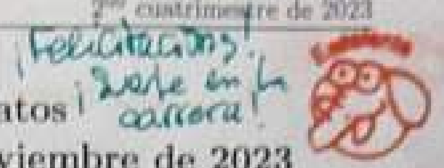


Algoritmos y Estructuras de Datos

Segundo Parcial — Sábado 25 de Noviembre de 2023

#Orden	Libreta	Apellido y Nombre	E1	E2	E3	Nota Final
[redacted]	[redacted]	[redacted]	40	30	30	100

- Es posible tener una hoja (2 carillas), escrita a mano, con las anotaciones que se deseen, además de los apuntes de la cátedra.
- Cada ejercicio debe entregarse en **hojas separadas**.
- Incluir en cada hoja el número de orden asignado, número de libreta, número de hoja, apellido y nombre.
- El parcial se aprueba con 60 puntos. Para promocionar es necesario tener al menos 70 y ningún ejercicio con 0 puntos (en ambos parciales).

E1. Elección de estructuras (40 pts)

Se quiere implementar el TAD Biblioteca que modela una biblioteca con su colección de libros. Por el momento la biblioteca cuenta con una sola estantería, dentro de la cual cada libro ocupa una posición. La biblioteca cuenta con un registro de socios que pueden retirar y devolver libros en cualquier momento. Por restricciones del sistema que se utiliza, un socio no puede registrarse con un nombre de más de 50 caracteres.

Cuando la biblioteca adquiere un nuevo libro o cuando un libro es devuelto, éste es insertado en el primer espacio libre de la estantería. Es decir, si los lugares ocupados son 1, 2, 3, 4 y se presta el libro en la posición 2, al agregar un nuevo libro al catálogo éste será ubicado en la posición 2. Cuando el libro que estaba originalmente en la posición 2 sea devuelto, será ubicado en la primera posición libre, que será la 5.

Dadas las siguientes operaciones y de acuerdo a las complejidades temporales de peor caso indicadas, donde L es la cantidad de libros en la colección, r es la cantidad de libros que el socio en cuestión tiene retirados y k la cantidad de posiciones libres en la estantería, respectivamente:

- `proc AgregarLibroAlCatálogo(inout b: Biblioteca, in l: idLibro)`
  Requiere: {`l` no pertenece a la colección de libros de `b`}
  Descripción: la biblioteca adquiere un nuevo libro, lo suma a su catálogo y lo pone en la estantería en el primer espacio disponible.
  Complejidad: $O(log(k) + log(L))$
- `proc PedirLibro(inout b: Biblioteca, in l: idLibro, in s: Socio)`
  Requiere: {`s` es socio de la biblioteca y el libro `l` no está entre los libros prestados}
  Descripción: el socio pasa a retirar un libro que se retira de la estantería y se acumula en sus libros prestados.
  Complejidad: $O(log(r) + log(k) + log(L))$
- `proc DevolverLibro(inout b: Biblioteca, in l: idLibro, in s: Socio)`
  Requiere: {`s` es socio de la biblioteca y el libro `l` está entre sus libros prestados}
  Descripción: el socio pasa a devolver un libro que previamente había tomado prestado. Vuelve a la estantería en el primer espacio disponible.
  Complejidad: $O(log(r) + log(k) + log(L))$
- `proc Prestados(in b: Biblioteca, in s: Socio): Conjunto<Libro>`
  Requiere: {`s` es socio de la biblioteca}
  Descripción: este procedimiento retorna los libros que el socio tomó prestados de la biblioteca y aún no devolvió.
  Complejidad: $O(1)$
- `proc UbicaciónDeLibro(in b: Biblioteca, in l: idLibro): Posicion`
  Requiere: {`l` pertenece a la colección de libros de `b` y no está prestado}
  Descripción: obtiene la posición del libro en la estantería.
  Complejidad: $O(log(L))$

Se pide:

a) Plantear la estructura de representación del módulo `BibliotecaImpl`, que provea las operaciones mencionadas. Se debe explicar detalladamente qué información se guarda en cada parte, las relaciones entre ellas, y cómo se aseguraría que la información registrada es consistente.

b) Justificar cómo se cumplen las complejidades pedidas con esta estructura por cada operación. Indicar suposiciones sobre la implementación de las estructuras usadas y aclaraciones sobre aliasing.

c) Escribir los algoritmos de `AgregarLibroAlCatálogo` y `DevolverLibro`, justificando **detalladamente** que se cumplen las cotas de complejidad requeridas.

