** j : `nat`) $\rightarrow$ res : nat

```
1: res ← 0
2: it ← CrearIt(m.celdasNoNulas)
3: while HaySiguiente(it) do
4:     if i = Siguiente(it).fila ∧ j = Siguiente(it).columna then
5:         res ← Siguiente(it).valor
6:     end if
7:     Avanzar(it)
8: end while
```

Complejidad: Las operaciones dentro del ciclo son $\Theta(1)$. Al igual que antes, el ciclo recorre todo *celdasNoNulas*, con lo cual el ciclo entero, y por ende el algoritmo entero, queda $\Theta(n)$.

iSumarMatrices(**in** A : `matriz`, **in** B : `matriz`) $\rightarrow$ res : matriz

```
    // Vamos a recorrer ambas listas al mismo tiempo. La idea de tenerlas ordenadas es ir insertando celdas
    // en la matriz suma de posiciones anteriores a posiciones posteriores. Esto nos va a permitir darnos
    // cuenta eficientemente qué posiciones están en ambas matrices.
 1: nuevasCeldas ← Vacia()
 2: itA ← CrearIt(A.celdasNoNulas)
 3: itB ← CrearIt(B.celdasNoNulas)

 4: while HaySiguiente(itA) ∧ HaySiguiente(itB) do
 5:     celdaA = Siguiente(itA)
 6:     celdaB = Siguiente(itB)
 7:     if celdaA.fila = celdaB.fila ∧ celdaA.columna = celdaB.columna then
            // Si las posiciones coinciden, sumamos los valores
 8:         AgregarAtras(nuevasCeldas, ⟨celdaA.fila, celdaA.columna, celdaA.valor + celdaB.valor⟩)
 9:         Avanzar(itA)
10:         Avanzar(itB)
11:     else if celdaA.fila < celdaB.fila ∨ (celdaA.fila = celdaB.fila ∧ celdaA.columna < celdaB.columna) then
            // Si la primer posición es anterior, la insertamos
12:         AgregarAtras(nuevasCeldas, celdaA)
13:         Avanzar(itA)
14:     else
            // Si la segunda posición es anterior, la insertamos
15:         AgregarAtras(nuevasCeldas, celdaB)
16:         Avanzar(itB)
17:     end if
18: end while

    // Si alguna lista nos quedó vacia, recorremos la otra e insertamos los elementos restantes
    // En vez de preguntar cual quedó vacia, directamente pedimos el siguiente a ambos iteradores
19: while HaySiguiente(itA) do
20:     AgregarAtras(nuevasCeldas, Siguiente(itA))
21:     Avanzar(itA)
22: end while
23: while HaySiguiente(itB) do
24:     AgregarAtras(nuevasCeldas, Siguiente(itB))
25:     Avanzar(itB)
26: end while

    // Finalmente, una vez que calculamos las nuevas celdas, devolvemos la matriz suma
27: res ← ⟨A.ancho, A.alto, nuevasCeldas⟩
```

Complejidad: Dentro de los ciclos las operaciones son todas $\Theta(1)$. Quizás lo menos trivial que sea $\Theta(1)$ es el *AgregarAtras* de Lista Enlazada. Puede verse en el apunte de módulos básicos que la complejidad de dicha operación es la complejidad de copiar un elemento de la lista. Como los elementos de nuestra lista son tuplas de 3 nats, copiarlos es $\Theta(1)$, y por ende agregarlos a la lista es también $\Theta(1)$.

La pregunta que queda entonces es cuantas iteraciones realizan los ciclos. Observar que en el primer ciclo, avanzamos al menos uno de los dos iteradores en cada iteración, y cortamos cuando un iterador se queda sin elementos. Luego, dos ciclos avanzan ambos iteradores asegurandosé de que hayamos recorrido ambas listas por completo. Aquí también se cumple que los iteradores avanzan en cada iteración. Por ende, el total de iteraciones entre los 3 ciclos será menor o igual a la cantidad de elementos de $A.celdasNoNulas$ y $B.celdasNoNulas$, que es igual a la cantidad de elementos no nulos de A y de B. En otras palabras, $\mathcal{O}(n+m)$.

