

**Ejercicio 1.** Comparar la complejidad de los algoritmos de ordenamiento dados en la teórica para el caso en que el arreglo a ordenar se encuentre perfectamente ordenado de manera inversa a la deseada.

Consideremos que tengo  $[4, 3, 2, 1]$  ¿ Cuánto cuesta ordenar hasta  $[1, 2, 3]$ ?

Vemos Selection sort, insertion sort, merge y Quick.

1)  $\square$  Comis en el peor caso como meya con la complejidad en  $O(n^2)$  pues debes comparar elem actual con todos los demás e intercambiar si cumplen q. lo de otros pase al orden.  $\hookrightarrow$  hasta encontrar min.

- $[4, 3, 2, 1] \Rightarrow [4, 3, 2, 1] \Rightarrow [4, 3, 2, 1] \Rightarrow [4, 3, 2, 1]$   
 $\Rightarrow [1, 3, 2, 4]$
- $[1, 3, 2, 4] \Rightarrow [1, 3, 2, 4] \Rightarrow [1, 2, 3, 4]$   
 $\hookrightarrow$  Caso 2 era el min del grupo.

En este Caso, en  $O(n^2)$  pues el segundo ciclo bucle recorre el elem actual hasta el final para buscar el mínimo.

$\hookrightarrow$  El peor caso es q el min esté al final, pero Otra vez no lo esté, Tengs q hacer ese recorrido igual.

2) En I.S la ventaja q tenemos es q vamos hacia adelante pues comparar con los de arriba. En el buq q el array esté ordenado al revés da el mismo caso q tengs q recorrer todo los elementos (se los m paga mas q los n).

Ej: 1er PASO NAT

2do PASO:  $2D0 \xrightarrow{C} 1F0$

3er PASO:  $3D0 \xrightarrow{C} 2D0 \quad 3D0 \xrightarrow{C} 1F0$

SEGUNDO PASO:  $4T_0 \rightarrow 3T_0$        $3T_0 \rightarrow 2T_0$        $2T_0 \rightarrow 1T_0$

4º PASO:  $4T_0 \rightarrow 3T_0$        $3T_0 \rightarrow 2T_0$        $2T_0 \rightarrow 1T_0$

3) Merge sort si mal no recuerda se ponen los datos a la mitad hasta llegar al caso base ( $\log(n) = 1$ ).

Otra vez no importa el orden del merge, siempre será  $O(m \log m)$

4) Quick Sort: similar a MERGE SORT pero elegir un PIVOT el cual si lo elegimos mal puede llegar a ser  $O(m^2)$ . La idea del PIVOT es que los de la izq sean menores y los de la der mayores.

Hay varias formas de que la complejidad mejore, pero en Quick Sort se usa cuando el PIVOT es el ELEMENTO MÁS CERCANO O MÁS GRANDE TODO EL TIEMPO.

$$\begin{aligned} [4, 3, 2, 1] &\Rightarrow [4, 3, 2] + [1] + [] \Rightarrow [4, 3] + ([2] + []) + ([1] + []) \\ &\quad \uparrow \quad \uparrow \quad \text{PIVOT} \quad \text{PIVOT} \\ &\Rightarrow [4] + ([3] + [2]) + ([2] + [1]) + ([1] + []) \\ &\Rightarrow [1] + [2] + [3] + [4] \end{aligned}$$

**Ejercicio 2.** Imagine secuencias de naturales de la forma  $s = \text{Concatenar}(s', s'')$ , donde  $s'$  es una secuencia ordenada de naturales y  $s''$  es una secuencia de naturales elegidos al azar. ¿Qué método utilizaría para ordenar  $s$ ? Justificar. (Entiéndase que  $s'$  se encuentra ordenada de la manera deseada.)

$$s = s' ++ s''$$

$$s' = \text{NÚMEROS ORDENADOS}$$

$$s'' = \text{NÚMEROS AL AZAR}$$

¿Qué métodos utilizaría para ordenar  $s$ ?

IDEAS:

$s''$  pueden ser todos diferentes o  $s'$ .

$s''$  no están escritos al tambo  $s$ .

$s$  tiene errores.

$S^{11}$  podría tener ordenados (ser imposible).

BUCKET SORT: No tendría sentido a menor que explícitamente pedamos partir la lista en ORDENADO y NO-ORDENADO.

COUNTING SORT: Desconozco K, y si llegase a ser muy grande sería muy costoso.

RADIX SORT: Menor aún, que COUNTING.

SELECTION: Su complejidad es  $O(n^2)$ , no opone nada el que esté ordenado.

INSERTION: Podría ser rápido si tendría  $n'$  y  $n''$  separados, pero que si  $n''$  tiene el mío quisiera la complejidad sea  $O(n')$ .

QUICK SORT: Depende del PIVOT, no es estable y en PEOR CASO su complejidad es  $O(n^2)$ .

MERGE SORT: Es estable y el algoritmo divide AND CONQUIER una parte ya está ordenada. El costo en el peor caso es  $O(n \log(n))$ .

PROC ORDENAR (inout : ARRAY<INT>){

A := MERGE SORT(A);

RETURN A;

}

Ejercicio 3. Escribir un algoritmo que encuentre los  $k$  elementos más chicos de un arreglo de dimensión  $n$ , donde  $k \leq n$ . ¿Cuál es su complejidad temporal? ¿A partir de qué valor de  $k$  es ventajoso ordenar el arreglo entero primero?