CamScanner

E2. Invariante de representación y función de abstracción (30 pts)

Tenemos un TAD que modela las ventas minoristas de un comercio. Cada venta es individual (una unidad de un producto) y se quieren registrar todas las ventas. El TAD tiene un único observador:

```
TAD Comercio {
    obs ventasPorProducto: dict<Producto, seq<tupla<Fecha, Monto>>>
}

Producto es string
Monto es int
Fecha es int (segundos desde 1/1/1970)
```

ventasPorProducto contiene, para cada producto, una secuencia con todas las ventas que se hicieron de ese producto. Para cada venta, se registra la fecha y el precio. Se puede considerar que todas las fechas son diferentes. Este TAD lo vamos a implementar con la siguiente estructura:

```
Modulo ComercioImpl implementa Comercio {
    var ventas: Secuencia<tupla<Producto, Fecha, Monto>>
    var totalPorProducto: Diccionario<Producto, Monto>
    var ultimoPrecio: Diccionario<Producto, Monto>
}
```

- **ventas** es una secuencia con todas las ventas realizadas, indicando producto, fecha y monto.
- **totalPorProducto** asocia cada producto con el dinero total obtenido por todas sus ventas.
- **ultimoPrecio** asocia cada producto con el monto de su última venta registrada.

Se pide:

a) Escribir en forma coloquial y detallada el invariante de representación y la función de abstracción.
b) Escribir ambos en el lenguaje de especificación.

E3. Sorting (30 pts)

Se nos pide ayudar a un herborista que quiere poder organizar sus ingredientes para determinar qué hierbas le conviene recolectar. Para ello cuenta con su propio inventario. Como no es una persona muy organizada, puede tener distintas hierbas del mismo tipo en distintas alacenas o cofres. Luego de realizar una inspección de su lugar de trabajo, nos entrega una secuencia de n tuplas que constan de una hierba, identificada por su nombre, y la cantidad que se encontró. El nombre de cada hierba tiene como máximo 100 caracteres, de acuerdo al estándar de la Organización Mundial de Herboristas. El herborista cuenta a su vez con su libro de creaciones, que le permite saber en cuántas recetas se utiliza cada hierba.

Se necesita saber cuáles son las hierbas que se usan en más creaciones y, en caso de empate, deberían aparecer primero aquellos de las que tiene menos reservas. La complejidad esperada en el *peor caso* es de $O(n + h\ log(h))$, donde h es la cantidad de hierbas distintas con las que cuenta el herborista.

```
proc Recolectar(in s:Vector<tupla<string,int>>, in u:Diccionario<string,int>):Vector<string>
```

Ejemplo:

```
stock = [ ("Diente de León", 10), ("Menta", 4), ("Margarita", 13),
          ("Lavanda", 12), ("Diente de León", 5), ("Margarita", 6) ]

usos = {"Diente de León": 5, "Menta": 1, "Margarita": 3, "Lavanda": 5}

Recolectar(stock, usos) = ["Lavanda", "Diente de León", "Margarita", "Menta"]
```

Los primeros son la lavanda y el diente de león porque ambos tiene 5 usos, pero aparece primera la lavanda porque hay menos stock. En tercer lugar tenemos la margarita que tiene 3 usos. Finalmente, en último lugar está la menta, que tiene un solo uso.

a) Se pide escribir el algoritmo de `Recolectar`. Justificar <u>detalladamente</u> la complejidad y escribir todas las suposiciones sobre las implementaciones de las estructuras usadas, entre otras.
b) ¿Cuál sería el *mejor caso* para este problema? ¿Cuál sería la cota de complejidad más ajustada?

CamScanner

HOJA N° 1

E1

a) Modulo BibliotecaImpl implementa Biblioteca {

var coleccion: ConjLog {idLibro}

var estanteria: DiccLog {idLibro : Posicion}

var posicionesLibres: ~~ListaEnlazada~~ MinHeap <Posicion>

var socios: DiccDig {Socio : ConjLog {idLibro}}

¿ ~~var prestados: ConjLog {idLibro}~~

· ~~En colección tengo todos los libros~~

· En socios tengo a todos los socios de mi biblioteca. Cada uno asociado con el conjunto de libros que el socio tiene retirados.