El peor caso es aquél en el que A y B no comparten posiciones no nulas, ya que esto implica que nunca se ejecuta la rama del if del primer ciclo que avanza ambos iteradores al mismo tiempo. En ese caso, la cantidad de iteraciones entre los 3 ciclos es exactamente $n+m$, y por ende el algoritmo es $\Theta(n+m)$.

1.2. Ejercicio 2: sistema de estadísticas

Vamos a concentrarnos primero en lograr la complejidad de $O(1)$ pedida para la operación *cantPersonas*.
Lo primero que podríamos pensar es en guardar un arreglo dimensionable $\mathbf{X} = [x_1 \dots x_n]$ tal que x_i sea la cantidad de ingresados en el dia i. Si esta fuese la estructura, para calcular la cantidad de ingresados entre dos días a, b el calculo necesario sería

$$\sum_{i=a}^{b} x_i$$

Que claramente con la estructura planteada no sería constante respecto a la cantidad de días, sino que depende del tamaño del intervalo. Esto ultimo nos puede llevar a pensar en una opción en donde cada vez que se ingresa un nuevo día, precalcular las diferencias de intervalos. Si se hace esto el costo claramente no estaría al momento de preguntarnos por la diferencia de ingresantes sino al pasar de día. Una posibilidad para guardar estos calculos y luego accederlos en tiempo constante sería utilizar una matriz $\mathbf{A}$ de manera que $a_{i,j}$ corresponda a la cantidad de ingresantes en el intervalo de días (i, j) con $i \leq j$.
La estructura planteada entonces sería de la forma:

`estadísticas` se representa con `est`

donde `est` es `tupla`(*ingresantesPorIntervalo*: `Vector(Vector(Nat))`)

Pensemos entonces como implementar las operaciones del módulo:

- Comenzar: es crear un vector vacío y asignarlo a *ingresantesPorIntervalo*
- cantPersonas(i,j): por la abstracción de la estructura, sería accederla en la posición i, j de la matriz y por lo tanto es un doble acceso a vectores que sabemos que es $O(1)$ cada uno, cumpliendo con el requerimiento temporal pedido.
- TerminaDía(n): Hace falta extender la matriz en una fila y columna con los calculos de intervalos entre el nuevo día y todos los anteriores. Sabemos que en la diagonal de la matriz se encuentran la cantidad de ingresados para un dia, luego la cantidad de personas entre el nuevo dia y un dia anterior i es la suma de los elementos de la diagonal desde i hasta el nuevo día. Es evidente entonces que esta nueva fila (y columna) se computa en $O(k)$ con k la cantidad de días que pasaron.

Esta representación movió el costo lineal sobre la cantidad de dias pasados a la operación de terminar el día. Como resultado de la estructura elegida el costo en memoria es $O(n^2)$, con n la cantidad de días pasados. Se podría argumentar que como la matriz resultante es simétrica ($A_{i,j} = A_{j,i}$) no hace falta guardar toda la matriz, sino de la diagonal para abajo, pero esto no cambia en que la cantidad de elementos es $O(n^2)$. Cómo podemos entonces lograr una complejidad espacial lineal y seguir cumpliendo con el requerimiento temporal pedido?
La clave se encuentra en reconocer que las sumatorias que aparecían con las estructuras anteriores son lo que verdaderamente necesitamos guardar. Una manera de pensar la cantidad de personas que ingresaron en un intervalo de dias i, j es que estas son la cantidad total de personas que ingresaron hasta j menos las totales que ingresaron hasta el día i. Claramente cada uno de estos valores representan sumas desde el día 0 hasta el día i o j. Entonces podríamos guardar en la estructura esta información: $\mathbf{Tots} = [t_1 \dots t_n]$ con $t_k = \sum_{i=1}^{k} ingresados_i$ (con $ingresados_i$ los que ingresaron el día i). Además es facil calcular t_{k+1} a partir de t_k utilizando que: $t_{k+1} = t_k + ingresados_{k+1}$. La estructura resultante sería entonces:

`estadísticas` se representa con `est`

donde `est` es `tupla`(*ingresadosHastaDia*: `Vector(Nat)`)

