

# Algoritmos y Estructuras de Datos II

## Segundo parcial — Miércoles 2 de Noviembre de 2016

\* 12 de junio de 2017

### Aclaraciones

- El parcial es a libro abierto.
- Cada ejercicio debe entregarse en hojas separadas.
- Incluir en cada hoja el número de orden asignado, número de hoja, apellido y nombre.
- Al entregar el parcial, completar el resto de las columnas en la planilla.
- Cada ejercicio se calificará con Promocionado, Aprobado, Regular, o Insuficiente.
- El parcial está aprobado si el primer ejercicio tiene al menos A, y entre los ejercicios 2 y 3 hay al menos una A.

### Ej. 1. Diseño

Estamos comenzando la exploración intensiva del planeta Marte utilizando unidades *Mars Rover*. El área de exploración se ha discretizado en coordenadas enteras y en cualquier momento pueden desplegarse unidades *rover*, aunque siempre en la celda (0,0) de la discretización. Los rovers desplegados pueden moverse en cualquier momento en cualquiera de sus cuatro direcciones (norte, sur, este u oeste), siempre que sea hacia una celda válida, y estos movimientos se registran con fines estadísticos. En cualquier momento puede también desactivarse definitivamente cualquiera de las unidades activas. Dado un *itinerario* (i.e., una secuencia de movimientos desde la celda (0,0)), la consulta más requerida es la de saber cuántos rovers (activos o inactivos) realizaron exactamente este itinerario. Si bien no interesa conocer el recorrido hecho por un rover inactivo, notar que este dato es importante para la consulta por itinerario.

TAD ROVER es NAT, TAD DIRECCION es {N, S, E, O}, TAD CELDA es tupla<NAT,NAT>

#### TAD EXPLORACIÓN

##### observadores básicos

activos	: exploracion	→ conj(rover)
inactivos	: exploracion	→ conj(rover)
recorridoActual	: exploracion e × rover r	→ secu(direccion)
vecesRecorrido	: exploracion e × secu(direccion) i	→ nat

{r ∈ activos(e)}

##### generadores

iniciar	: exploracion	→ exploracion
desplegar	: exploracion e × rover r	→ exploracion
mover	: exploracion e × rover r × dirección d	→ exploracion
desactivar	: exploracion e × rover r	→ exploracion

{r ∉ rovers(e)}

{r ∈ activos(e) ∧ movimientoValido(e,r,d)}

{r ∈ activos(e)}

##### otras operaciones

rovers	: exploracion e	→ conj(rover)
ubicación	: exploracion e × rover r	→ celda

{r ∈ activos(e)}

axiomas	Ve: exploracion, ∀r: rover, ∀d: dirección
	rovers(e) = activos(e) ∪ inactivos(e)

Fin TAD

Se debe realizar un diseño que cumpla con los siguientes órdenes de complejidad temporal en el peor caso, siendo  $r$  la cantidad de rovers activos en el sistema:

- Desplegar y desactivar un rover en  $O(\log r)$ .
- Mover un rover en  $O(\log r)$ .
- Saber dónde está un rover dado en  $O(\log r)$ .
- Saber cuántos rovers realizaron el itinerario  $T$  en  $O(\text{long}(T))$ .
- Obtener el conjunto de rovers activos y el de inactivos, cada operación en  $O(1)$ .

Se pide:

1. Escriba la estructura de representación del módulo “Exploración” explicando detalladamente qué información se guarda en cada parte de la misma y las relaciones entre las partes. Describa también las estructuras de datos subyacentes.
2. Escriba el algoritmo para mover un rover y justifique el cumplimiento de los órdenes solicitados. Para cada una de las demás funciones, describalas en castellano, justificando por qué se cumple el orden pedido con la estructura elegida.

## Ej. 2. Ordenamiento

Se tiene un arreglo  $R$  con  $n$  strings sin repeticiones que define un *ranking*. Se tiene además un arreglo  $A$  de  $m$  strings tal que todos ellos aparecen en el ranking  $R$ . Se quiere ordenar el arreglo  $A$  en función del ranking definido por  $R$ . Es decir, dados dos elementos  $s$  y  $t$  de  $A$ ,  $s$  será “menor” que  $t$ , si aparece en  $R$  antes que  $t$ . Por ejemplo, si tenemos

$$R = [\text{Brasil}, \text{Argentina}, \text{Alemania}, \text{Chile}, \text{Colombia}, \text{Francia}] \text{ y}$$
$$A = [\text{Chile}, \text{Francia}, \text{Brasil}, \text{Chile}, \text{Argentina}, \text{Brasil}],$$

entonces el orden correcto para  $A$  sería  $[\text{Brasil}, \text{Brasil}, \text{Argentina}, \text{Chile}, \text{Chile}, \text{Francia}]$ .