· ~~En posicionesLibres tengo las posiciones libres que tengo en mi estantería. Siempre voy a trabajar con la primera lo cual me toma O(1), si la uso la elimino en O(1), si tengo nuevas posiciones las agrego al final en O(1) y pregunto la longitud de la lista en O(1).~~

· En estanteria ~~guardo~~ tengo a todos mis libros disponibles con su respectiva posición.

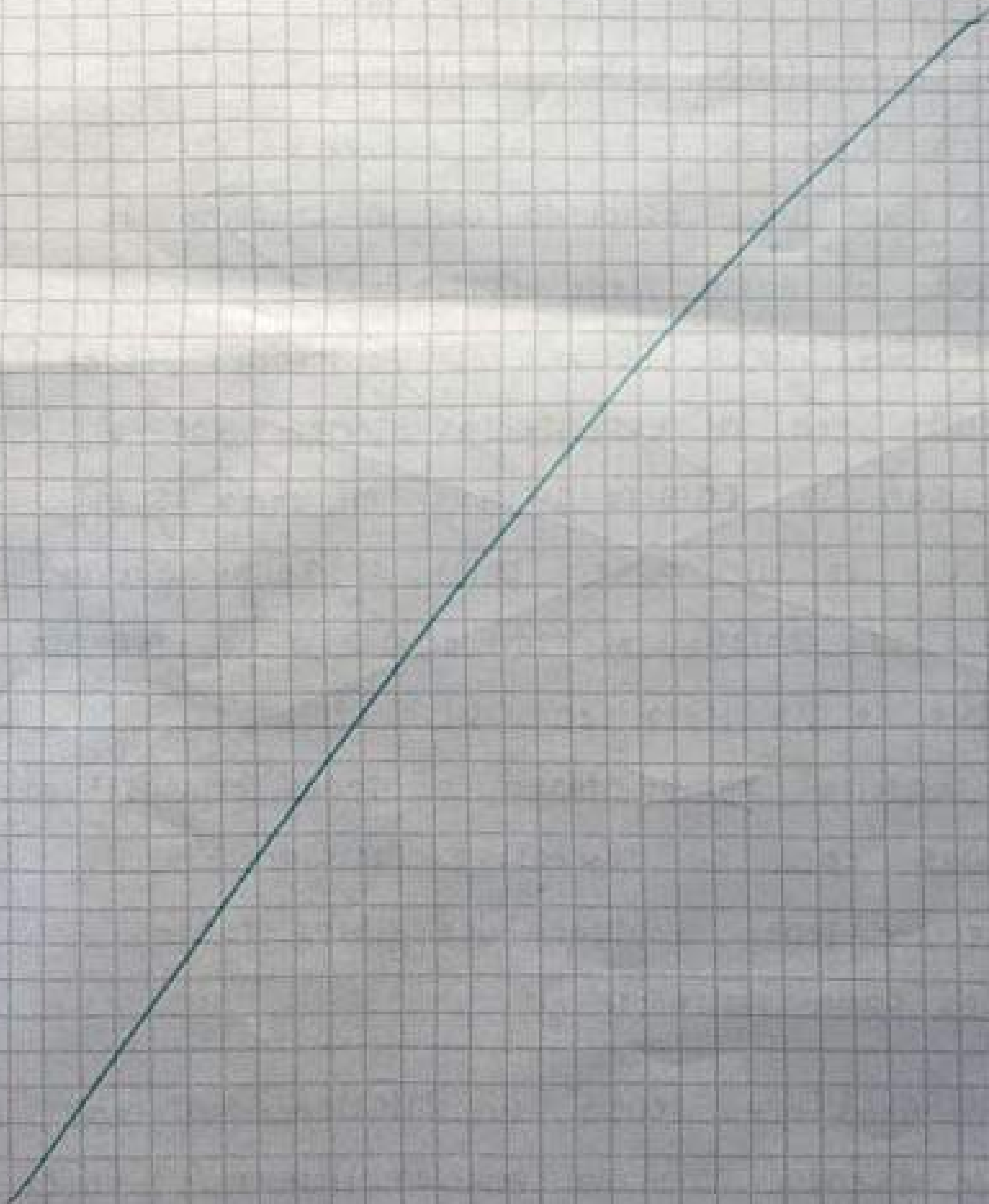
· En Coleccion tengo a todos los libros tanto disponibles como retirados.