Los algoritmos para esta estructura:

- Comenzar: inicializar el vector como uno vacío
- TerminaDía(n): agregar una posición al final del vector con valor: $ingresadosHastaDia.Ultimo + n$ con complejidad $O(1)$
- cantPersonas(i,j): por la abstracción de la estructura es unicaménte calcular $ingresadosHastaDia[j] - ingresadosHastaDia[i]$, que son dos accesos a vector y una resta, resultando en una complejidad temporal de $O(1)$

Además de cumplir el requerimiento temporal, esta estructura ahora ocupa $O(n)$ en memoria, mejorando la primer solución planteada.

1.3. Ejercicio 3: ránking

La clave del ejercicio está en una estructura dinámica. Los equipos van a estar en un AVL con una referencia a sus posiciones que van a estar en una lista enlazada quedando:

- DiccAVL para equipos con itLista
- Lista de Posición, Puntaje, CantEquipos

La idea de tener CantEquipos es que cuando una posición quede con CantEquipos = 0 se elimine el nodo. Fíjense que todo cierra en cada caso cuando querés actualizar todas las operaciones que podés llegar a tener están en O(1) una vez encontrado el equipo con O(log n).

De esta manera, para conocer los *puntos* o la *posición* de un equipo, lo vamos a buscar en el diccionario (con costo logarítmico) y con el iterador, vemos en qué nodo de la lista está (que contiene la información tanto del puntaje como de la posición).

Esto suena hermoso, pero ¿cómo lo construimos? ¿Cómo hacemos *regPartido*? Para pensarlo, reflexionemos sobre qué información nueva o cambia con esta operación.

...

...

¿Ya reflexionamos? Mirando el TAD, vemos que es un generador. ¿Qué hace? Pensemos un poco más, entonces.

...

Exacto. Modifica el puntaje de un equipo. De este modo, al registrar un nuevo partido lo que cambia para empezar es la cantidad de puntos del equipo ganador. Ahora bien, esto también puede producir un cambio en la posición de ese equipo pero también en la de otros. Si había dos punteros en el torneo, si uno de ellos gana, ese sigue siendo primero pero el otro pasa a ser segundo.

Entonces, yendo a la estructura en sí, si tenemos una lista ordenada por posición donde en cada nodo nos guardamos la cantidad de equipos que están en esa posición y su puntaje (el mismo para cada equipo, por lo que solo guardamos un Nat), lo que hace *regPartido* es buscar al equipo ganador en el diccionario, ir al nodo asociado, actualizar los valores del nodo (si es necesario, eliminarlo) y pasarlo al nodo asociado al siguiente puntaje (si hace falta, crearlo).

Ahora sí pasemos a la formalidad:

donde `estr` es `tupla`(

TORNEO **se representa con** estr

donde estr es tupla ⟨ *equipos*: dicc(equipo, itLista),
posiciones: lista(tupla⟨ *posicion*: nat, *puntaje*: nat, *cantEquipos*: nat ⟩)

Explicación de la estructura

Ya lo estuvimos haciendo pero es mucho muy importante indicar en esta sección el por qué de la elección de dicha estructura y las complejidades de cada operación.

Justificación de la complejidad pedida

Para justificar la complejidad de las operaciones pedidas se puede explicar en castellano el por qué se cumplen estas complejidades (si es que no se pide escribir el código) o se lo puede escribir directamente.

En este caso, nos piden los pseudocódigos de dichas operaciones así que matamos dos pájaros de un tiro.

---

iPuntos(**in** t : `torneo`, **in** e : `equipo`) → res : nat

1: $itPosiciones \leftarrow obtener(equipo, t.equipos)$
2: $res \leftarrow Siguiente(itPosiciones).puntaje$

Complejidad: Básicamente tenemos dos funciones: una de costo logarítmico porque la búsqueda y obtención de un elemento en un AVL es $O(log(n))$ (con n el tamaño del árbol) y la otra es pedirle el elemento al iterador, o sea, una operación constante. En síntesis, es $O(log(n))$.

---

iPos(**in** t : `torneo`, **in** e : `equipo`) $\rightarrow$ res : nat

1: $itPosiciones \leftarrow obtener(equipo, t.equipos)$
2: $res \leftarrow Siguiente(itPosiciones).posicion$

Complejidad: Análogamente, es $O(log(n))$.

---

iRegPartido(**in** t : `torneo`, **in** e : `equipo`)

1: $itPosiciones \leftarrow obtener(equipo, t.equipos)$
2: $nodoActual \leftarrow Siguiente(itPosiciones)$
   //Actualicemos el nodo viejo
3: **if** $nodoActual.cantEquipos == 1$ **then**
   //Si no quedaban equipos en esa posición, lo eliminamos
4: $EliminarSiguiente(itPosiciones)$
5: **else**
6: $++nodoActual.posicion$
7: $--nodoActual.cantEquipos$
8: **end if**
   //Actualizamos o creamos la posición nueva
9: **if** $HayAnterior(itPosiciones) \wedge_{\text{L}} Anterior(itPosiciones).puntaje == nodoActual.puntaje + 1$ **then**
   //Si al sumar el punto va a estar empatado en puntaje con otros equipos
10: $++nodoAnterior.cantEquipos$
11: $definir(equipo, Retroceder(itAnterior), t.equipos)$
12: **else**
   //Vamos a tener que crear un nuevo nodo asociado a la nueva posición del equipo
13: $nuevoNodo \leftarrow \langle nodoActual, nodoActual.puntaje + 1, 1 \rangle$
14: $AgregarComoAnterior(itPosiciones, nuevoNodo)$
15: **end if**

Complejidad: Esta función parece más difícil pero no lo es tanto. Nuevamente, la búsqueda y obtención de un elemento en un árbol tiene costo logarítmico. Pero el resto de las operaciones son $O(1)$ (se podría hacer un análisis más detallado pero es trivial) excepto *definir* que como tiene costo logarítmico, no nos hace a la cuestión. Finalmente, la complejidad total es $O(log(n))$.

Aclaración: *definir* se podría haber evitado manteniendo un iterador al nodo del AVL. Queda de tarea hacer esa modificación para que sea más eficiente.

1.4. Ejercicio 4: sistema de transporte

Recordemos las complejidades requeridas:

- DarDeAlta / DarDeBaja Universidad: $O(\log U)$
- DarDeAlta / DarDeBaja Línea: $O(\log l)$
- ConsultarLíneasPorUniversidad: $O(\log U)$
- ConsultarUniversidadesPorLínea: $O(\log l + Ml)$ (aunque se podría en $O(\log l)$)

Donde l = número de línea, U = cantidad de universidades, Ml = Universidades por las que pasa la línea. Vamos intentando resolver el ejercicio por lo básico.
En primer lugar queremos tener líneas por universidad en $O(\log U)$, y universidades por número línea, por lo que empecemos planteando lo siguiente:

`sistemaTransporte` **se representa con** `estr`

donde `estr` es `tupla`(*Universidades*: `DiccAVL<Universidad, conjLineal<Línea> >`
, *Líneas*: `DiccTrie<Línea, conjLineal<Universidad> >`)

Ambas consultas se pueden satisfacer, ¿pero qué ocurre con el alta?
La verdad es que cuando uno da de alta una universidad tiene que a ver que líneas pasa, y eso es O(#Líneas), por otro lado cuando uno da de alta una línea tiene que ir a ver que universidades pasa, y eso es O(#Universiades). Se nos va el costo. ¿Qué dato clave tenemos? **la ciudad está acotada, sus límites son fijos y no cambian**.

Entonces, extraigamos la información de las ubicaciones en una grilla, y reemplacemos la info que teníamos en nuestros diccionarios por las posiciones i, j de la matriz que son las *celdas* de la ciudad:

`sistemaTransporte` **se representa con** `estr`

donde `estr` es `tupla`(*Universidades*: `DiccAVL<Universidad, < Dirección : <i : Nat, j : Nat> >`
, *Líneas*: `DiccTrie<Línea, <Recorrido : secu<<i : Nat, j : Nat>> > >`
, *Matriz*: `tupla de < Universidades : conjAVL<Universidad>`
`Líneas : conjTrie<Línea> > )`

Ahora todo cierra porque si tenemos que dar de alta una universidad la damos de alta en el diccionario y en su celda (a la cual se accede en O(1)) todo en $O(\log U)$ mientras que dar de alta una línea sería sobre el diccTrie y el conjTrie en $O(\log l)$.
Para eliminar una línea hay que ir a cada casilla del recorrido (acotado) y eliminarla del conjunto trie cada operación en $O(\log l)$ por lo que la complejidad total es $O(\log l)$.

Veamos una versión que utiliza iteradores:

`sistemaTransporte` **se representa con** `estr`

donde `estr` es `tupla`(*Universidades*: `DiccAVL<Universidad, < Dirección : <i : Nat, j : Nat>,`
`UbicaciónEnCelda : itLista >`
, *Líneas*: `DiccTrie<Línea, <Recorrido : secu<<i : Nat, j : Nat>>,`
`PosicionesEnCeldas : lista<itLista> > >`
, *Matriz*: `tupla de < Universidades : conjLineal<Universidad>`
`Líneas : conjLineal<Línea> > )`

Entonces en vez de tener conjuntos *pesados* en la matriz tenemos conjuntos lineales donde agregar y eliminar va a ser O(1) ya teniendo el iterador.

Para poder devolver todas las universidades por línea en $O(\log l)$ (sin copiar todas las universidades del recorrido) se podría devolver un iterador que recorra los iteradores internos de su secuencia. Por ejemplo:

IteradorPepe:
Su estructura es: secu<Celda>, dondeEstoy: itConjAVL
Su operación Siguiente():
Si dondeEstoy.HaySiguiente? –> dondeEstoy.Siguiente
Si dondeEstoy.NoHaySiguiente? –> dondeEstoy = moveteDeCelda y ActualizarIt con siguiente celda O(1)... luego *return dondeEstoy.Siguiente()*

Cada movimiento es en O(1).

1.5. Ejercicio 5: sistema de multas

Una opción puede ser estr es tupla de:

- vehículos: diccTrie(patente, <
  localidad: localidad/string,
  itLocalidad: itDiccTrie,
  multas: conjLista<multa, itAMultasDeSusCámaras, itAMultasDeSusVehículos>
  >)
- localidades: diccTrie(localidad, <
  * cámaras: conjLista<cámara>
  * vehículos: conjLista<vehículo>
  * multasDeSusCámaras: conjLista<multa>
  * multasDeSusVehículos: conjLista<multa>
  >)
- cámaras: diccAVL(cámara, <localidad: string, itLocalidad: itDiccTrie>)

Pseudocódigo asociado:

RegistrarMulta(**in** c: `cámara`, **in** p: `patente`, **in** m: `m` onto)

1: $itDicc = e.vehculos.Obtener(patente)$
2: $itLocalidadDelVehculo = itDicc.itLocalidad.MultasDeSusVehculos.Agregar(multa)$
3: $itAVL = e.cmaras.Obtener(cmara)$
4: $itLocalidadCmara = itAVL.itLocalidad.MultasDeSusCmaras.Agregar(multa)$
5: $itDicc.multas.Agregar(multa, itLocalidadCmara, itLocalidadDelVehculo)$

PagarMultas(**in** p: `patente`)

1: $itDicc = e.vehculos.Obtener(patente)$
2: **for** $menitDicc.Siguiente().multas$ **do**
3: $m.itAMultasDeSusCmaras.EliminarSiguiente()$
4: $m.itAMultasDeSusVehculos.EliminarSiguiente()$
5: **end for**
6: $it = itDicc.Siguiente().multas.obtenerIt()$
7: **while** $HaySiguiente(it)$ **do**
8: $EliminarSiguiente(it)$
9: **end while**

Idea:

- Los autos tienen las multas, y las localidades también, el tema es que cuando querés pagar las multas tenés que acceder y eliminarlas en O(1) de las localidades que tienen la info, por eso te guardás el puntero.
- Por otro lado tanto como por vehículo como por cámara hay que guardarse un iterador a la localidad para no tener que buscarla