

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}  $\Rightarrow$  TRIE.  
Recolectar(stock, usos) = ["Lavanda", "Diente de León", "Margarita", "Menta"]
```

Los primeros son la lavanda y el diente de león porque ambos tienen 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.

- Se pide escribir el algoritmo de `Recolectar`. Justificar detailedadamente la complejidad y escribir todas las suposiciones sobre las implementaciones de las estructuras usadas, entre otras.
- ¿Cuál sería el mejor caso para este problema? ¿Cuál sería la cota de complejidad más ajustada?

1) Ordenar hierbas & el criterio de desempate, la secu.,
POR STOCK.

Luego, por USOS.

Para manejar la lógica mejor a usar MERGE SORT.

2) Por finales al final se tarda en $O(1)$ si buscan
en Trie (usos) finales crecer o los bus en $O(1)$.

Me voy a crear en TRIE para almacenar, agregar hasta $O(1)$ y así sacarlos.

{ HIERBA: (HIERBA, STOCK, USOS) ... }.

Luego por O ALTA DE TUVAS, orden, y luego sacar un array
de CANT TUVAS y hacer los nombres.

Recolectar: $O(n)$, Almacenar en TRIE: $O(1)$, Recorrer array: $O(1)$, Almacenar en TRIE: $O(1)$,

Si $n = m = O(n)$ es lo más rápido. P. P. $O(n)$ $O(n)$

función $T(n)$ & ALGOR $O(k)$ A su vez, función $n \cdot \log(n)$, $n \cdot \log(k)$

$$O(m + 1 + 1 + \underbrace{R}_{\text{DEP}} + R \cdot \log(R) + R \cdot \log(k)) = O(m + R \cdot \log(k))$$

$\text{Si } R \neq \text{DEP}$

PROC RECOLECTAR (IN S: VECTOR<TUPLA<STRING, INT>>, IN U: DICCIONARIO DIGITAL<STRING, INT>): ARRAY<STRING>

VAR D: DICCIONARIO DIGITAL<HIERBA, (HIERBA, STOCK, USOS)> := NEW DICCIONARIO DIGITAL<HIERBA, (HIERBA, STOCK, USOS)>();

//NE COMO STOCK 0(M)

VAR i: INT := 0;

WHILE (i < S.LENGTH) DO //O(M)

VAR HIERBAACTUAL: TUPLA<STRING, INT> := S.OBTENER(i); // (H, S)

VAR STOCK := HIERBAACTUAL.1;

IF (D.ESTA(HIERBAACTUAL.0) == TRUE) THEN

VAR HIERBAACTUALD: TUPLA(HIERBA, STOCK, USOS) := D.OBTENER(H0); //O(1) (H, S, U)

STOCK := HIERBAACTUALD.1 + HIERBAACTUAL.1

ENDIF

Forwards loop.

D.DEFINIR(HIERBAACTUAL.0, TUPLA(HIERBAACTUALD.0, STOCK, 0))

i++;

ENDWHILE

VAR IT := U.ITERADOR();

WHILE (IT.HAYSIGUIENTE()) DO //O(R * max(k)) = O(R) ^{para hierbas sin DEP} (H, U).

VAR HIERBAUSOS: TUPLA<STRING, INT> := IT.SIGUIENTE();

VAR HIERBAACTUALD := D.OBTENER(HIERBAUSOS.0); // (H, S, U) \Rightarrow ASUMO q si tiene hierba, \exists en stock.