· En posicionesLibres tengo un MinHeap<Posicion> en el que siempre tengo la posición libre más chica al principio.

NOTA

- n = |b.estanterial y como n ≤ L siempre, voy a acotar ~~sieme~~ por O(log L)
- L = |b.coleccion|
- K = |b.posiciones Libres|
- r es la longitud del conjunto asociado a un socio en particular

OK



NOTA

* Agrego el libro a colección en $O(\log L)$

HOJA N° 2

b) Voy a justificar las complejidades de cada operación y porque son consistentes.

- agregarLibroAlCatalogo

* Me pregunto si posicionesLibres. ~~longitud() != 0~~ estaVacia()

- En caso ~~afirmativo~~ negativo, significa que debo ocupar la posición libre ~~que fue dejada hace más tiempo~~ más pequeña. Accedo a ella en $O(1)$, la guardo en una variable p y ~~borro el primer elemento de mi lista en O(1)~~ la desencolo en $O(\log k)$ (ya no es libre).
Defino mi par (idLibro, p) en mi b.estanteria lo cual me toma $O(\log n) \leq O(\log L)$

- En caso negativo, no tengo posiciones libres. Debo usar una nueva posición.

p := estanteria.length() + 1 // $O(1)$

definir (b.estanteria, idLibro, p) // $O(\log n)$ A $\leq O(\log L)$

Luego, ~~ambos casos toman O(log L) y cumplo la complejidad~~ tomo el peor caso y mi complejidad queda $O(\log k + \log L + \log n) = O(\log k + \log L)$ por A

- pedirLibro

libros Retirados := b.socios[~~socio~~ s] // $O(1)$ → Referencia

Socios está implementado sobre un trie y por consigna sé que las keys están acotadas. Luego, puedo operar en $O(1)$

p := obtener(b.estanteria, idLibro) // $O(\log L)$

eliminar (b.estanteria, idLibro) // $O(\log L)$

NOTA

6. posicionesLibres. ~~agregarAtras~~ encolar (p) // ~~O(1)~~ O(log k)

librosRetirados. agregar (idLibro) // O(log r)

Complej: ~~O(log r + log L) ∈~~ O(log r + log L + log k)

- devolverLibro:

Vuelvo a pedir el conjunto de libros del socio en O(1) y elimino al libro de dicho conjunto en O(log r). Luego, lo agrego a la estanteria en O(log L) como hice en el primer método.

¿y las posiciones libre?

- Proc prestados:

Devuelvo una referencia al conjunto de libros que tiene retirados el socio. Esto toma O(1). ~~Hay aliasing por~~ Habrra que tener cuidado con posibles aliasing al trabajar por referencia.

- Proc ubicacionDelLibro: busco la posición del libro en estanteria lo cual toma O(log L) y la devuelvo en O(1).

· Notar que en mi módulo supongo que Biblioteca está implementada mediante tries, AVLs y minHeap lo cual me permite operar con estas complejidades.

HOJA Nº 3

c) proc agregarLibroAlCatalogo (inout b: Biblioteca, in l: idLibro)

```
b.coleccion.agregar (l)              // O(log L)

if ( b.posicionesLibres.vacia () ) {     // O(1)
    p := b.estanteria.longitud()         // O(1)
    b.estanteria.definir (l, p+1)        // O(log L)
} else {
    min := b.posicionesLibres.proximo ()     // O(1)
    b.posicionesLibres.desencolar ()         // O(log K)
    b.estanteria.definir (l, min)            // O(log L)
}
endif
```

(B)

Como hago análisis del tpeor me voy a quedar con el máx del if-else.