K ELEMENTOS MÁS CHICOS de un array de long m donde k ≤ m.

Al me viene a la mente UN MIN-HEAP y sacar los  $k$  elementos q me piden. K es a lo mucho m.

regar en MIN-HEAP es  $O(1)$  por el BALANCEO de  $O(M)$ . Entonces sacar los  $k$  elementos (tardaría  $O(k \cdot \log(m))$ ) y si  $k$  es  $M$ , sum  $O(M \cdot \log(m))$

Quiero q si hay repeticiones los devuelva igual.

PROC SACARKMASCHICOS (IN A: ARRAY<T>, IN K: INT) : ARRAY<T> {

VAR AR: ARRAY<T> := NEW ARRAY<T>[1..K];

```

VAR MH.COLAPRIORIDAD<T>:= NEW COLAPRIORIDAD<T>(); //MIN-HEAP
VAR i:int := 0;
WHILE(i < A.LENGTH) DO //|A|=m => O(m.log(m))
    MH.ENCOLAR(A[i]); //O(log m)
    i++;
ENDWHILE

```

i:=0;

```

WHILE(i < k) DO //O(k.log(m))
    VAR E:T := MH.DESENCOLARMIN(); //O(log m)
    ARR[i] := E;
    i++;
ENDWHILE

```

RETURN ARR

}

Complejidad:  $O(m \cdot \log(m) + k \cdot \log(m)) \equiv O(m \log(m))$   $k$  es a lo sumo  $m$   
 (separados)

Algunas son L.E

Ejercicio 4. Se tiene un conjunto de  $n$  secuencias  $\{s_1, s_2, \dots, s_n\}$  en donde cada  $s_i$  ( $1 \leq i \leq n$ ) es una secuencia ordenada de naturales. ¿Qué método utilizaría para obtener un arreglo que contenga todos los elementos de la unión de los  $s_i$  ordenados. Describirlo. Justificar.

BUCKET: Sí, cada secuencia es ordenada.

COUNTING: Sí, se cuanta cuantos hay en el arreglo.

RADIX: Sí, reutiliza COUNTING.

SELECTION: No, queremos algo  $O(m^2)$  mi m-s-embree.

INSERTION: Podría servir, pero el peor caso es  $O(m^2)$ .

Concatenar las listas individualmente es del  $O(n^2)$  de L.E

PROC ORDENAR(IN S: ARRAY<LISTAENLAZADA<INT>>){

VAR i:int := 1;

VAR L: LISTAENLAZADA<INT>; = S[0];

WHILE ( $i < s.LENGTH$ ) DO //  $O(|s|)$

L.CONCATENAR( $s[i]$ ) //  $O(1)$

$i++$

END WHILE

VAR  $it := L.ITERADOR();$  //  $O(1)$

VAR  $TAMAÑO := L.LONGITUD();$  //  $O(1)$

VAR ARRAY<int> := NEW ARRAY<int>() {TAMAÑO};

$i := 0;$

WHILE ( $i < TAMAÑO$ ) DO //  $O(E)$

ARR[i] := it.SIGUIENTE(); //  $O(1)$

$i++$

END WHILE

ARR := MERGESORT(ARR); // ORDEN ASC  $\Rightarrow E \cdot \log(E)$

RETURN ARR;

}

Complejidad:  $O(|s| + E + E \cdot \log(E)) \equiv O(E \cdot \log(E))$  ✓

Ejercicio 5. Se tiene un arreglo de  $n$  números naturales que se quiere ordenar por frecuencia, y en caso de igual frecuencia, por su valor. Por ejemplo, a partir del arreglo [1, 3, 1, 7, 2, 7, 1, 7, 3] se quiere obtener [1, 1, 1, 7, 7, 7, 3, 3, 2]. Describa un algoritmo que realice el ordenamiento descrito, utilizando las estructuras de datos intermedias que considere necesarias. Calcule el orden de complejidad temporal del algoritmo propuesto.

ESTRUCTURAS

• Hay dos ordenaciones:

- 1: POR FREQ
- 2: POR VALOR

En este caso mismo VAL  $\Rightarrow$  MENOR CLAVE ANTES. No AV1 para ordenar por clave, y luego un algoritmo estable para ordenar por valor. Si el menor es igual, no cambia y se mantiene el orden.

```

PROC FREQ (inout A:ARRAY<INT>){
    VAR AVL: DICCIONARIO<INT, INT> := NEW DICCIONARIO<INT, INT>();
    VAR i: INT := 0;
    VAR j: INT := 0;
}

```

```

WHILE (i < A.LENGTH) DO
    VAR ESTA: BOOL := AVL.ESTA(A[i]); // O(log(A))
    VAR VAL: INT := 0;
    IF (ESTA == TRUE) THEN
        VAL := AVL.OBTENER(A[i]); // O(log(A))
    ENDIF;
    AVL.DEFINIR(A[i], VAL + 1); // O(log(A))
    i++;
ENDWHILE;

```

TUPAS  $\rightarrow$  A LOS VMS K = |A|

```

VAR B: ARRAY<INT, INT> := AVL.inorder(); // O(K) ORDEN X CLAVE
B := MERGESORT(B); // ORDENAR DESC POR SEGUNDA CLAVE // O(k · log(k))