D.DEFINIR(HIERBAACTUALD.0, (HIERBAACTUALD.0, HIERBAACTUALD.1, HIERBAUSOS.1))

ENDWHILE

// PASO EL TNE A ARRAY DE TUPLAS

VAR MA: ARRAY<STRING, INT, INT> := NEW ARRAY() {D.TAMAÑO()};

IT := D.ITERADOR();

i := 0;

WHILE (IT.HAYSIGUIENTE()) DO //O(R) \Rightarrow 1 TUPLA X CADA HIERBA

MA[i] := IT.SIGUIENTE();

$i++$

ENDWHILE

HA := MERGESORT(HA); // ASC POR POS 1 DE TUPA $\Rightarrow (H, S, V) \xrightarrow{\text{POrN STACK}} O(n \cdot \log(n))$ ✓

HA := MERGESORT(HA); // DESC POR POS 2 DE TUPA $\Rightarrow (H, S, V) \xrightarrow{\text{USOS}} O(n \cdot \log(n))$

VAR ORDENADAS: ARRAY<STRING> := NEW ARRAY<STRING>() { HA.LENGTH }

$i := 0;$

WHILE ($i < HA.LENGTH$) DO // $O(R)$

ORDENADAS[i] := HA[i] o i // $\xrightarrow{\text{MIRADA}}$

$i++;$

ENDWHILE

RETURN ORDENADAS;

}

Complejidad: $O(M + R + R + R \cdot \log(R) + R \cdot \log(R) + R) = O(n + R \cdot \log(R))$ ✓

El mejor caso sería si stock no tuviera repeticiones, en ese caso sería $O(R \cdot \log(R))$ ✓
Lo que más optimiza es $O(M + R \cdot \log(R))$ ✓

bj) Escribir algoritmo en el lenguaje de pseudocódigo.

E3. Sorting (30 pts)

Se cuenta con un sistema de seguimiento de contaminación ambiental que cubre toda la ciudad de Buenos Aires. Así, regularmente se registra información de diferentes sensores en diferentes momentos y se quiere saber en qué zonas se registra mayor contaminación, ya que cada sensor está asociado a una zona particular de la urbe.

De esta forma, dados ciertos registros realizados, que constan del identificador del sensor, que es un valor alfanumérico de máximo 64 caracteres, el momento en que se tomó el registro y el valor medido, donde ambos son naturales no acotados, se quiere obtener el acumulado de cada sensor para las últimas k mediciones (se puede considerar cualquier valor de k , en tanto se tiene una gran cantidad de registros por cada sensor). Con esta información se desea hacer un ranking, donde aparezcan primero los sensores con mayor contaminación registrada acumulada. En caso de empate, se mostrarán primero los sensores con registros más recientes.

Se quiere implementar la función MÁS CONTAMINADOS, que recibe un arreglo que contiene mediciones de los sensores y la cantidad de mediciones que se desean considerar para obtener un acumulado, y se desea obtener un ranking, donde primero aparecen los sensores que tienen mayor valor total:

MásContaminados(in a: Array<struct<sensor: string, in t:int>>, in k: int): Array<string>
La función debe tener cota de peor caso $O(n \cdot \log n)$, siendo n la cantidad de mediciones registradas entre todos los sensores.

Por ejemplo, dados el A indicado abajo y k igual a 2, MásContaminados(A , 2) retornará [48A, 1AB, 1C], ya que tanto el sensor 1AB como el 48A registran un acumulado de 120 en las últimas 2 mediciones, pero el 48A tiene un registro más reciente, por lo que queda primero, y luego queda el 1C, que sólo acumula 55.

A = [(1AB, 8, 100), (48A, 10, 100), (1AB, 9, 25), (48A, 9, 25),
(1AB, 14, 95), (48A, 15, 20), (1C, 12, 5), (1C, 17, 50)]

- a) Se pide escribir el algoritmo de MásContaminados, justificando detaladamente la complejidad.
b) ¿Cuál sería el mejor caso para este algoritmo? ¿Cuál sería la cota de complejidad más ajustada?

LUEGO POR ACUM VALOR.

PRIMEROS medicos que se oyo (los {"SENSOR": MEDICINES}), pero revisa las leyes k veces

$$A = [(1AB, 8, 100), (48A, 10, 100), (1AB, 9, 25), (48A, 9, 25), (1AB, 14, 95), (48A, 15, 20), (1C, 12, 5), (1C, 17, 50)]$$

· OBJETO X RECIPIENTE $\Rightarrow M \cdot \log(m)$

$$[(1C, 17, 50), (48A, 15, 20), (1AB, 14, 95), (1C, 12, 5), (48A, 10, 100), (1AB, 9, 25), (48A, 9, 25), (1AB, 8, 100)]$$

· AHORA medico ordena por los k mas ACUM.