$t_{peor} = O(\log L) + O(\max\{\log L, \log K + \log L\})$
$= O(\log K + \log L)$ ✓

proc devolverLibro (inout b: Biblioteca, in l: idLibro, in s: Socio)

```
librosRetirados := b.socios [socio]     // O(1)
librosRetirados . eliminar (l)           // O(log r)
Idem (B)                                 // O(log K + log L)
```

Complej: $O(\log r + \log K + \log L)$

NOTA

E2) TAD Comercio {

obs ventasPorProducto: dict<Producto, seq<tupla<Fecha,Monto>>>

}

Modulo ComercioImpl implementa Comercio {

(1) var ventas: Secuencia <tupla <Producto, Fecha, Monto>>

(2) var totalPorProducto: Diccionario <Producto, Monto>

(3) var ultimoPrecio: Diccionario <Producto, Monto>

a) Voy a comentar sobre el invRep:

- (2): todo ~~elemento~~ clave de este diccionario debe estar asociado a un valor ≥ 0. Esta variable va a contener todos los productos del comercio (incluso los que tienen 0 ventas). ✓ OK

- (1): no van a haber elementos repetidos ni 2 tuplas que compartan el mismo producto y la misma fecha. Todas las tuplas tienen fecha ≥ 0 y monto ≥ 0. ✓

- Toda tupla que está en (1) tiene que tener al producto en (2) y con valor asociado $\neq 0$. ✓

- Todo producto en (2) con valor asociado $\neq 0$ debe aparecer como primer elemento en una tupla de (1) al menos una vez. ✓

- El valor de un producto va a valer 0 en (2) si ~~y solo si~~ el producto no está en (1). ✓ OK

En caso contrario, el valor va a ser igual a la suma del monto de todas las ventas del producto en (1).

- Un producto está en (3) si y solo si el producto está en (2) y su valor asociado es > 0.

- El valor asociado al producto en (3) es el del monto que está en la tupla (Producto, Fecha, Monto) con la fecha más grande del producto en ventas.

Ahora voy a comentar sobre el predAbs:

- (4) obs ventasPorProducto
- (2) y (4) tienen las mismas claves.
- Un producto de (2) vale 0 si y solo si la longitud de su secuencia en (4) es 0. En caso contrario, todas las apariciones del producto en (1) deben aparecer únicamente en la secuencia de (4) (sin repetirse).

Aclaraciones:

1- Asumí que un producto no puede venderse por 0 porque no tiene sentido (no cambia mucho).

2- Usé in cuando me refería a que un elemento pertenece a una secuencia porque no quería que sea demasiado ilegible. Es análogo a hacer: $(\exists i : \mathrm{int})(0 \leq i < |s| \wedge s[i] = e)$.

NOTA

CT ≡ c'.totalPorProducto , CV ≡ c'.ventas , CV ≡ c'.ultimoPrecio

OV ≡ c.ventasPorProducto ok

HOJA Nº 5

FE[redacted]

b) pred invReps (c': ComercioImpl) {

(∀p: Producto) (p in c'.totalPorProducto ⇒L

c'.totalPorProducto [p] ≥ 0) ✓ ∧ (∀i : int) (0 ≤ i < |~~c'.totalPor~~ CV

~~Producto~~| ⇒L ~~c'.totalPorProducto~~ CV $[i]_1$ ≥ 0 ∧ ~~CT~~ CV $[i]_2$ > 0) ✓ ∧

(∀i,j : int) (0 ≤ i,j < |cv| ∧ i ≠ j ⇒L (cv $[i]_0$ = cv $[j]_0$

⇒ cv $[i]_1$ ≠ cv $[j]_1$)) ✓ ∧ (∀t : <Producto, Fecha, Monto>)

CT ≡ c'.totalPorProducto , CV ≡ c'.ventas , CU ≡ c'.ultimoPrecio
OV ≡ c.ventasPorProducto ok