```

```

VAR INDICE: INT := 0;
i := 0, j := 0;

```

```

WHILE (i < B.LENGTH) DO
    WHILE (j < B.LENGTH) DO
        A[INDICE] := B[i][j];
        INDICE++;
        j++;
    ENDWHILE;
    j := 0;
    i++;
ENDWHILE;

```

$O(A)$

```

    RETURN A;
}

```

Complejidad:  $O(A + K + K \cdot \log(K) + A) = O(A \cdot \log(A))$  ✓

**Ejercicio 6.** Sea  $A[1 \dots n]$  un arreglo que contiene  $n$  números naturales. Diremos que un rango de posiciones  $[i \dots j]$ , con  $1 \leq i \leq j \leq n$ , contiene una escalera en  $A$  si valen las siguientes dos propiedades:

1.  $(\forall k : \text{nat}) i \leq k < j \implies A[k+1] = A[k] + 1$  (esto es, los elementos no sólo están ordenados en forma creciente, sino que además el siguiente vale exactamente uno más que el anterior).
2. Si  $i < i$  entonces  $A[i] \neq A[i-1] + 1$  y si  $j < n$  entonces  $A[j+1] \neq A[j] + 1$  (la propiedad es *maximal*, es decir que el rango no puede extenderse sin que deje de ser una escalera según el punto anterior).

Se puede verificar fácilmente que cualquier arreglo puede ser descompuesto de manera única como una secuencia de escaleras. Se pide escribir un algoritmo para reposicionar las escaleras del arreglo original, de modo que las mismas se presenten en orden decreciente de longitud y, para las de la misma longitud, se presenten ordenadas en forma creciente por el primer valor de la escalera.

El resultado debe ser del mismo tipo de datos que el arreglo original. Calcule la complejidad temporal de la solución propuesta, y justifique dicho cálculo.

Por ejemplo, el siguiente arreglo

5	6	8	9	10	7	8	9	20	15
---	---	---	---	----	---	---	---	----	----

debería ser transformado a

7	8	9	8	9	10	5	6	15	20
---	---	---	---	---	----	---	---	----	----

Ayuda: se aconseja comenzar el ejercicio con una clara descripción en castellano de la estrategia que propone para resolver el problema.

$\Rightarrow [1, 2, 3]$

$$\underbrace{A[k+1]}_2 = \underbrace{A[k]}_1 + 1$$

$\forall i : i > 1 \text{ en } A[i] \neq A[i-1] + 1$

Lo primero que debes hacer es "DETECTOR" de grupos de escaleras.

Una vez que los grupos debes ordenar los "BLOQUES" más grande a la izq y luego, si algunos llegan a serlo lo mismo, mantener el orden original los mueres según qué escalera tiene el MENOR VALOR.

$\forall i : A[i+1] \neq A[i] + 1$  Termina el BLOQUE.

Lo que Tengo "Cien" orden para la operación de las escaleras.

$[20, 15, \underline{1, 2, 3}, 0, 1, 2]$  IDE: DETECTO BLOQUES Y LOS AGREGO

$[(\underline{1, 20}), (\underline{1, 15}), (\underline{3, 1}), (\underline{3, 0})]$

PROC ESCALERA(inout A:ARRAY<INT>){

VAR i:int:=0;

VAR BF:=1; //EL EN BLOQUE

VAR L: LISTAENLAZADA<(int,int)> := NEW LISTAENLAZADA<(int,int)>();

VAR MIN:int := A[0];

WHILE ( $i < A.LENGTH - 1$ ) // O(A)

IF (A[i] == A[i+1]) THEN

BE++;

ELSE

L.AGREGARATRAS((BE, MIN)); // O(1)

BE := 1;

ENDIF

i++;

ENDWHILE

LISTA AMÉGICO y ANDEGLA LISTA

VAR IT := L.ITERADOR();

VAR B: ARRAY<(int,int)> := NEW ARRAY<(int,int)>(); {L.LONGITUD();};

i := 0;

WHILE ( $i < L.LONGITUD()$ ) Do O( $\tau$ ) u ls nros O(A)

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

i++;

ENDWHILE

B := MERGESORT(B); // Por segunda COMP (min) Asc, O( $\tau \cdot \log(\tau)$ )

B := MERGESORT(B); // Por primera COMP (CE) Desc, O( $\tau \cdot \log(\tau)$ )

i := 0;

VAR INDICE: int := 0;

VAR j: int := 0;

WHILE ( $i < B.LENGTH$ ) Do

WHILE ( $j < B[i]$ ) Do

A[INDICE] := B[i] + j;

INDICE++;

j++;

a ls nros m [(4,1),(2,2)] = 11122 /

O(A)

ENDWHILE

$j := 0;$

$i++;$

ENDWHILE

RETURN A;

}

Complejidad:  $O(m + \cancel{C\tau} + C\tau \cdot \log(\cancel{C\tau}) + C\tau \cdot \log(C\tau) + m) \equiv O(m \cdot \log(m))$