Suponiendo que el largo de todos los strings está acotado por una constante, proponga un algoritmo de ordenamiento que resuelva el problema en una complejidad  $O(n + m)$ , donde  $n$  y  $m$  son las cantidades de elementos en el ranking y en el arreglo a ordenar, respectivamente. Justifique la correctitud del algoritmo y su complejidad temporal.

## Ej. 3. Dividir y Conquistar

Una empresa informa todos los días el cierre de sus ganancias netas (las cuales pueden ser negativas si tuvo pérdida ese día). Dado un arreglo con tales montos para un período de  $n$  días consecutivos, el directorio de la empresa necesita conocer en qué intervalo de días (dentro de este período) se obtuvo la mayor ganancia (la ganancia en un intervalo de días es la suma de los montos informados en los días dentro de ese intervalo).

Usando la técnica de D&C, dar un algoritmo que tome un arreglo con los montos de  $n$  días consecutivos (i.e., los días de 0 a  $n - 1$ ) e indique un intervalo de días  $[i, j]$ , con  $0 \leq i \leq j < n$ , en el cual se maximiza la suma de tales montos (puede asumirse que  $n > 0$ ). El algoritmo propuesto debe tener complejidad estrictamente menor que  $O(n^2)$ , lo cual debe justificarse adecuadamente.

**Ayuda:** Si pidiésemos que el intervalo devuelto contenga sí o sí un determinado día  $k$ , entonces el problema se resuelve en tiempo  $O(n)$  (¡pensar cómo!). Aprovechar ese procedimiento en el algoritmo global de D&C implementado.

P

ej1 | ej2 | ej3  
P | P | P

; Excelente parcial!

## Algoritmos y Estructuras de Datos II Flaja 1/4 2º parcial

### Ejercicio 1

EXPLORACIÓN se representa con estr, donde

estr es tupla ItinerarioPorRover: diceLog(rover, infoRover),  
positionPorRover: diccLog(rover, celdas) → puedes ponerlo como en infoRover para simplificar  
cantidadPorItinerario: diccTrie (seco(dirección, nat)),  
activos: conjLog(rover),  
inactivos: conjLineal(rover) )  
→ no hace falta, ya está en el Trie

infoRover es tupla (itinerario: seco(dirección), it: itDiccTrie (seco(dirección, nat)))

- ✓ Los diccionarios y conjuntos logarítmicos pueden estar implementados con un AVL, un red-black tree o alguna otra estructura; lo que me importa es que permiten borrado, inserción y búsqueda en  $O(\log(r))$  tal como se pide.
- ✓ El conjunto lineal puede estar implementado con una lista no ordenada o con alguna otra estructura, mientras permita inserción en  $O(1)$ .

### Partes de la estructura:

/ itinerarioPorRover guarda, para cada rover activo, el recorrido que hizo (la información que nos interesa conocer) y un iterador a cantidadPorItinerario de manera que este diccionario se pueda actualizar en  $O(1)$  cuando se modifica el recorrido de un rover.

/ positionPorRover guarda, para cada rover activo, la posición en la que se encuentra. Los rovers deben ser los mismos que en itinerarioPorRover y su posición debe corresponderse con el recorrido hecho. En el algoritmo para mover un rover se detallará cómo se mantiene actualizada esta información.

/ cantidadPorItinerario guarda, para un itinerario dado, la cantidad de rovers que lo realizaron, incluyendo los inactivos. Debido a esto último, no es necesario actualizarlo cuando se desactiva un rover, pero sí cuando se mueve (detallado en el algoritmo) o se despliega (simplemente se incrementa en 1 la cantidad de rovers que realizaron el recorrido vacío, como buscar y definir en un trie son  $O(\text{longitud de la clave})$ , esto es  $O(\text{long(}secuencia\ vacía\text{)}) = O(1)$ ). \*

/ activos guarda los rovers activos, que deben ser los mismos definidos en itinerarioPorRover y positionPorRover. Mantenerlo actualizado cuando se despliega o se desactiva un rover no es un inconveniente para la complejidad de estas operaciones, ya que debería tener la misma cantidad de rovers que ellos. Siempre voy a tener que insertarlo o borrarlo de las tres estructuras pero esto es  $O(\log(r)) + O(\log(r)) + O(\log(r)) \in O(\log(r))$ .

inactivos guarda los rovers que fueron desactivados. Cada vez que desactive un rover, además de borrarlo de itinerarioPorRover, posicionPorRover y activos, debo agregarlo a este conjunto. Esta operación mantiene la complejidad  $O(\log(r))$  pedida anteriormente, ya que como detallé al principio, la estructura elegida para inactivos permite inserción en  $O(1)$ , y

$$O(\log(r)) + O(1) \in O(\log(r)).$$

## Funciones:

```
mover (in/out e:exploracion, in r:rover, in d:direccion) {
    infoRover info <- obtener(e.itinerarioPorRover, r)  $O(\log(R))$ , aliasing
    siguiente(info.it) <- siguiente(info.it) - 1  $O(1)$ , aliasing
    incrementarSegunDireccion(info.it, d)  $O(1)$ 
    secu(direccion) recorrido <- info.itinerario  $O(1)$ , aliasing
```

\* / Implemento estas estructuras de tal manera que obtener genere aliasing y la secuencia se devuelva por referencia (o sea que también genera aliasing, además de ser  $O(1)$  por no copiar la secuencia). De esta forma, los cambios que se hacen en estas variables afectarán el diccionario subjacente y se cumplirán los requerimientos. /\*

```
agregarAtras(recorrido, d)  $O(1)$ , aliasing
celda posicion <- obtener(e.posicionPorRover, r)  $O(\log(R))$ , aliasing
posicion <- actualizarPorDireccion(posicion, d)  $O(1)$ , aliasing
```

$$O(\log(R)) + O(1) + O(1) + O(1) + O(1) + O(\log(R)) + O(1) \in O(\log(R)).$$

La complejidad pedida podría mantenerse si se devolvieran por copia y se redefiniera  $r$  en posicionPorRover, ya que copiar una tupla de naturales es  $O(1)$  y definir es  $O(\log(r))$ . Sin embargo, elegí devolverlas por referencia y generar aliasing para que sea coherente con el otro diccionario.

Siguiente:  $\text{itDiccTri}(k, \text{net}) \rightarrow \text{net}$  es una función que, dado un iterador a un diccionario sobre trie, devuelve el significado asociado a la posición del iterador y genera aliasing. En este caso la uso para decrementar en 1 la cantidad de rovers con el itinerario anterior del rover que se movió.

incrementarSegunDireccion:  $\text{itDiccTri}(\text{secu(direccion)}, \text{net}) \times \text{direccion} \rightarrow \text{diccTri}(\text{secu(direccion)}, \text{net})$

es una función que, dado un iterador  $\text{it}$  y una dirección  $d$ , incrementa en 1 el significado que se encuentra en la posición a la que se llega bajando desde la posición de  $\text{it}$  en el nodo  $d$ . En este caso la uso para incrementar en 1 la cantidad de rovers cuyo recorrido es el nuevo itinerario del rover que se movió.

actualizarPorDireccion:  $\text{celda} \times \text{direccion} \rightarrow \text{celda}$  incrementa o decremente las coordenadas de la celda pasada como parámetro según la dirección (aumenta 1 si la dirección es E, decrementa 1 si la dirección es S, etc.).

## Algoritmos y Estructuras de Datos II Folio 2/4 2º parcial

desplegar (in/out e: exploracion, in r: rover)

- Defino r en itinerarioPorRover con el itinerario vacío  $O(\log(R))$
- Defino r en posiciónPorRover con  $(0, 0)$   $O(\log(R))$
- Agrego r a activos  $O(\log(R))$
- Aumento en 1 la cantidad de rovers con itinerarios vacíos en cantidadPorItinerario  $O(1)$

$$O(\log(R)) + O(\log(R)) + O(\log(R)) + O(1) \in O(\log(R))$$

desactivar (in/out e: exploración, in r: rover)

- Borro r de itinerarioPorRover  $O(\log(R))$
- Borro r de posiciónPorRover  $O(\log(R))$
- Borro r de activos  $O(\log(B))$
- Agrego r a inactivos  $O(1)$

Complejidad ídem anterior

/ ubicacion (in e: exploraciones, in r: rover)  $\rightarrow$  res: celda

- Busco r en posiciónPorRover  $O(\log(R))$

/ cuentasRealizadas (in e: exploración, in T: seco(dirección))  $\rightarrow$  res: nat

- Busco T en cantidadPorItinerario  $O(\log(T))$

/ activos e inactivos consisten simplemente en devolver por referencia los conjuntos correspondientes (si fuera por copia no sería  $O(1)$ ).

\* Técnicamente la longitud de la secuencia vacía es 0, pero definir algo en un trío para la secuencia vacía es  $O(1)$  ya que sólo implica recorrer una cantidad fija de nodos.

Excelente!!!

# Algoritmos y Estructuras de Datos II

## Hoja 3/4

2º parcial

### Ejercicio 2

Observemos que, como el largo de todos los strings está acotado por una constante, buscarlos o definirlos en un "diccionario de ranking" implementado con un trie es de complejidad ✓

$$O(\text{máxima longitud}) = O(\text{cte.}) = O(1).$$

Entonces, a partir de  $R$ , puedo generar un diccionario donde a  $R[0]$  le corresponde 0, a  $R[1]$  1, etc. Dado que  $R$  tiene  $n$  elementos, la complejidad es ✓

$$n \cdot O(\text{definir en el dicc.}) = n \cdot O(1) = O(n).$$

Al habiendo definido el ranking, podemos hacer una variante de counting sort en la que, en vez de aumentar la posición  $A[i]$ -ésima del arreglo auxiliar de cantidades, agregamos  $A[i]$  a la posición  $\text{ranking}(A[i])-ésima$  de un arreglo de listas (buckets). Como calcular  $\text{ranking}(A[i])$  es  $O(1)$  con el trie descrito anteriormente, ubicar los elementos en los buckets es

✓  $\rightarrow O(\text{acceder a una pos. de un arreglo})$

$$m \cdot O(\text{copiar algo de longitud acotada}) \cdot O(\text{agregar atrás en una lista}) = m \cdot O(1) \cdot O(1) = O(m).$$

✓  $\hookrightarrow O(1)$

Por ahora, nuestra complejidad es  $O(n) + O(m) = O(n+m)$ . Queda recorrer el arreglo ordenado. Como los buckets del arreglo auxiliar están ordenados entre sí por ranking creciente (\*) y cada uno contiene todos los elementos del arreglo original a los que les corresponde un ranking determinado, podemos recorrer secuencialmente el arreglo auxiliar e ir ubicando los elementos de cada bucket en un arreglo de tamaño  $m$ , y la complejidad es

$$\sum_{i=0}^{\text{long}(aux)-1} O(\text{acceder a aux}[i]) + O(\text{copiar todos los elementos de aux}[i])$$

$$= \sum_{i=0}^{\text{long}(aux)-1} O(1) + O(\text{long}(aux[i])) = \sum_{i=0}^{n-1} O(1) + \sum_{i=0}^{n-1} O(\text{long}(aux[i]))$$

$$= O\left(\sum_{i=0}^{n-1} 1\right) + O\left(\sum_{i=0}^{n-1} \text{long}(aux[i])\right) = n \cdot O(1) + O(m) = O(n) + O(m)$$

$$= O(n+m).$$

Como  $O(n+m) + O(n+m) \in O(n+m)$ , nuestro algoritmo satisface lo pedido.

( $\text{long}(aux) = n$  porque hay  $n$  buckets de ranking correspondientes a los  $n$  elementos de  $R$  y  $\sum_{i=0}^{n-1} \text{long}(aux[i]) = m$  porque en la unión de todos los buckets tienen que estar los  $m$  elementos de  $A$ , ni más ni menos).

```

ordenarPorRanking (in/out A: arreglo(string), in R: arreglo(string) ) {
    diccTri (string, net) diccRanking ← crearTriRanking (R) O(n)
    not m ← tam(A), n ← tam(R) O(1) *
    arreglo (List2(string)) aux ← arreglo [n] O(n) ⊕
    for m-1 i<0 .. m-1 {
        net posición ← obtener (diccRanking, A[i]) O(1) } mO(1) = O(m)
        agregarAtras (aux[posición], A[i]) O(1)
    }
    not j < O(1) n iteraciones
    for not i < [0..n-1] {
        itList2 it ← crearIt (aux[i])
        long (aux[i])
        while (haySiguiente (it)) {
            A[i] ← siguiente (it)
            avanzar (it) O(1)
            j++ O(1)
        }
    }
}

```

justificado anteriormente

\*) Asumimos que se inicializa con listas vacías (crear una lista vacía es  $O(1)$ ).

\*) Hace lo detallado en la cajilla anterior.

(\*) Me refiero a que el arreglo de listas está ordenado por clase de ranking de los buckets

## Algoritmos y Estructuras de Datos II Hoja 4/4

2º parcial

3) Si divido el arreglo en 2 mitades, tengo 3 casos posibles:

- la mayor ganancia está entre las dos mitades
- la mayor ganancia está en la mitad izquierda
- la mayor ganancia está en la mitad derecha

Cuando es lo primero es porque, si  $m = \lfloor \text{tam}(A)/2 \rfloor$ , la suma de la mayor ganancia del subarreglo izquierdo que contiene a  $m-1$  y la mayor ganancia del subarreglo derecho que contiene a  $m$  es mayor que la de la mitad izquierda y la de la mitad derecha.  
La mayor suma de cada mitad puede calcularse con este mismo algoritmo de manera recursiva.

~~mayorGanancia (in a: arreglo (int)) → res: tupla (nat, nat) {~~

~~nat n ← tam(A)~~

~~if (n == 1) {~~

~~return (0, 0)~~

~~} else {~~

~~mayorIzq ← (<sup>? sum</sup> mayorGanancia (a[0..n/2-1]),~~

~~mayorIzq ← mayorSumaAv,~~

~~mayorIzq ← mayorGananciaAux (a[0..n/2-1])~~

~~mayorIzq ← mayorGananciaIndices (a[0..n/2-1])  $T(n/2)$~~

~~mayorDer ← mayorGananciaIndices (a[n/2..n-1])  $T(n/2)$~~

~~mayorAcIzq ← mayorConK (a[0..n/2-1], n/2-1)  $O(n)$~~

~~mayorAcDer ← mayorConK (a[n/2..n-1], 0)  $O(n)$   $n/2$  es la pos. 0 del derecho~~

~~return maximaGanancia~~

~~return indicesDeLaMaximaGananciaEntre (mayorIzq, mayorDer, mayorAcIzq + mayorAcDer)~~

3

(si elige  $\text{mayorAcIzq} + \text{mayorAcDer}$ , debería devolver el índice izquierdo de  $\text{mayorAcIzq}$  y el derecho de  $\text{mayorAcDer}$ )

~~mayorConK es la función sugerida en el enunciado.~~

→ Ten los que hacerlo...

La complejidad, con  $n = \text{tam}(A)$ , es

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + f(n), & f(n) \in O(n) \\ \Theta(1) & n=1 \end{cases} \quad n > 1$$

porque en cada paso recursivo resuelve 2 subproblemas de tamaño  $n/2$  (buscar en las mitades) y hace algo de orden  $O(n)$  (calcular las sumas acumulables)

Tenemos  $\log_2(2) = 1$  y  $f(n) \in \Theta(n)$  \*

$$\Theta(n^{\log_2(2)})$$

Entonces, por el Teorema Maestro,  $T(n) \in \Theta(n^{\log_2(2)} \log(n))$

$\Rightarrow T(n) \in \Theta(n \log(n))$ , estrictamente menor a  $O(n^2)$ .

\* Hecho desde un extremo del arreglo, mayor con  $K$  es sencillo y consiste simplemente en mirar todos los elementos desde el extremo dado, guardar la ~~mas~~ el índice con el que se obtuvo la máxima suma y comparar con la suma incluyendo a cada elemento hasta llegar al final. Se ve fácilmente que esta función pertenece a  $\Theta(n)$ .

(El Teorema Maestro se aplica a recurrencias del tipo

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > k \\ \Theta(1) & n \leq k \end{cases}, f \text{ polinómica}$$

En este caso,  $a=2$ ,  $b=2$  y  $k=1$ , por lo que podemos usarlo.)