HOJA N° 5

b) pred invRep (c': ComercioImpl) {
(∀p: Producto) (p in c'.totalPorProducto ⟹$_L$
c'.totalPorProducto[p] ≥ 0) ∧ (∀i: int) (0 ≤ i < |~~c'.totalPorProducto~~ CV| ⟹$_L$ ~~c'.totalPorProducto~~ CV[i]$_1$ ≥ 0 ∧ ~~CX~~ CV[i]$_2$ > 0) ∧
(∀i, j: int) (0 ≤ i,j < |CV| ∧ i ≠ j ⟹$_L$ (CV[i]$_0$ < CV[j]$_0$
⟹ CV[i]$_1$ ≠ CV[j]$_1$)) ∧ (∀t: ⟨Producto, Fecha, Monto⟩)
(t in CV ⟹ t$_0$ in CT ∧$_L$ CT[t$_0$] > 0) ∧
(∀p: Producto) (p in CT ∧$_L$ CT[p] ≠ 0 ⟹
A = (∃t: ⟨Fecha, Monto⟩) ((p, t$_0$, t$_1$) in CV)) ∧
(∀p: Producto) (p in CT ∧$_L$ CT[p] = 0 ⟹ ¬A) ∧
(∀p: Producto) (p in CT ∧$_L$ CT[p] > 0 ⟹$_L$
CT[p] = montoTotal (CV, p)) ∧
(∀p: Producto) (p in CU ⟺ p in CT ∧$_L$ CT[p] > 0)
∧ (∀p: Producto) (p in CU ⟹$_L$ (∃f: Fecha)
((p, f, CU[p]) in CV ∧ ¬ (∃t: ⟨Fecha, Monto⟩)
((p, t$_0$, t$_1$) in CV ∧ t$_0$ ≠ f ∧ t$_0$ > f)))

}

pred predAbs (c': ComercioImpl, c: Comercio) {
(∀p: Producto) (p in CT ⟺ p in OV) ∧
(∀p: Producto) (p in CT ∧$_L$ CT[p] = 0 ⟺
p in OV ∧$_L$ |OV[p]| = 0) ∧
(∀p: Producto) (p in OV ∧$_L$ |OV[p]| ≠ 0 ⟹
(∀t: ⟨Fecha, Monto⟩) (t in OV[p] ⟹$_L$ (p, t$_0$, t$_1$) in CV

NOTA

CS CamScanner

$\wedge$ ($\forall$ p: Producto) (p in ov $\Rightarrow_L$ ($\forall$ i, j: int)

(0 ≤ i, j < |ov[p]| $\wedge$ i ≠ j $\Rightarrow_L$ ov[p][i] ≠ ov[p][j]$_0$))

$\wedge$ ($\forall$ p: Producto) (p in ov $\wedge_L$ |ov[p]| ≠ 0 $\Rightarrow_L$

($\forall$ t: <Fecha, Monto>) ((p, t_0, t_1) in cv $\Rightarrow_L$ t in ov[p]))

}

aux montoTotal (s : Secuencia <tupla <Producto, Fecha, Monto>>,

p: Producto) : int

$$\sum_{i=0}^{|s|-1} \text{if } s[i]_0 = p \text{ then } s[i]_2 \text{ else } 0 \text{ fi} ;$$

NOTA

E3)

a) proc recolector (in s: Vector <tupla <string, int>>,
in u: Diccionario <string, int>) : Vector <string>

```
stockTotal := DiccionarioVacio()   // O(1)
for each t in s{   // O(n)
    if (stockTotal.pertenece(t[0])){   // O(1)
        old := obtener(stockTotal, t[0])   // O(1)
        definir(stockTotal, t[0], old + t[1])   // O(1)
    } else {
        definir(stockTotal, t[0], t[1])   // O(1)
    } endif
} h := stockTotal.longitud()   // O(n)
aOrdenar := new Array(h)  VectorVacio()   // O(1)  // O(h)
i := 0   // O(1)
for each elem in u{   // O(h)   *
    aOrdenar[i] = agregarAtras (elem[0], elem[1],   // O(1)
    obtener(stockTotal, elem[0]))
    i++   O(1)
}
```

// Me piden que el vector tenga primero a los más usados y que menos reservas tengan. Esto es equivalente a que mi vector sea decreciente respecto al 2° elemento de las tuplas y creciente respecto al 3°. Obvio que cada orden tiene una prioridad.

NOTA

(*) que quiero ordenar cosas en base a criterios, debo empezar con el menos significativo y terminar con el más significativo.