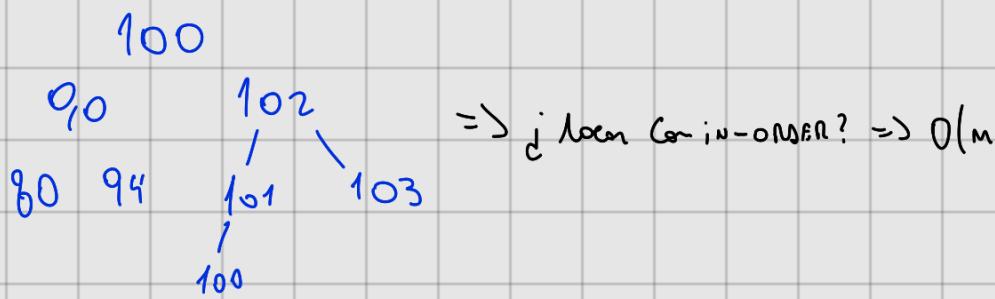
**Ejercicio 7.** Suponga que su objetivo es ordenar arreglos de naturales (sobre los que no se conoce nada en especial), y que cuenta con una implementación de árboles AVL. ¿Se le ocurre alguna forma de aprovecharla? Conciba un algoritmo que llamaremos *AVL Sort*. ¿Cuál es el mejor orden de complejidad que puede lograr?

Ayuda: debería hallar un algoritmo que requiera tiempo  $O(n \log d)$  en peor caso, donde  $n$  es la cantidad de elementos a ordenar y  $d$  es la cantidad de elementos distintos. Para inspirarse, piense en *Heap Sort* (no en los detalles puntuales, sino en la idea general de este último).

Justifique por qué su algoritmo cumple con lo pedido.

AVL; BÁNUCEAD, ES BUSQUEDA  $O(\log m)$

¿Mejoría total en un AVL, luego, mejoría en vector, luego ARRAY y luego MERGE SORT?



?

UV QUÍMICA

**Ejercicio 8.** Se tienen dos arreglos de números naturales,  $A[1..n]$  y  $B[1..m]$ . Nada en especial se sabe de  $B$ , pero  $A$  tiene  $n'$  secuencias de números repetidos continuos (por ejemplo  $A = [3333111888887771145555]$ ,  $n' = 7$ ). Se sabe además que  $n'$  es mucho más chico que  $n$ . Se desea obtener un arreglo  $C$  de tamaño  $n + m$  que contenga los elementos de  $A$  y  $B$ , ordenados.

1. Escriba un algoritmo para obtener  $C$  que tenga complejidad temporal  $O(n + (n' + m) \log(n' + m))$  en el peor caso. Justifique la complejidad de su algoritmo.

2. Suponiendo que todos los elementos de  $B$  se encuentran en  $A$ , escriba un algoritmo para obtener  $C$  que tenga com-

## METES 9 ODS DE B EN T UPAS A

$m$ : Max molar en A

$m'$ : Max molar en B

$m'$ : CANT deq en  $m$  (Bloques de los elem en  $m$ .  $m' < m$ )  $\Rightarrow [1,1,1] = m$   
 $[1,3] = m'$

Por complejidad, té q recorrer un MERGE para

$$m' + m \cdot \log(m' + m) = m \cdot \log(m).$$

Como el repartimiento se ordenan en C, A y B los q fuesen hacer las quiebras los molar de ambos arreglos en  $O(1)$  en algún lado, luego para la unión (costo  $O(n)$ ) y luego el MERGE y luego q A sea la otra vez.

Nó q el q qregar en  $O(1)$  lo vere de la lista enlazada q tiene el finito q entra en ULTIMO.

PROC ORDENAR (IN A: ARRAY<INT>, IN B: ARRAY<INT>): ARRAY<INT>

VAR  $l$ : LISTAENLAZADA := NEW LISTAENLAZADA((ELEM, CANT)>1); //  $O(1)$

VAR S: INT := 1; // CANTAP EN SEC

VAR i: INT := 0;

WHILE ( $i < A.LENGTH - 1$ ) DO //  $O(|A|) \geq O(m)$

IF ( $A[i] == A[i+1]$ ) THEN

S++;

ELSE

$l.AGREGARATRAS((A[i], S))$ ; //  $O(1)$

S := 1;

ENDIF

$i++$ ;

ENDWHILE

$i := 0;$

WHILE ( $i < B.LENGTH$ ) DO //  $O(|B|) = O(m)$

$l.AGREGARATRAS((B[i], 1));$  //  $O(1)$

$i++;$

ENDWHILE

// PASO L.E A ARRAY DE TUPLAS.

VAR C: ARRAY<(ELEM, CANT)> := NEW ARRAY<(ELEM, CANT)>(); //  $O(1)$

VAR IT := l.ITERADOR();

WHILE (IT.HAYSIGUIENTE() == TRUE) DO //  $O(|A| + |B|) = O(m + m)$

$C[i] := IT.SIGUIENTE();$

ENDWHILE

$C := MERGESORT(C);$  // Ordena por primera (cmp de forma ASC).  $O((m+m) \cdot \log(m+m))$

VAR D: ARRAY<INT> := NEW ARRAY<INT>{A.LENGTH + B.LENGTH};

VAR INDICE: INT := 0;

$i := 0;$

VAR j: INT := 0;

WHILE ( $i < C.LENGTH$ ) DO

WHILE ( $j < C[i]$ ) DO

$D[INDICE] := C[i];$

INDICE ++;

$j++;$

ENDWHILE

$j := 0;$

$i++;$

ENDWHILE

RETURN D;

}

[Red bracket from the inner WHILE loop to the outer WHILE loop.]

$O(m + m) = O(m + m)$

$\rightarrow m + m \leq 2m \Rightarrow$  Se suman los bloques

Complejidad:  $O(\cancel{m+m} + (\cancel{m'+m}) + (m'+m \cdot \log(m'+m)) + \cancel{(m+m)}) = O(m'+m \cdot \log(m'+m))$  ✓

Ejercicio 9. Considera la siguiente estructura para guardar las notas de un alumno de un curso:

```
alumno es tupla<nombre: string, género: GEN, puntaje: Nota>
donde GEN es enum{masc, fem} y Nota es un nat no mayor que 10.
```

Se necesita ordenar un arreglo(alumno) de forma tal que todas las mujeres aparezcan al inicio de la tabla según un orden creciente de notas y todos los varones aparezcan al final de la tabla también ordenados de manera creciente respecto de su puntaje, como muestra en el siguiente ejemplo:

Entrada	Salida
Ana F 10	Rita F 6
Juan M 6	Paula F 7
Rita F 6	Ana F 10
Paula F 7	Juan M 6
Jose M 7	Jose M 7
Pedro M 8	Pedro M 8

1. Proponer un algoritmo de ordenamiento `ordenaPlanilla(inout p: planilla)` para resolver el problema descrito anteriormente y cuya complejidad temporal sea  $O(n)$  en el peor caso, donde  $n$  es la cantidad de elementos del arreglo. Justificar.
2. Modificar la solución del inciso anterior para funcionar en el caso que GEN sea un tipo enumerado con más elementos (donde la cantidad de los mismos sigue estando acotada y el orden final está dado por el orden de los valores en el enum. Puedo hacer `for g in GEN`).
3. ¿La cota  $O(n)$  contradice el "lower bound" sobre la complejidad temporal en el peor caso de los algoritmos de ordenamiento? (El Teorema de "lower bound" establece que todo algoritmo general de ordenamiento tiene complejidad temporal  $\Omega(n \log n)$ .) Explique su respuesta.

POR NOTA, LUEGO X GEN

- NOTAS EN LOS ÓRDENES, MAS MÁS ALTO (11 POS).
- GEN EN LOS ÓRDENES (2 POS)

PROC ORDENAPLANILLA(inout A: PLANILLA) {

VAR AN: ARRAY<ALUMNO> := BUCKETSORT(A,11); // POR NOTA ASCENDENTE ARREGLO L.E. => O(M+k)

VAR VARONES: LISTAENLAZADA<ALUMNO> := NEW LISTAENLAZADA<ALUMNO>();

VAR MUJERES: LISTAENLAZADA<ALUMNO> := NEW LISTAENLAZADA<ALUMNO>();

VAR i, j: INT := 0;

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

VAR IT := AN[i].ITERADOR(); // GOMPA CON NOTA X

WHILE (IT.HAYSIGUIENTE == TRUE) DO

VAR ALUMNO: ALUMNO := IT.SIGUIENTE();

IF (ALUMNO<sub>1</sub> == "FEM") THEN

MUJERES.AGREGARATRAS(ALUMNO); // O(1)

ELSE

VARONES.AGREGARATRAS(ALUMNO); // O(1)

ENDIF

ENDWHILE

j++;

ENDWHILE

MUJERES.CONCATENAR(VARONES); // O(1)

```
iT := MUJERES.ITERADOR();
```

```
WHILE (iT.HAYSIGUIENTE() == TRUE) DO // O(m)
```

```
    A[i] := iT.SIGUIENTE();
```

```
i++;
```

```
ENDWHILE
```

```
RETURN A;
```

```
}
```

a lo sumo 10,ijo en la mitad máxima ✓

Complejidad:  $O(m+k) = O(m)$  ✓

**Ejercicio 10.** Suponer que se tiene un hipotético algoritmo de "casi" ordenamiento *casiSort* que dado un arreglo *A* de *n* elementos, ordena  $n/2$  elementos arbitrarios colocándolos en la mitad izquierda del arreglo *A*[1... $n/2$ ]. Los elementos no ordenados de *A* se colocan en la mitad derecha *A*[ $n/2 + 1 \dots n$ ].

1. Describir un algoritmo de ordenamiento para arreglos de *n* elementos (con *n* potencia de 2) utilizando el algoritmo *casiSort*.
2. Obtener la complejidad temporal en el peor caso del algoritmo dado en el punto anterior, suponiendo que *casiSort* tiene complejidad temporal  $\Theta(n)$ .
3. Cree que es posible diseñar un algoritmo para *casiSort* que realice  $O(n)$  comparaciones en el peor caso? Justifique su respuesta.

DATOS:

longitud PAR.

PARTO A la mitad ARRORIGUAL, GUARDANDO LA MITAD IZQ EN UNO Y LA DERECHA EN OTRO

que son volver en ARREGU ORIGINAL

```
PROC CASISORT(INOUT A:ARRAY<INT>){
```

```
VAR TAMAÑO : INT := A.LENGTH/2;
```

```
VAR B:ARRAY<INT>:= NEW ARRAY<INT>() {TAMAÑO};
```

```
VAR i := 0;
```

```
WHILE (i < TAMAÑO) DO:
```

```
    B[i] := A[i]
```

```
i++
```

```
ENDWHILE.
```

$O(m/2)$

CASISORT(A)

if A == 1

RETURN A

ELSE

CASISORT(iZQ)

SORT(DER)

MERGE(A)

$B := \text{MENGE SORT}(B)$ ; //ASC  $\Rightarrow O(\frac{m}{2} \cdot \log(\frac{m}{2}))$

$i := 0;$

WHILE ( $i < \text{RAN}$ ) DO //  $O(\frac{m}{2}) \Rightarrow m = 2^k$

$A[i] := B[i]$

$i++$

END WHILE.

RETURN A;

¿ how to do it  $\Theta(n)$ ?

}

$m \text{ par}$   
↑

?

Complejidad:  $O\left(\frac{m}{2} \cdot k\right) \equiv O(m \cdot k)$

Ejercicio 11. Sea  $A[1 \dots n]$  un arreglo de números naturales en rango (cada elemento está en el rango de 1 a  $k$ , siendo  $k$  alguna constante). Diseñe un algoritmo que ordene esta clase de arreglos en tiempo  $O(n)$ . Demuestre que la cota temporal es correcta.

Podemos usar Counting Sort para cada elem en A entre de 1 a K y K en una Contraseña.

PROC ORDENAR( $i:n$ ; $A:\text{ARRAY}\langle\text{INT}\rangle$ ) {

VAR  $K:\text{INT} := A[0]$ ;

VAR  $i:\text{INT} := 0$ ;

WHILE ( $i < k$ ) DO

IF ( $A[i] > k$ ) THEN

$K := A[i]$

ENDIF

$i++$

END WHILE.

PROC ORDENAR( $i:n$ ; $A:\text{ARRAY}\langle\text{INT}\rangle$ ; $k:i$ ) {

$A := \text{COUNTING SORT}(A, K)$  // OMENAR A SC

RETURN A;

}

↳ respuesta q Counting sort je tiene los desempeños óptimos con la menor complejidad.

?

A := COUNTINGSORT(A, K) // ORDENAR A SC

RETURN A;

}

?

Podríamos simplificar a  $O(m)$  para  $K$  es un elem del ARR  
Nótese q coincide?

Complejidad:  $O(m+k)$

Ejercicio 12. Se desea ordenar los datos generados por un sensor industrial que monitorea la presencia de una sustancia en un proceso químico. Cada una de estas mediciones es un número entero positivo. Dada la naturaleza del proceso se sabe que, dada una secuencia de  $n$  mediciones, al sumo  $\lfloor \sqrt{n} \rfloor$  valores están fuera del rango  $[20, 40]$ .

Proponer un algoritmo  $O(n)$  que permita ordenar ascendentemente una secuencia de mediciones y justificar la complejidad del algoritmo propuesto.

Algunos 3 DATOS:

- FUERA DE RANGO ( $< 20$ )
- FUERA DE RANGO ( $> 40$ )
- EN RANGO ( $[20, 40]$ )

OJO: El algoritmo debe ser  $O(n)$  y para (1) y (3) no  $\sqrt{m}$ . Delo ordenando con un algoritmo CUADRÁTICO para  $(\sqrt{m})^2 = m$

No podes usar MERGE, QUICK ni radix, ellos ordenan q tengan q usar límites delegados q no se operan en  $O(1)$ .

PROC ORDENARMEDICIONES(*in* A: ARRAY<INT>){

VAR MAYO: LISTAENLAZADA<INT>:= NEW LISTAENLAZADA<INT>(); // SELECTION

VAR ME20: LISTAENLAZADA<INT>:= NEW LISTAENLAZADA<INT>(); // SELECTION

VAR ER: LISTAENLAZADA<INT>:= NEW LISTAENLAZADA<INT>(); // COUNTING, si MAY o 40.

VAR i: INT := 0;

WHILE(i < A.LENGTH) DO //  $O(|A|)$

IF(A[i] < 20) THEN

ME20.AGREGARATRAS(A[i]); //  $O(1)$

ELSE IF(A[i] >= 40) THEN

```

ELSE IF (A[i] > 0) THEN
    MAYO. AGREGARATMAS(A[i]); // O(1)
ELSE
    ER. AGREGARATMAS(A[i]); // O(1)
ENDIF
i++;
ENDWHILE.

```

VAR ME20A: ARRAY<INT> := NEW ARRAY<INT>() {ME20.LONGITUD};

VAR MAYO: ARRAY<INT> := NEW ARRAY<INT>() {MAYO.LONGITUD};

VAR ERA: ARRAY<INT> := NEW ARRAY<INT>() {ER.LONGITUD};

```

VAR iT := ME20.ITERADOR();
WHILE (iT.HAYSIGUIENTE() == TRUE) DO // O(|ME20|)
    ME20A[i] := iT.SIGUIENTE();
ENDWHILE.

```

```

VAR iT := MAYO.ITERADOR();
WHILE (iT.HAYSIGUIENTE() == TRUE) DO // O(|MAYO|)
    MAYOA[i] := iT.SIGUIENTE();
ENDWHILE.

```

```

VAR iT := ER.ITERADOR();
WHILE (iT.HAYSIGUIENTE() == TRUE) DO // O(|ER|)
    ERA[i] := iT.SIGUIENTE();
ENDWHILE.

```

ME20A := SELECTIONSORT(ME20); // ASC => O( $\sqrt{n}^2$ ) = O(n)

MAYOA := SELECTIONSORT(MAYO); // ASC => O( $\sqrt{m}^2$ ) = O(m)

ERA := COUNTINGSORT(ERA, 40); // ASC => O(m)

MAYO : LISTAENLAZADA<INT> := NEW LISTAENLAZADA<INT>();

ME20 : LISTAENLAZADA<INT> := NEW LISTAENLAZADA<INT>();

ER : LISTAENLAZADA<INT> := NEW LISTAENLAZADA<INT>();

i := 0;

WHILE (i < MAYO) DO

MAYO.AGREGARATRAS(MAYO[i]); // O(1)

i++;

ENDWHILE

i := 0;

WHILE (i < ME20) DO

ME20.AGREGARATRAS(ME20[i]); // O(1)

i++;

ENDWHILE

i := 0;

WHILE (i < ERA) DO

ERA.AGREGARATRAS(ERA[i]); // O(1)

i++;

ENDWHILE

ME20.CONCATENAR(ERA); // O(1)



ME20.CONCATENAR(MAYO); // O(1)

IT := ME20.ULTIMO();

i := 0;

WHILE (IT.HAYSIGUIENTE() == TRUE) DO // O(|A|)

A[i] := IT.SIGUIENTE(); // O(1)

i++

ENDWHILE

RETURN A;

}

Complejidad:  $O(m)$

Ejercicio 13. Se tiene un arreglo  $A[1 \dots n]$  de  $T$ , donde  $T$  son tuplas  $\langle c_1 : \text{nat} \times c_2 : \text{string}[\ell] \rangle$  y los  $\text{string}[\ell]$  son strings de longitud máxima  $\ell$ . Si bien la comparación de dos  $\text{nat}$  toma  $O(1)$ , la comparación de dos  $\text{string}[\ell]$  toma  $O(\ell)$ . Se desea ordenar  $A$  en base a la segunda componente y luego la primera.

1. Escriba un algoritmo que tenga complejidad temporal  $O(n\ell + n \log(n))$  en el peor caso. Justifique la complejidad de su algoritmo.
2. Suponiendo que los naturales de la primer componente están acotados, adapte su algoritmo para que tenga complejidad temporal  $O(n\ell)$  en el peor caso. Justifique la complejidad de su algoritmo.

Los strings tienen ordenación  $\Rightarrow$  El ultimo char es 256.

Los números comparan en  $O(1)$   $\Rightarrow$  No ordenada. Hay que voltear regres.

M Canti elem en A.

PROC ORDENAR(*inout* A: ARRAY<int, STRING>){

A := RADIXSORT(A, 256); // ASC POR SEGUNDA COMP

A := MERGESORT(A); // ASC POR PRIMERA COMP

RETURN A;

}

Complejidad:  $O(m\ell + m \cdot \log(m))$

(?)  $\rightarrow$  ¿Qué ocurre si ordenas primero la base  
o viceversa? Igual me dice más se "en base de"  
luego  $\ell \times m$ .

b) Podemos usar Counting sort. Aquí nos da el número de veces en un k.

PROC ORDENAR(*inout* A: ARRAY<int, STRING>){

A := RADIXSORT(A, 256); // ASC POR SEGUNDA COMP

A := COUNTINGSORT(A); // ASC POR PRIMERA COMP  $\Rightarrow$  El k le busca el algoritmo

RETURN A;

}

Complejidad:  $O(m\ell + m) = O(m\ell)$

PARA CADA VAL DE:

$A = [1, 2]$   $k=2$

$A = [1, 2]$   $A = [2, 1]$

Ejercicio 14. Se tiene un arreglo  $A$  de  $n$  números naturales y un entero  $k$ . Se desea obtener un arreglo  $B$  ordenado de  $n \times k$  naturales que contenga los elementos de  $A$  multiplicados por cada entero entre 1 y  $k$ , es decir, para cada  $1 \leq i \leq n$  y  $1 \leq j \leq k$  se debe incluir en la salida el elemento  $j \times A[i]$ . Notar que podría haber repeticiones en la entrada y en la salida.

a) Implementar la función

```
proc ordenarMúltiplos(in A: array<nat>, in k: nat): array<nat>
```

que resuelve el problema planteado. La función debe ser de tiempo  $O(nk \log n)$ , donde  $n = A.length$  (el tamaño del arreglo).

b) Calcular y justificar la complejidad del algoritmo propuesto.

AVL-MIN-HEAP (M con los elementos originales)

"Ejecución de FOR"  $\forall$  los elem.

$$A = [2, 1] \quad K = 2$$

$$\begin{matrix} 2 & 1 \\ \downarrow 1 & \downarrow 2 & \downarrow 1.1 & \downarrow 1.2 \end{matrix}$$

$$|B| = |A| \cdot K$$

$$B = [2, 4, 1, 2]$$

$$A = [3, 2, 1] \quad K = 3$$

$$B = [3, 6, 9, 2, 4, 6, 1, 2, 3] \Rightarrow \text{ORDENAR MERGE, ASC } MK \cdot \log(MK)$$

PROC ORDENARMULTIPLICOS (in A: ARRAY<INT>, in K: INT): ARRAY<INT> {

VAR L: LISTAENLAZADA<INT> := NEW LISTAENLAZADA<INT>();

VAR i: INT := 0;

VAR j: INT := 1;

WHILE (i < A.LENGTH) DO //  $O(m) \in O(MK)$

WHILE (j < K+1) DO //  $O(k)$

L.AGREGARATRMS(A[i] \* j) //  $O(1)$

j++;

END WHILE

i++;

END WHILE

VAR B: ARRAY<INT> := NEW ARRAY<INT>() { A.LENGTH, K }

VAR iT := L.ITERADOR()

i := 0;

WHILE (iT.HAYSIGUIENTE()), DO

B[i] := iT.SIGUIENTE();

i++;

END WHILE

$B := \text{MERGESORT}(B); // O(mk \cdot \log(mk))$

↳ ME MATA.