El k me dice cuantos medicos acuerdan de cada remedio.

k=1

$$[(1C, 17, 50), (48A, 15, 20), (1AB, 14, 95)] = [1AB, 1C, 48A]$$

k=2

$$[(1C, 17, 50), (\cancel{48A}, \cancel{15, 20}), (\cancel{1AB}, \cancel{14, 95}), (1C, 12, 5), (\cancel{48A}, \cancel{10, 100}), (\cancel{1AB}, \cancel{9, 25})]$$

$$[1C=55, 48A=120, 1AB=120] = [48A, 1AB, 1C]$$

Revisa k veces x cada SENSOR. GUARDA EN UNA TABLA CADA KEY.

Si m es la cantidad de MEDICINES entre John en Remedio, m >= CANTSENSES

"SEN": (ACUM, k) \Rightarrow Si k=2 no hay mas

PROC MASCONTAMINADOS(in A: ARRAY<STRUCT<SENSOR> STRING, T: INT, V: INT>; in K: INT): ARRAY<STRING>

A := MERGESORT(A); //Por 2da COMP DESC. M. log(m)

//Otro medico numero k de cada remedio, y ordena los que no sea 0.

VAR SENSORES := NEW DiccionarioDigital<SENSOR, (ACUM, K, POSARR)>();

\hookrightarrow guarda los pos relativos.

VAR i: INT := 0;

WHILE (i < A.LENGTH) DO //O(n)

if (SENSORES.ESTA(A[i])) == False)

}

0(s) \Rightarrow S < m

SENSORES.DESINIR(A[i]_0, (0,0,i)); //O(1) ✓
 ELSE IF (SENSORES.ESTA(A[i]_0) == true)
 VAR SD : TUPLA(A[0],K) := SENSORES.OBTENER(A[i]_0);
 IF (SD₁ != K) THEN
 SENSORES.DESINIR(A[i]_0, (SD₀+A[i]₂, SD₁+1, SD₂)); *Luego del sensor tipo de k
Ocurrieron o no cambios.*
 ENDIF
 ENDIF
 i++
 ENDWHILE

// Otras maneras de ordenar en base a pos negativa por ms. ✓
 VAR IT := SENSORES.ITERADOR(); O(Θ)
 VAR AR: ARRAY<(SENSOR, A[0], K, POS)> := NEW ARRAY(); {SENSORES.TAMAÑO();}
 i := 0;
 WHILE (IT.MAYSIGUIENTE()) DO //O(s) ✓
 VAR VAL := IT.SIGUIENTE();
 AR[i] := (VAL[0], VAL[1]_0, VAL[1]_1, VAL[1]_2);
 i++
 ENDWHILE; O(UWU)

AR := MERGESORT(AR); //Por que pos (indice tupla) \Rightarrow RECUPERA pos negativa
 \Rightarrow O(s.log(s)) \Rightarrow ASC

AR := MERGESORT(AR); //Por ten pos (indice tupla) \Rightarrow DESCAUM.
 \Rightarrow O(s.log(s))

VAR SEN: ARRAY<STRING> := NEW ARRAY<STRING>(); {AR.LENGTH};
 i := 0

WHILE (i < AR.LENGTH) DO //O(s)
 SEN[i] := AR[i]_0;
 i++; O(:)

ENDWHILE

RETURN SEN

}

O la linea m.

Complejidad: $O(m \cdot \log(m) + \overbrace{s + s \cdot \log(s) + s \cdot \log(s)}^{\text{O la linea m.}} + s) \equiv O(m \log m)$

```
TAD Materia {
    obs calificaciones: dict<Alumno, struct<primero: Nota, segundo: Nota>>
    obs aprobados: conj<Alumno>
    .obs reprobados: conj<Alumno>
}
Alumno es int
Nota es int
```

calificaciones relaciona a cada alumno con la nota que obtuvo en cada examen. La materia cuenta con dos exámenes, la primera posición del struct corresponde a la nota del primer examen y la segunda, a la del segundo. Las calificaciones toman valores entre 0 y 10. Si un alumno no se presenta a un parcial, su nota en ese parcial será cero. aprobados es el conjunto de los alumnos que aprobaron la materia. Un alumno aprueba si tiene más de 6 en ambos exámenes. reprobados es el conjunto de alumnos que reprobaron la materia.

Se utiliza la siguiente estructura para implementar el TAD descrito:

```
Módulo MateriaImpl implementa Materia {
    var primerParcial: Diccionario<Alumno, Nota>
    var segundoParcial: Diccionario<Alumno, Nota>
    var alumnos: Conjunto<Alumno>
    var aprobados: Conjunto<Alumno>
}
```

primerParcial y segundoParcial asocian a cada alumno con la nota que obtuvieron en cada examen. Alumnos son todos los alumnos que tienen la materia. aprobados son aquellos alumnos que aprobaron la materia, es decir, que tienen más de 6 en ambos exámenes.

Se pide:

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

TODOS (λ) ALUMNOS APARECEN EN PARCIALES

APROBADOS \rightarrow PP

APROBADOS \rightarrow SP

PP \wedge SP $> 6 \rightarrow$ APROBADOS.

ALUMNOS \rightarrow PP

ALUMNOS \rightarrow SP

PP \rightarrow ALUMNOS

SP \rightarrow ALUMNOS

APROBADOS \rightarrow ALUMNOS

NOTA: 0

Nota: entre 0 y 10

Castor los alumnos tienen las calificaciones aunque no hayan rendido.

Opelketa tienen los alumnos q tienen 6 o más en PRIMER, q 6 y más en SEGUNDO.

Rebelde tienen los q tienen alguno de los dos más de 6.

a) INVERP:

- Para todos alumnos en APROBADOS, tienen en PRIMERPARCIAL y SEGUNDOPARCIAL
low Clark q las NOTAS son > 6 . (3)
- Para tales alumnos en PP y SP es q las notas son > 6 , tienen en APROBADOS. (3)
- Para tales alumnos en ALUMNOS, están en PP y SP. (2)
- Para tales alumnos en PP tienen en ALUMNOS. (2)
- Para todos alumnos en SP tienen en ALUMNOS. (2)
- Para tales alumnos en APROBADOS tienen en ALUMNOS. (1)



Toda nota en PP o SP tiene que ser 0 ó 10.

ABS:

- Los Alumnos APROBADOS (TAD) están en APROBADOS (IMPL) (no hace falta que los alumnos que lo sea el inverso) (1)
- Los Alumnos APROBADOS (IMPL) están en APROBADOS (TAD) (1)
- Para cada Alumno en CALIFICACIONES, APARECE EL PRIMER ELEMENTO DE LA TUPLA EN PRIMERPARCIAL Y EL SEGUNDO EN SEGUNDOPARCIAL (NOTA 0 SI ESTUVO Ausente) (3)
- Para cada alumno en Primer parcial tiene en calificaciones los alumnos en el primer elemento de la tupla. (2)
- Para cada alumno en Segundo parcial tiene en calificaciones los alumnos en el segundo elemento de la tupla. (2)
- REPROBADOS son todos los alumnos que NO tienen un apellido y si es alumno. (4)
- Para estos alumnos que NO son APROBADOS tienen que ser REPROBADOS. (4)

PRED) APROBADOS VALIDOS (M: MATERIA IMPL){

($\forall A: ALUMNO$) ($A \in M. APROBADOS \rightarrow A \in M. ALUMNOS$) (1)

}

Por enunciado se vé que tienen todos en ambos

PRED) ALUMNOS EN PARCIALES VALIDOS (M: MATERIA IMPL){

($\forall A: ALUMNO$) ($A \in M. PRIMERPARCIAL \wedge A \in M. SEGUNDOPARCIAL \Rightarrow A \in M. ALUMNOS$) (2)

}

Alumnos > 6

PRED) ALUMNOS EN APROBADOS APROBARON PARCIALES (M: MATERIA IMPL){

(3)

✓

($\forall A: ALUMNO$) ($(A \in M. PRIMERPARCIAL \wedge M. PRIMERPARCIAL[A] > 6 \wedge A \in M. SEGUNDOPARCIAL \wedge M. SEGUNDOPARCIAL[A] > 6) \Leftrightarrow A \in M. APROBADOS$)

}

PRED) NOTAS VALIDAS (M: MATERIA IMPL){

($\forall A: ALUMNO$) ($A \in M. PRIMERPARCIAL \wedge A \in M. SEGUNDOPARCIAL \rightarrow L($

$M. PRIMERPARCIAL[A] \geq 0 \wedge M. PRIMERPARCIAL[A] \leq 10 \wedge$

$M. SEGUNDOPARCIAL[A] \geq 0 \wedge M. SEGUNDOPARCIAL[A] \leq 10)$

}

PRED INVREP(m : MATERIAIMPL){

APROBADOS VALIDOS (M)

\wedge

ALUMNOS EN PARCIALES VALIDOS (M)

\wedge

✓

ALUMNOS EN APROBADOS APRIBARON PARCIALES (M)

\wedge

NOTAS VALIDAS (M)

}

PRED ABS(M : MATERIA, M' : MATERIAIMPL){

ALUMNOS APROBADOS VALIDOS (M, M')

\wedge

ALUMNOS EN PARCIALES EN CALIFICACIONES (M, M')

\wedge

ALUMNOS EN CALIFICACIONES EN PARCIALES (M, M')

\wedge

REPROBADOS EN ALUMNOS PERSONA EN APROBADOS (M, M')

}

PRED ALUMNOS APROBADOS VALIDOS (M : MATERIA, M' : MATERIAIMPL){

($\forall A$: ALUMNO) ($A \in M$. APROBADOS $\Leftrightarrow A \in M'$. APROBADOS)

(1)

}

PRED ALUMNOS EN PARCIALES EN CALIFICACIONES (M : MATERIA, M' : MATERIAIMPL){ (2)

($\forall A$: ALUMNO) (($A \in M'$. PRIMERPARCIAL $\wedge A \in M'$. SEGUNDOPARCIAL $\wedge A \in M$. CALIFICACIONES) \rightarrow

M' . PRIMERPARCIAL[A] = M . CALIFICACIONES[A]₀ \wedge M' . SEGUNDOPARCIAL[A] = M . CALIFICACIONES[A]₁)

}

PRED ALUMNOS EN CALIFICACIONES EN PARCIALES (M : MATERIA, M' : MATERIAIMPL){ (3)

($\forall A$: ALUMNO) ($A \in M$. CALIFICACIONES $\wedge A \in M'$. PRIMERPARCIAL $\wedge A \in M'$. SEGUNDOPARCIAL \rightarrow

M . CALIFICACIONES[A]₀ = M' . PRIMERPARCIAL[A] \wedge M . CALIFICACIONES[A]₁ = M' . SEGUNDOPARCIAL[A])

}

PRED REPROBADOS EN ALUMNOS PERSONA EN APROBADOS (M : MATERIA, M' : MATERIAIMPL){ (4)

($\forall A$: ALUMNO) ($A \in M$. REPROBADOS $\Leftrightarrow A \in M'$. ALUMNOS $\wedge A \in M'$. APROBADOS)

✓

E1. Elección de estructuras (40 pts)

Se quiere implementar el TAD Agenda que modela una agenda semanal donde se registran actividades. Cada actividad tiene un identificador, un horario de inicio y uno de finalización. No puede haber dos actividades con el mismo identificador. Para simplificar, las actividades sólo pueden comenzar y terminar en horarios en punto (por ejemplo, 21:00 hs) y terminan en un horario posterior a su inicio. Tampoco pueden empezar y terminar en días diferentes. Para contar la cantidad de actividades que transcurren en un determinado horario no se debe tener en cuenta aquellas que finalizan en ese momento. En cada actividad se pueden agregar tags que permiten agrupar las distintas actividades por temáticas. Los tags tienen como máximo 20 caracteres.

Dadas las siguientes operaciones y de acuerdo a las complejidades temporales de peor caso indicadas, donde d es la cantidad de días registrados, hasta el momento y a la cantidad total de actividades, respectivamente:

- proc RegistrarActividad(inout ag: Agenda, in act: IdActividad, in dia: Día, in inicio: Hora, in fin: Hora)
 - Requiere: la hora de fin sea mayor que la de inicio y la actividad no está actualmente registrada.
 - Descripción: se agrega la actividad a la agenda, marcando en el día indicado la hora de inicio y finalización. 0
 - Complejidad: $O(\log(a) + \log(d))$
- proc VerActividad(in ag: Agenda, in act: IdActividad): struct<dia: Día, inicio: Hora, fin: Hora>
 - Requiere: la actividad debe estar registrada en la agenda.
 - Descripción: se devuelven el día y horario en que se realiza la actividad.
 - Complejidad: $O(\log(a))$.
- proc AgregarTag(inout ag: Agenda, in act: IdActividad, in t: Tag)
 - Requiere: la actividad está registrada en la agenda y aún no tiene registrado ese tag.
 - Descripción: se agrega el tag a la actividad indicada.
 - Complejidad: $O(1)$.
- proc HoraMásOcupada(in ag: Agenda, in d: Día): Hora
 - Requiere: el día indicado tiene al menos una actividad registrada.
 - Descripción: se devuelve la hora que tiene más actividades registradas.
 - Complejidad: $O(\log(d))$.
- proc ActividadesForTag(in ag: Agenda, in t: Tag): Conjunto<IdActividad>
 - Requiere: true.
 - Descripción: se devuelven las actividades que tienen registrado el tag t .
 - Complejidad: $O(1)$.

Se pide:

- Definir la estructura de representación del módulo AgendaImpl, que provea las operaciones solicitadas. Se debe explicar detalladamente qué información se guarda en cada parte y las relaciones entre ellas.
- Explicar cómo se cumplen las complejidades pedidas por cada operación, haciendo aclaraciones sobre aliasing. Indicar todas las suposiciones tomadas sobre la implementación de las estructuras mencionadas.
- Escribir el algoritmo de RegistrarActividad justificando detalladamente que se cumple la cota de complejidad requerida.

- ACTIVIDADES VAN A TENER ID (ÚNICOS), HI, HF.
- HI & HF TIENEN EN 00.
- HF > HI;
- DIA HI & DIA HF SON IGUAL PARA CADA ACT
- CONTAN ACT: SÓLO LAS QUE TIENEN ESA HF.
- TAGS TIENEN ACTIVIDADES
 - ↳ MAX 20 CHARS. ACORDADO, O(1).

me quiero
matar

Primeros OBSERVOS que tienen la complejidad más lógica.

- ACTIVIDADES POR TAG EN O(1) SIQUE TAG EN ACORDADO, EL O(1) VIENE DE OFÍ, SI UN ÚNICO. Otro modo AGREGAR TAG EN O(1)

Como el require es que no tiene poderes bien algún DEFINIMIENDO.

↳ CONSUMO LINEAL.

ENT: TAGS: DICCIONARIO DIGITAL (STRING, CONJUNTO LINEAL (IDACTIVIDAD)) ✓

- HORAS OCUPADA: $O(\log d)$, devolver las horas con más act (que tienen HF en ese hor). Pueden en log d tener intersección en una especie de AVL.

Por el lado del cono tienen horas en cada día las en ADDRS de 24 pos (24HS) y

para tanto el signo ACORDADO es $O(1)$.

OJO: NIA ESINT, ni fure string podna estan ACOVADO & SEN O(1)

ENT: DIAS: DICCIONARIOLOG<DIAS, ARRAY<INT>>

V.E.N ACTIVIDAD : $O(\log(A))$ Considera en ID numérico, no podes saber si es un AVL.

ACTIVIDADES: DICCIONARIO LOGICA DE ACTIVIDAD, TUPLA(DIA, HI, HF) >

• REGISTRATION ACTIVATION: $O(\frac{\log(A)}{\text{freq}} + \frac{\log(d)}{\text{sum freq HF}})$

↳ el reductores en los (A)

Der rechteckige Kasten ist ein Rechteck:

- Cada DÍA son CADA ACTIVIDADS duró EN DIAS.
 - La suma de ACT en determinado DÍA & MF duró en DIAS, hasta el inicio de la hora Conexión.
 - El número que este en cada elem de DIAS, es CANTIDAD en la suma de ACTIVIDADES o las horas y días.
 - Los ítems tienen ACTIVIDADES y existen en ACTIVIDADES.

En Tener se dividió en bolígrafos de los tipos siguientes:

NEQUIENTE: ACR NO ESTÁ.

```
PROC REGISTRARACTIVIDAD (INOUT A:AGENDA, IN iACTIVIDAD: INT, IN DIA:DIA, IN iINICIO:HORA, IN iFIN:HORA) {
```

VAR A: DICCIONARIO LOG<ACTIVIDAD, (DIA, HI, HF)> := A. ACTIVIDADES; // O(1)

VAR D: Diccionario<Dia, ARRAY<INT>> := A.DIAS; //O(1)

A. DEFINIR(iDACTIVIDAD, TUPLA(DIA, INICIO, FIN)) // O(lg A)

VAR ACTIVIDADES DI HORAS : ARRAY SINT>

IF ($D.$ ESTA(DIA) == TRUE) THEN // $O(\log d)$

ACTIVIDADES DIAHORA := D. OBTENER(DIA); // $O(\log d)$

ENDIF

ACTIVIDADES DIAHORA[i] := ACTIVIDADES DIAHORA[i] + 1 // $O(1)$

D. DEFINIR (DIA, ACTIVIDADES DIAHORA) // $O(\log d)$

}

Complejidad: $O(\log a + \log d)$

Ej. 1. Complejidad

a) Dado el siguiente algoritmo:

```
función ALGORITMOEXÓTICO(A: arreglo de int,
R: arreglo de tupla(val:int, reps:int))
n := LONG(A)
R[0] := (A[0], 1)
r := 0
i := 1
res := 0
mientras i < n hacer
    si A[i] == R[r].val entonces
        | R[r] := (R[r].val, R[r].reps + 1)
    en otro caso
        | j := 1
        | mientras j ≤ R[r].reps hacer
            |   | res := res + R[r].val * j
            |   | j := j + 1
            | fin
            | r := r + 1
            | R[r] := (A[i], 1)
        | fin
        | i := i + 1
    fin
devolver res
fin
```

Se pide responder lo siguiente:

i. ¿Cuál es la complejidad temporal del peor caso de ALGORITMOEXÓTICO? Describir el tipo de entradas que corresponden a este caso. Dar la cota más ajustada posible. Justificar.

ii. Hacer exactamente lo mismo que para el punto anterior, pero en mejor caso.

b) Dadas funciones $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{>0}$, decidir si las siguientes afirmaciones son verdaderas o falsas. En caso de que fueran verdaderas se deberá demostrarlas, mientras que si fueran falsas se debe mostrar un contraejemplo justificando claramente por qué contradice la afirmación.

i. Si $f(n) \in O(h(n)) \cap \Omega(g(n))$ entonces $h(n) \in \Omega(f(n))$ y $g(n) \in \Omega(f(n))$.

ii. Si para todo $n : \mathbb{N}$, $f(n) < g(n)$ entonces $O(f(n)) \cap \Omega(g(n)) = \emptyset$.

b) $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_{>0}$.

i) $f(n) \in (O(h(n)) \cap \Omega(g(n)))$

$\Rightarrow f(n) \in \Omega(f(n)) \wedge g(n) \in \Omega(f(n))$

$f(n)$ MAS CHIC $\in O(h(n))$

$h(n) \in \Omega(f(n)) \checkmark$