Por el orden de importancia voy a ordenar primero al vector según el 3° elemento con un mergeSort() (asc) y luego lo vuelvo a ordenar respecto al 2° con un mergeSortInvertido() (desc) que me va a dar un arreglo de forma decreciente. /// Esto porque siempre (*)

CamScanner

empezar con el menos significativo y terminar con el más significativo.
Por el orden de importancia voy a ordenar primero al vector según el 3° elemento con un mergeSort() (asc) y luego lo vuelvo a ordenar respecto al 2° con un mergeSortInvertido() (desc) que me va a dar un arreglo de forma decreciente. // Esto porque siempre (*)

* La complejidad de agregar h elementos a un vector vacío es O(h) (y amortizado O(1))

```
b := a Ordenar. mergeSort()           // O(h. log h)
c := b. mergeSortInvertido()          // O(h. log h)
res := VectorVacio()                  // O(1)
for each elem in c{
    res. agregarAtras(elemCos)        } O(h)
}

return res   // O(1)
```

Complej: O(h log h + n)

Aclaraciones:

- Supuse que U y stockTotal son diccionarios implementados sobre un trie. Esto para aprovechar que las keys están acotadas y que puedo operar con las variables en O(1).
- Cada mergeSort tiene un método compare especial para cada caso. En el normal, evalúa en base al 3° elemento y el invertido en base al segundo.

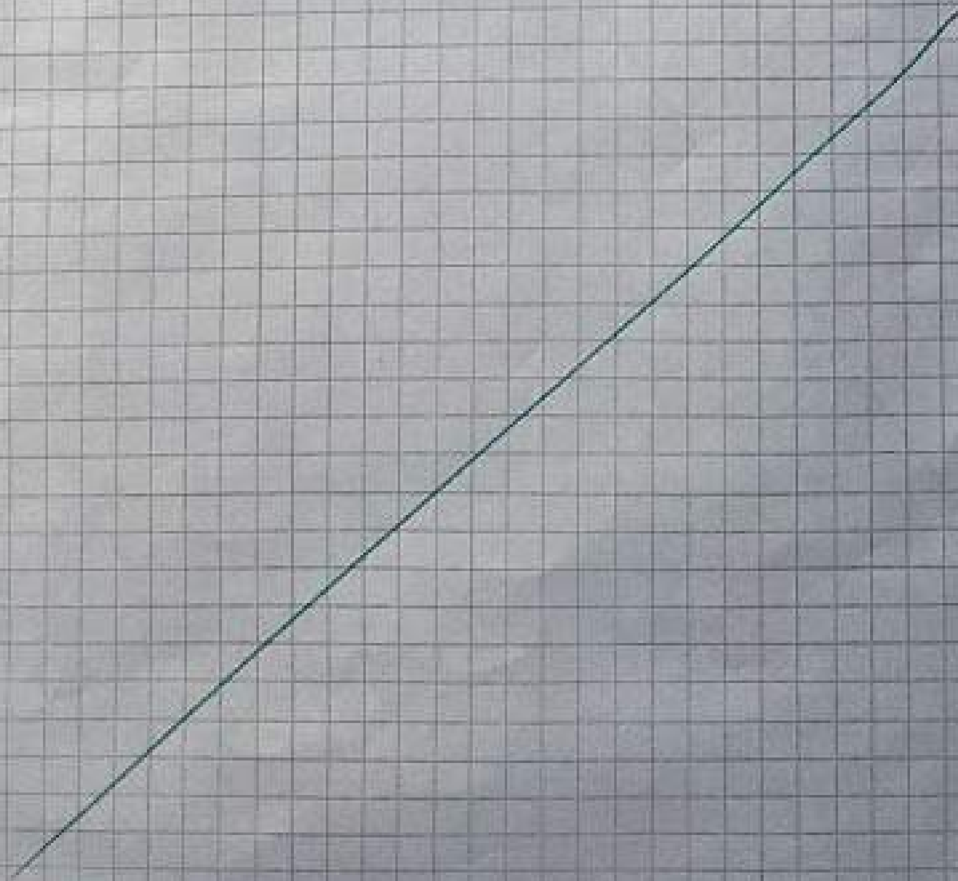
NOTA

b) El mejor caso sería tener un único elemento repetido porque h sería 1 y mi complejidad queda $O(n)$.

Sé que mi algoritmo obedece según lo siguiente: $\Omega(n)$ y $O(n + h \cdot \log h)$

Tengo dos opciones para Θ:
$\Theta(n)$ o $\Theta(n + h \cdot \log h)$

Como $\Theta(n)$ solo ocurre en casos muy particulares, creo que $\Theta(n + h \cdot \log h)$ sería lo más adecuado para catalogarla como la cota más ajustada. OK



NOTA: