# Algoritmos y Estructuras de Datos
## Segundo Parcial – Viernes 28 de Junio de 2024

| LU | Apellido y Nombre | #Hojas | E1 | E2 | E3 | E4 | Nota | Corrigió |
|---|---|---|---|---|---|---|---|---|
| 1257/0 | Kruel Magali | 5 | 13,3 | 28,5 | 20 | 30 | 92 | Belén |

- Es posible tener una hoja (2 carillas), escrita a mano, con los anotaciones que se deseen, más los apuntes del campus
- Cada ejercicio debe entregarse en **hojas separadas**
- Incluir en cada hoja nombre y apellido, número de libreta y número de hoja
- El parcial se aprueba con 70 puntos

## E1. Complejidad [20 pts]

Responder, justificando adecuadamente su respuesta:

a) Decimos que la complejidad de insertar en una lista enlazada ordenada es $O(n)$ en el peor caso, pero eso sólo es cierto si podemos comparar dos elementos en $O(1)$ y lo mismo para la operación de copia. Calcule la complejidad de insertar en una lista enlazada ordenada *sin aliasing* (es decir, copiando en la lista el elemento a insertar y no haciendo una referencia al mismo) si comparar es $O(\log n)$ y copiar es $O(n)$.

b) Si la complejidad en el peor caso de un algoritmo es $\Omega(n)$, ¿es verdad que la complejidad de mejor caso no puede ser $O(1)$?

c) Indique si la siguiente expresión es verdadera o falsa y justifique: Si $O(f(n)) \cap \Omega(g(n)) = \emptyset$, entonces $O(g(n)) \cap \Omega(f(n)) = \emptyset$.

## E2. Elección de estructuras e invariante de representación [30 pts]

Una *relación parcial* es una correspondencia entre dos conjuntos en la cual a cada elemento del primer conjunto le corresponde cero o más elementos del segundo conjunto. Especificamos una relación parcial con el siguiente TAD:

```
TAD Relacion<T1,T2> {
    obs elems: conj<tupla<T1, T2>>

    proc relacionar(inout r: Relacion<T1, T2>, in t1: T1, in t2: T2)
        requiere {r = R_0}
        asegura {r = R_0 ∪ {< t1, t2 >}}

    proc relacionadosDer(in r: Relacion<T1, T2>, in t1: T1): Conjunto<T2>
        asegura {(∀ t2 : T2)(< t1, t2> ∈ r.elems ↔ t2 ∈ res)}

    proc relacionadosIzq(in r: Relacion<T1, T2>, in t2: T2): Conjunto<T1>
        asegura {(∀ t1 : T1)(< t1, t2> ∈ r.elems ↔ t1 ∈ res)}
}
```

Queremos las siguientes complejidades para las operaciones:

- relacionar debe ser $O(\log n + \log m)$.
- relacionadosIzq debe ser $O(\log m)$.
- relacionadosDer debe ser $O(\log n)$.

donde $n$ es la cantidad de elementos de $T1$ que tienen al menos un elemento de $T2$ relacionado, y $m$ es la cantidad de elementos de $T2$ que tienen al menos un elemento de $T1$ relacionado.

a) Proponga una estructura para implementar este TAD cumpliendo esas complejidades.

b) Se quiere agregar la operación `másRelacionadoIzq` que dada una relación devuelve el `T1` que tiene más `T2` relacionados (en caso de haber más de un `T1` con la cantidad máxima de relacionados, se puede devolver cualquiera de ellos). Dicha operación debe tener complejidad $O(1)$. Explique cómo modificaría la estructura propuesta en el punto anterior para también resolver este nuevo problema.

c) Escribir en castellano pero en forma precisa el invariante de representación de la estructura (incluyendo los cambios realizados en el punto b)

d) Escribir en castellano, también en forma precisa, la función de abstracción para dicha estructura.

## E3. Estructuras [20 pts]

¿Qué estructura elegiría para implementar cada uno de los siguientes conjuntos? Justifique su respuesta indicando qué ventajas tiene la estructura que eligió.

a) un conjunto de a lo sumo 100 números de 64 bits (entre 1 y 18446744073709551615)
b) un conjunto de mediciones en el que se agregan nuevas mediciones muy frecuentemente y se consulta muy poco
c) un conjunto de palabras sobre un alfabeto no acotado en el que se inserta poco y se consulta mucho

## E4. Sorting [30 pts]

Dado un arreglo de pares <estudiante, nota>, queremos devolver un arreglo de pares <estudiante, promedio>, ordenado en forma descendente por promedio y, en caso de empate, por número de libreta.

Los estudiantes se representan con un número de libreta (puede considerar que es un número de a lo sumo 10 dígitos), y la nota es un número entre 1 y 100. La cantidad de estudiantes se considera no acotada.

Se pide dar un algoritmo que resuelva el problema con complejidad $O(n + m \log m)$, donde $n$ es la cantidad total de notas (es decir, el tamaño del arreglo de entrada), y $m$ es la cantidad total de estudiantes.

Ejemplo:

```
Entrada: <12395, 72>, <45615, 81>, <12395, 94>, <45615, 100>, <78920, 81>,
         <12395, 90>, <45615, 71>, <78920, 50>
Salida: <12395, 85.3333>, <45615, 84>, <78920, 65.5>
```

a) Describa cada etapa del algoritmo con sus palabras, y justifique por qué cumple con las complejidades pedidas.
b) Escriba el algoritmo en pseudocódigo. Puede utilizar (sin reescribir) todos los algoritmos y estructuras de datos vistos en clase.

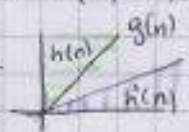
Magali Kruel LU 1257/23 Hoja Nº 1

**E1.**

a) Primero tengo que hacer una copia del elemento que me pasan, que lleva $O(n)$. Luego recorro con un ciclo donde comparo el valor del nodo actual con el que voy a insertar para asegurarme que la lista siga ordenada luego de la inserción. Esa comparación lleva $O(\log n)$ por enunciado, y en el peor caso (tener que insertar al final), tendré que recorrer el ciclo "n" veces; luego la complejidad de este paso es $O(n.\log n)$. Por último tengo que insertar el elemento, donde antes debo realizar algunas comparaciones ($O(\log n)$). La complejidad final será:

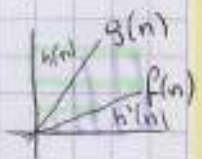
$$O(n) + O(n.\log n) + O(\log n) = O(\max\{n\log n, n, \log n\}) = O(n.\log n)$$

c) Veamos que si $O(f(n)) \cap \Omega(g(n)) = \emptyset$, de alguna forma $g(n) > f(n)$, pues $\Omega(g(n))$ ~~implica que~~ tiene en su conjunto a toda función $h(n)$ tal que $c * g(n) < h(n)$ para algún $c \in \mathbb{R}$ y $O(f(n))$ tiene en su conjunto a toda función $h'(n)$ tal que $h'(n) < c . f(n)$ para algún $c \in \mathbb{R}$, y $h(n) \cap h'(n) = \emptyset$. Gráficamente queda expresado de la siguiente manera:



$\Rightarrow h \cap h' = \emptyset$

Ahora, si vemos $O(g(n)) \cap \Omega(f(n))$, tenemos que $c.f(n) < h(n)$ y que $h'(n) < c. g(n)$, pero al ser $g(n) > f(n)$, la intersección entre $h$ y $h'$ no es nula. Es decir, $O(g(n)) \cap \Omega(f(n)) \neq \emptyset$, por lo que la expresión es FALSA.



b) ~~Incorrecto~~

Es verdad. En un mejor caso, $\Omega$ será mayor, por lo que $O$ no puede ser menor que n nunca ✗

Confundis mejor y peor caso con $\Omega$ y $O$

E2-

a) Modulo relacionImpl implementa Relacion<T1, T2>

var cT1 : DiccionarioLogaritmico<T1, ConjuntoLog<T2>> ✓

var cT2 : DiccionarioLogaritmico<T2, ConjuntoLog<T1>> ✓

Esta estructura cumple la complejidad de:

- relacionar(), pues insertar en DictLog o en ConjLog es O(log n) para T1 y O(log m) para T2, y chequear si pertenece o no un elemento también O(log n) y O(log m) respectivamente
- relacionadosDer() obtiene el valor de T1 en O(log n) y lo devuelve
- relacionadosIzq() misma lógica que relacionadosDer : O(log m) ✓

b) agregaría una variable "var maxT1: <T1, tamaño>" que guarde el T1 que tenga más T2 relacionados. Para esto, cambio relacionar() para que, una vez agregado T2 al ConjLog de T1 en cT1, me compare el tamaño del ConjLog (O(1) por modulos basicos) con maxT1.tamaño, y si es mayor, cambiar maxT1 (O(1) porque solo requiere setear cosas ya obtenidas previamente). Esto no me modifica la complejidad del punto a) porque todas mis operaciones se realizan en O(1).

La operación masRelacionadoIzq solo deberá retornar maxT1.T1, que se hace en O(1). ✓

c) InvRep:

Todo t1 perteneciente a cT1 debe aparecer 1 sola vez dentro de cada conjLog de cada t2 ∈ cT2 tal que t2 pertenezca al conjunto de ese t1. Mismo razonamiento para todo t2 ∈ cT2.
El tamaño del conjLog de cada t1 ∈ cT1 debe ser un entero entre los valores 1 hasta tamaño(cT2). El tamaño del ConjLog de cada t2 ∈ cT2 debe ser un entero entre 1 hasta tamaño(cT1). La variable máxT1.T1 debe pertenecer a cT1. y máx.T1.tamaño es igual a tamaño(obtener(cT1, maxT1.T1)).
No hay repetidos en ningun conjLog de ningún t1 ∈ cT1 ni ningún t2 ∈ cT2. ✓

no hace falta

d) Para cada tupla<T1,T2> perteneciente a Relación.elems, T1 pertenece a cT1 y T2 pertenece al conjLog de ese T1, y T2 pertenece a cT2 y T1 pertenece al conjLog de ese T2. ✓

En relaciónImpl, máxT1.T1 es igual al T1 con mayor apariciones de relacion.elems, y maxT1.tamaño es igual a la cantidad de esas apariciones, es decir la sumatoria de 1 por cada aparición de T1 en relación.elems. La cantidad de apariciones de t1 en relación.elems es igual al tamaño del conjLog en t1 en cT1 (mismo razonamiento para cada t2 ∈ relación.elems con t2 ∈ cT2)

ya lo puse en el InvRep. es redundante

Magali Kruel | LU: 1257/23 | Hoja nro: 3

E3.

a) Elegiría un Array (vector), ya que al crearlo le pongo tamaño 100, agregar un elemento costaría O(1) teniendo un observador "tamaño" ya que nunca tendría que "agrandar" el array y obtener una posición tambien es O(1). Recorrer el array es O(100) en el peor caso ⇒ O(1). ✓

b) Elegiría una lista enlazada ya que son muy eficientes para agregar elementos (tardan O(1)), pero no muy eficientes para consultar ya que requiere de costo lineal (O(n)), pero no es un obstaculo ya que este conjunto será poco consultado. ✓
ojo con los repetidos

c) Elegiría AVL (o conjuntoLog), pues es la estructura con menor costo para esas operaciones. No utilizo Trie ya que solo es eficiente en abecedarios finitos, y este no es el caso. ✓

Magali Kruel | LU: 1257/23 | Hoja Nº 4

## E4.

2) proc algoritmoOrdenarEstudiantes(in info: Array<<estudiante, nota>>) : Array<<estudiante, promedio>>

O(1) → var estudiantes : DiccionarioDigital<estudiante, <cNotas, suma>>

O(n) → Agrego mediante un for cada estudiante ∈ info a mi variable estudiantes. Si el estudiante no está (O(|e|)), lo agrego a estudiantes (O(|e|)) junto con la tupla <1, estudiante.nota> (O(1)). Si ya estaba (O(|e|)), sumo 1 a la variable cNotas y a la segunda variable de la tupla (suma) le sumo estudiante.nota (O(1)). Este paso tiene complejidad O(n.O(|e|+|e|+1)) siendo |e| = logitud del ~~DU~~ nro. de libreta del estudiante. Como el nro. de libreta esta acotado, las operaciones O(|e|) pasan a ser O(1), por lo que la complejidad final de este paso es O(n)

O(m) → var res : new Array<<estudiante, promedio>>(estudiantes.tamaño())

O(m) → Utilizo el iterador proporcionado en el apunte de Modulos Básicos para ir agregando cada estudiante ∈ estudiantes a mi variable res (O(m), pues inicie el tamaño de res en m para asegurarme de no tener que "agrandarlo"); donde agrego la tupla <estudiante, suma/cNotas>, ya que ese será el promedio del estudiante (crear esa tupla "cuesta" O(1)). El iterador me asegura que los estudiantes se agregaron en orden según su nro. de libreta.

O(m log m) → Por último, hago mergeSort sobre res respecto de "promedio", que me los ordenará de mayor a menor, dejandolo ordenado según libreta en caso de empate, pues mergeSort es estable.

$O(1)$ → Por último devuelvo res.

Habiendose justificado las complejidades, individuales, la complejidad final será $O(1+n+m+m+m.\log m) = O(n + m.\log m)$ ✓

b) "estudiante" es un número de libreta acotado. "nota" es un número del 1 al 100. "cNota" es un natural que referencia a la cantidad de notas de un estudiante. "suma" es un número que referencia la suma de todas las notas de un estudiante. "Promedio" es un número real.

```
proc ordenarEstudiantes(in info: Array<<estudiante, nota>>) : Array<<estudiante, promedio>>
  Var estudiantes : DiccDig<estudiante, <cNota, suma>>
  for i = 0... info.length do
    if !esta(estudiantes, info[i].estudiante) then
      Var tup := <1, info[i].nota>
      definir(estudiantes, <info[i].estudiante, tup>)
    else
      Var est := obtener(estudiantes, info[i].estudiante)
      est.cNota := est.cNota + 1
      est.suma := est.suma + info[i].nota
    endif
  endfor

  Var res := new Array<<estudiante, promedio>>(tamaño(estudiantes))
  Var ind := 0
  Var it := new iterador(estudiantes)
  while it.haySiguiente do
    Var aAgregar := it.siguiente
    Var tup := <aAgregar.estudiante, dividir(aAgregar.suma, aAgregar.cNota)>
    res[ind] := tup
    ind := ind + 1
  endwhile
```

(Continúo en hoja nro. 5)

E4. b)

Continuacion...

```
mergeSort(res)   //sobre promedio
return res

ProcAux dividir(in n:int, in m:int):float
  return n/m
```