RETURN B

}

Complejidad:  $O(mk + mk \cdot \log(mk)) \equiv O(mk \cdot \log(mk))$

La idea es no ordenar, las operaciones tener ordenadas.

$\log(m)$  el único q me garantiza orden en MIN-HEAP pero j'pono?

$$[2, 1] \quad k=2 \Rightarrow [2, 4 | 1, 2]$$

↓      ↓      ↓      ↓  
 k=1    k=2    k=1    k=2  
 i=0      i=1

El AVL debería poner el 2, pero luego el 1 xq es más gris

$$[2, 3, 1] \quad k=2 \Rightarrow [2, 4, 3, 6, 1, 2]$$

↓  
 [1, 2, 2, 4, 3, 6]  
 ↓  
 falso.

$[2, 3, 1, 2], k=2$ . 3 sublistas, voy cada índice.

$$\underbrace{[2, 4]}_k, \underbrace{[3, 6]}_k, \underbrace{[1, 2]}_k \quad [2, 4]$$

El MIN-HEAP hace  $[1, 2] [2, 4] [2, 4] [3, 6]$

$mk \cdot \log(m)$  "enamorados"

Ejercicio 16. Se tiene un arreglo  $A$  de  $n$  números naturales. Sea  $m := \max\{A[i] : 1 \leq i \leq n\}$  el máximo del arreglo. Se desea dar un algoritmo que ordene el arreglo en  $O(n \log m)$ , utilizando únicamente arreglos y variables ordinarias (i.e., sin utilizar listas enlazadas, árboles u otras estructuras con punteros).

a) Implementar la función

proc raroSort(in A: array<nat>): bool

(ANT DIGITOS)

que resuelve el problema planteado. La función debe ser de tiempo  $O(n \log m)$ , donde  $n = A.length$  (el tamaño del arreglo) y  $m = \max\{A[i] : 1 \leq i \leq n\}$ .

b) Calcular y justificar la complejidad del algoritmo propuesto.

CAN DIGITOS DEL NUM. TODOS SON  $\leq$  MAX EN  
CAN DIGITOS

- c) Calcula y justifica la complejidad del algoritmo propuesto.
- c) Hay al menos 3 formas de resolver este ejercicio, pensar las que se puedan, discutir luego con los compañeros las que encontraron ellos y si hace falta, consultar.

$$[9, 10, 1, 100, 450, 2] \quad n = 450$$

$n$  es el máximo

$$O(m \cdot \log(m))$$

( $\downarrow$ ) Cantidad de números

Primero Busco  $m$ . Precio  $O(m)$ .

"el logaritmo de un número es una hora en la medida de dígitos que tiene escrito en esa hora"

PROC RADIXSORT (in A: ARRAY<INT>): Bool {

VAR M: INT := A[0];

VAR i: INT := 0;

WHILE (i < A.LENGTH) DO

IF (A[i] > M) THEN

M := A[i]

ENDIF

i++;

ENDWHILE;

A := RADIXSORT(A,  $\log(m)$ ) //ASC

RETURN A;

}

Ejercicio 15. Dado un conjunto de naturales, diremos que un agujero es un natural  $x$  tal que el conjunto no contiene a  $x$  y sí contiene algún elemento menor que  $x$  y algún elemento mayor que  $x$ .

Diseñar un algoritmo que, dado un arreglo de  $n$  naturales, diga si existe algún agujero en el conjunto de los naturales que aparecen en el arreglo. Notar que el arreglo de entrada podría contener elementos repetidos, pero en la vista de conjunto, no es relevante la cantidad de repeticiones.

a) Implementar la función

```
proc tieneAgujero?(in A: array<nat>): bool
```

que resuelve el problema planteado. La función debe ser de tiempo lineal en la cantidad de elementos de la entrada, es decir,  $O(n)$ , donde  $n = A.length$  (el tamaño del arreglo).

b) Calcular y justificar la complejidad del algoritmo propuesto.

-> Lo lográs para cada x tiene uno

en  $O(n^2)$

$[1, 1, 4, 20, 10] \Rightarrow$  PASSA A L.F. S,N REP  $\Rightarrow O(n)$

$[1, 4, 20, 10] \Rightarrow$  BOSCS ORDENAR EN  $O(n)$

$[1, 4, 10, 20] \Rightarrow$  JUG. ORDENS EN EL C. S.O. MESSON EN  $O(1)$   
PER COM  $O(n)$ .

UNES COL. PRIORITAT UNES EXCEPCIONS PER EN LOG N OPERACIONS + UNES M. LOG(n).

¿POTS PUNTAR ORDENAR EN  $O(n)$ ?

