

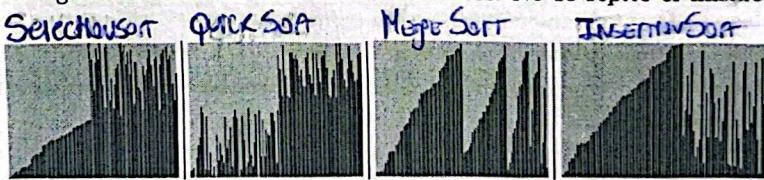
Algoritmos y Estructuras de Datos
Segundo Parcial – Miércoles 25 de Junio de 2025

#Orden	Libreta	Apellido v Nombre	E1	E2	E3	E4	E5	Nota Final
	10012		10	20	25	20	20	95

- Es posible tener una hoja (2 carillas), escrita a mano, con las anotaciones que se deseen, más los dos apuntes de la materia
- 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 y al menos 2 preguntas teóricas correctas

E1. Teórico (10 pts) (EN LA HOJA)

- a. **Diseño.** Elija la/s opciones correctas entre las siguientes afirmaciones que hablan sobre el invariante de representación:
- i. Se puede programar para validar la correctitud de la representación.
 - ii. Se debe programar para validar la correctitud de la representación.
 - iii. No tiene sentido programarlo porque no es una función inyectiva.
 - iv. No tiene sentido programarlo porque su dominio es en el mundo abstracto.
- b. **Diseño.** Elija la/s opciones correctas entre las siguientes afirmaciones que hablan sobre la función de abstracción:
- i. Determina qué operaciones públicas tendrá el TAD.
 - ii. Verifica que los valores del TAD cumplan ciertas propiedades lógicas.
 - iii. Define cómo se representa internamente el TAD usando estructuras básicas.
 - iv. Relaciona el estado interno de una representación con el valor abstracto que modela.
- c. **Diseño.** Determine si las siguientes afirmaciones son verdaderas o falsas:
- i. El invariante vale en la pre pero no en la post de todas las funciones de la interfaz.
 - ii. El invariante vale en la pre y en la post de todas las funciones de la interfaz.
 - iii. El invariante vale en la pre y en la post de todas las funciones auxiliares de la implementación.
 - iv. El invariante vale durante la ejecución de las funciones de la interfaz.
- d. **Sorting.**
- i. Cada una de las figuras a continuación muestra el estado de un algoritmo de ordenamiento diferente capturado en un momento intermedio de su operación sobre un arreglo. Usa tu conocimiento de los algoritmos para decir cuál es el algoritmo usado en cada uno de los casos. No se repite el mismo algoritmo en dos de las figuras.



ii. ¿Alguno de los algoritmos anteriores es naturalmente estable? De ser así, nombrarlo.

- e. **Complejidad.** Determine si las siguientes afirmaciones son verdaderas o falsas:
- i. Si f y g son funciones tales que $f \in O(g)$ y $g \in \Omega(f)$, entonces $f \in \Theta(g)$.
 - ii. Si la complejidad del mejor caso de un algoritmo es $f(n)$, entonces el número de pasos que el algoritmo realiza, cualquiera sea la entrada, es $\Omega(f)$.
 - iii. Si la complejidad de peor caso de un algoritmo es $f(n)$, entonces el número de pasos que el algoritmo realiza, cualquiera sea la entrada, es $\Theta(f)$.
 - iv. Un algoritmo con complejidad asintótica exponencial es menos eficiente para todas las entradas posibles que uno con complejidad asintótica cuadrática.

E2. Complejidad (20 pts)

- A. Elegir las opciones correctas.

- | | | |
|--|---|--------------------------------------|
| a. $n \log(n) + n^d \in O(n^2)$, si d es: | b. $n \log(n) + n^d \in \Theta(n^d)$, si d es: | c. $n^d + n^k \in \Omega(n^k)$, si: |
| i. 0 | i. 0, 1 ó 2 | i. $d = k$ |
| ii. 1 | ii. Cualquier $d > 3$ | ii. $d < k$ |
| iii. 2 | iii. 2 | iii. $d > k$ |
| iv. Cualquier número $\mathbb{R}_{>0}$ | iv. Cualquier número $\mathbb{R}_{>0}$ | iv. $d = 0$ |

- B. Determinar y explicar con palabras la complejidad del peor y mejor caso del siguiente algoritmo. Describir y dar un ejemplo de un valor de entrada para cada caso.

```

1 func SumaVecinosAPositivos (s : Array < int >):
2   i ← 0;
3   while i < |s| do
4     if s[i] < 0 then
5       skip;
6     else
7       suma ← s[i];
8       j ← i + 1;
9       while j < |s| do
10      suma ← suma + s[j];
11      j ← j × 2;
12      s[i] ← suma;
13   i ← i + 1;

```

E3. Rep&Abs (25 pts)

Persona, Mesa es \mathbb{Z}

```

TAD Elecciones {
  obs padron: dict<Persona, Mesa>
  obs votaron: conj<Persona>
  obs votaron: dict<Persona, Z> VotosPorCandidato
  proc iniciar(in cs : conj<Persona>, in vs : dict<Persona, Mesa>) : Elecciones
    // Comienzan las elecciones con los candidatos de cs con 0 votos y el conjunto de votaron de vs vacío.
    // Los candidatos pueden ser votantes
  proc votar(inout e : Elecciones, in v : Persona, in m : Mesa, in c : Persona)
    // El votante v está registrado en la mesa y todavía no había votado y el candidato c compite.
    // Se registra el voto a favor del candidato en e.votaron y e.votosPorCandidato
  proc participación(in e : Elecciones) : R
    // Se devuelve el porcentaje de votantes que ya sufragaron
  proc primerLugar(in e : Elecciones) : Persona
    // Se devuelve el candidato que va ganando. En caso de empate se devuelve cualquiera
    // de los que vayan primero
}

```

```

Módulo EleccionesImpl implementa Elecciones <
var candidatos: Conjunto<Persona>
// Todos los candidatos que compiten en la elección
var votantesPorMesa: Diccionario<Mesa, conj<Persona>>
// Tiene asociada a cada mesa con los votantes registrados en ella
var ranking: ColaDePrioridadMax<Tupla<Persona, int>>
// Cola de prioridad que tiene en la primera componente el candidato y en la segunda la cantidad de
// votos que tiene hasta el momento (todos > 0). Se ordenan según la cantidad de votos
var sufragaron: Conjunto<Persona> // Todos los votantes que ya emitieron su voto
var empadronados: Conjunto<Persona> // Todos los votantes registrados en el padrón
>

```

- a) Indique si cada una de las sentencias propuestas pertenecen al invariante de representación para la estructura propuesta
- Todo votante de sufragaron pertenece al conjunto asociado a alguna mesa de votantesPorMesa
 - Todo votante que pertenece al conjunto asociado a alguna mesa de votantesPorMesa también pertenece a sufragaron
 - Los conjuntos asociados a todas las mesas registradas en votantesPorMesa son disjuntos entre sí

- iv) Para cada tupla de ranking su primer elemento pertenece a candidatos ✓✓
- v) Para cada tupla de ranking su segundo elemento no es negativo ✓
- vi) Los conjuntos de votantes empadronados y candidatos son disjuntos F✓
- vii) Todo votante de empadronados está presente en sufragaron F
- viii) La unión de todos los conjuntos de votantes asociados a cada mesa de votantesPorMesa es igual a empadronados ✓

b) Escribir la función de abstracción en lógica de primer orden

E4. Diseño (25 pts)

Como saben que somos muy buenos haciendo nuestro trabajo, nos piden ayuda desde un Museo para poder organizar las colecciones que poseen. En cada colección se registran las piezas que serán expuestas al público en algún momento. Una vez registradas, un curador analiza cada obra para poder determinar el año de su creación entre otras cosas. Los especialistas saben que las únicas obras realmente valiosas para sus colecciones datan aproximadamente desde el 4000 a.C hasta el 2025 d.C. Finalmente, es de interés para el museo conocer cuáles son las obras que pertenecen a un año particular y cuál es la obra más antigua de cada una de sus colecciones para poder hacer difusión y atraer al público. El sistema deberá implementar las siguientes operaciones en la complejidad pedida, donde c es la cantidad de colecciones registradas, p es la cantidad de piezas. Colección, Pieza, Año son int.

- proc AbrirMuseo() : Museo
 - Descripción: Inicia el museo sin colecciones ni piezas registradas
 - Complejidad: $O(1)$
- proc PrepararColección(inout m : Museo, in c : Colección, in ps : Conjunto(Pieza))
 - Descripción: Se registra la colección con las piezas recibidas en ps sin las etiquetas de los años
 - Requiere: El conjunto contiene al menos una pieza
 - Complejidad: $O(\log(c) + p \cdot \log(p))$
- proc RegistrarPieza(inout m : Museo, in c : Colección, in p : Pieza, in a : Año)
 - Descripción: Se registra el año de la pieza
 - Requiere: La pieza está registrada en la colección y el año está en el rango indicado
 - Complejidad: $O(\log(c) + \log(p))$
- proc PiezasPorAño(in m : Museo, in a : Año) : Conjunto(Pieza)
 - Descripción: Devuelve las piezas creadas en ese año
 - Complejidad: $O(1)$
- proc KPiezasMásAntiguaPorColección(in m : Museo, in c : Colección, in k : int) : Conjunto(Pieza)
 - Descripción: Devuelve las k piezas más antiguas de la colección
 - Requiere: k toma valores entre 0 y la cantidad de piezas en la colección
 - Complejidad: $O(\log(c) + k \cdot \log(p))$

Se pide:

- Plantear la estructura de representación del módulo Museo que permita cumplir con las complejidades requeridas. Justificar con palabras de forma concisa las operaciones PrepararColección, PiezasPorAño y KPiezasMásAntiguaPorColección.
- Escribir el algoritmo de la operación RegistrarPieza. Justifique detalladamente que se cumple la complejidad obtenida.

E5. Sorting (20 pts)

Desde un reconocido sitio web de calificación de producciones audiovisuales, nos contactaron para proveerles un algoritmo de ordenamiento más eficiente para su sección de Anime. En esta sección contamos con una clasificación dividida en dos modalidades: la cantidad de *likes* que tienen y el puntaje asignado a cada serie animada. Los puntajes se representan con números enteros que van de 0 a 1000, mientras que los *likes* se representan con un entero positivo. Además, nos dicen desde el sitio que dentro de la entrada solamente \sqrt{n} elementos tienen más de 5000 *likes*, siendo n la cantidad de series registradas en esta categoría. Nuestra entrada será de un arreglo de triples compuestas por el nombre de la serie en formato *string*, su puntaje y la cantidad de *likes* (por ejemplo, <“Pokemon”, 974, 102433>). Se busca ordenar primero por el puntaje de forma descendente (de mayor a menor) y en caso de empate de forma descendente por los *likes*. Se nos pide dar un algoritmo que ordene según el criterio presentado en complejidad $O(n)$. Justifique correctamente las decisiones tomadas y las complejidades obtenidas de los algoritmos utilizados.

Ejemplo

Entrada: [⟨“Dragon Ball”, 960, 8000⟩, ⟨“One Piece”, 950, 3000⟩, ⟨“Attack on Titan”, 950, 7000⟩,
 ⟨“Naruto”, 800, 4500⟩, ⟨“Bleach”, 800, 6000⟩, ⟨“Pokemon”, 974, 102433⟩]

Salida: [⟨“Pokemon”, 974, 102433⟩, ⟨“Dragon Ball”, 960, 8000⟩, ⟨“Attack on Titan”, 950, 7000⟩,
 ⟨“One Piece”, 950, 3000⟩, ⟨“Bleach”, 800, 6000⟩, ⟨“Naruto”, 800, 4500⟩]

Ej) Teórico

- a) Se puede programar para validar la correctitud de la representación ✓
- b) Recauda el estado interno de una representación con el valor abstracto que modela ✓
- c) El invariante vale en la pre y en la post de todas las funciones de la interfaz ✓
- d) EN orden: Selection Sort, Quick Sort, Merge Sort, Insertion Sort ✓
 - II Merge Sort e Insertion Sort son estables. ✓
- e) Falso, Verdadero, Falso, ~~Falso~~ Falso ✓

E2) Complejidad

- a) Opciones correctas: i, ii, iii. D puede ser 0, 1, 2. ✓
 b) ~~Algunos, pero solo algunos~~ QUALQUIER d > 3 ✓
 c) TODAS SON CORRECTAS

B) Explicar con palabras la complejidad del peor y mejor caso.

Func Suma Vecinos A Positivos (S: Array < INT >):

```

i ← 0
while i < |S| do
    if S[i] < 0 then
        skip
    else
        SUMA ← S[i]
        j ← i + 1
        while j < |S| do
            SUMA ← SUMA + S[j]
            j ← j × 2
        S[i] ← SUMA
    i ← i + 1
  
```

- ~~Algunos~~ Vemos que este algoritmo tiene dos ciclos: El primero recorre SIEMPRE LA TOTALIDAD DEL ARREGLO. PARA CADA N (COMPLEJIDAD O(N)) VAMOS A VER EL VALOR QUE GUARDA S EN ESA POSICIÓN. SI TENEMOS SUENO EN CUANTO A TIEMPOS, EL VALOR VA A SER NEGATIVO Y SE VA A REALIZAR UN SKIP. POR OTRO LADO, SI EL VALOR ES MAYOR A CERO, VAMOS A IMPLICAR UNA SEGUNDA VARIABLE PARA UN SEGUNDO CICLO. ESTE WHILE SE EJECUTARÍA $\log(N)$ VECES, PUES VEMOS QUE EN CADA ITERACIÓN EL ÍNDICE j SE ESTÁ MULTIPLICANDO POR DOS.

Conclusión: PARA UNA ENTRADA, EL MEJOR CASO SERÍA QUE TODOS LOS ELEMENTOS SEAN MENORES A CERO, QUE POR LO QUE VENIOS TENDRÍA UNA COMPLEJIDAD DE O(N).

EL PEOR CASO SERÍA QUE SEAN TODOS POSITIVOS, QUE HABRÍA QUE PARA CADA ELEMENTO SE EJECUTE EL SEGUNDO WHILE Y MOS DANIA UNA COMPLEJIDAD DE O(N · log N).

UN EXEMPLO PARA EL MEJOR CASO SERÍA UNA ENTRADA S = [-2, -1, -4, -12] -

PARA EL PEOR CASO, UNA ENTRADA PUEDE SER S = [2, 4, 8, 3, 1].

E3) Prep n Abs.

Paso en lenguaje observadores y variar

TAD Elecciones {

obs padron: dict < Persona, Mesa >

obs VOTARON: Conj < Persona >

obs VOTOS POR CANDIDATO: dict < Persona, Z >

}

Modulo Elecciones Impl Implementa Elecciones <

VAR candidatos: Conj < Persona >

VAR VOTANTES POR MESA: Diccionario < Mesa, Conj < Persona > >

VAR ranking: GA De Prioridad MAX < Tupla < Persona, INT > >

VAR SUSPACION: Conjunto < Persona >

VAR EMPADRONADOS: Conjunto < Persona >

>

2) Todos respondidos EN LA IMPRESIÓN.

B) Escribir función de abstracción

// ME ANOTO RELACIONES ENTRE VARIABLES Y OBSERVADORES ANTES

- PARA toda persona EN candidatos, persona ES Elecciones.CLAVES(padron) y tambien a Elecciones.CLAVES(VOTOSPORCANDIDATO)
- PARA toda mesa EN VOTANTESPORMESA, SI UNA PERSONA PERTENECE A SU CONJUNTO ENTONCES Elecciones.padron[persona] = mesa.
- PARA cada persona EN LAS TUPLAS DE RANKING, persona ES UNA KEY de Elecciones.VOTOSPORCANDIDATO. Ademas, NO SOLO ES UNA KEY SINO QUE DEBE EXISTIR ESE TUPLA <CLAVE, VALOR> EN VOTOSPORCANDIDATO.
- Toda persona EN SUSPACION, PERTENECE A VOTARON Y TAMBIEN AL REVES
- Toda persona EN EMPADRONADOS, ES UNA KEY DEL padron del TAD.

(Escrito EN LA OTRA CARTILLA)

Pred Abs (TAO: ELECCIONES, IHp: Elecciones Imp) =
 $(\forall p: z)(p \in \text{Imp.CANDIDATOS} \Leftrightarrow (p \in \text{TAO.CLAVES(padron)} \wedge p \in \text{TAO.CLAVES(Votos Por CANDIDATO}))$
^
 $(\forall n: z)(n \in \text{IHp.CLAVES(VOTANTES POR MESA}) \Leftrightarrow (\forall p: z)(p \in \text{IHp.VOTANTES POR MESA}[p] \wedge$
 $\rightarrow p \in \text{TAO.CLAVES(PADRON)} \wedge \overset{\text{no}}{\underset{\text{no}}{\wedge}} \text{padron}[p] = n))$
^
 $(\forall t: \text{tupla} < \text{IND}, \text{IND} >)(t \in \text{IHp.RANKING} \Leftrightarrow t_0 \in \text{TAO.CLAVES(Votos Por CANDIDATO}) \wedge$
 $\text{TAO.Votos Por CANDIDATO}[t_0] = t_1)$
^
 $(\forall p: z)((p \in \text{Imp.SUFRAGACION} \Leftrightarrow p \in \text{TAO.VOTACION}) \wedge (p \in \text{Imp.ELECTORADOS} \Leftrightarrow p \in \text{TAO.CLAVES(PADRON)})$

Algoritmos y Estructuras de Datos - Segundo PARCIAL -

Hoja 4

E5) Sorting

TENEMOS $\langle \text{Nombre}, \text{Puntaje}, \text{Likes} \rangle$ TENEMOS \sqrt{N} elementos con MÁS DE 5000 Likes
Acordeado de 0 A 100

⇒ Ordenar por el puntaje de forma descendente, y desempate descendente por likes.

IDEA. Voy a separar dos grupos, uno con los de +5000 likes que se que es \sqrt{N} MAXIMO y uno con el resto. Esto es $O(N)$ pues miro cada elemento y lo decido.

⇒ Uso INSERTION SORT EN los de +5000 likes, pues su complejidad es EN EL PEOR CASO $O(N^2)$
lo que nos deja $O(\sqrt{N}^2) = O(N)$

Luego, se que el otro grupo está acorreado por 5000 en likes, si hago Radix Sort
El peor caso es $O(N \cdot 4) = O(N)$
dígitos

Ahora concateno ambos arrays y tengo todo ordenado por likes.

Debo ordenar por puntaje con un algoritmo estable y me queda finalizado.

~~Porque Insertion Sort es estable. De puntajes distintos pases sus valores sin cambiarlos.~~

Finalizo con Radix Sort con complejidad EN EL PEOR CASO $O(N+100) = O(N)$.
por Puntaje Ascendente

⇒ Function Ordenar (S: Array $\langle \text{Nombre}, \text{Puntaje}, \text{Likes} \rangle$) AUXILIARES DENTRO.

Lista Mayores = lista vacia (, // O(1)

Lista Menores = lista vacia () // O(1)

for i=0 to tam(S-1)

| | S[i].Likes > 5000

| | Mayores. Agregar Array(S) // O(1)

else

| | Menores. Agregar Array(S) // O(1)

| O(N)

Array May = ListToArray(Mayores) // O(N)

Array Men = ListToArray(Menores) // O(N)

INSERTION SORT (May) // DESCENDENTE POR LIKES O(N)

RADIX SORT (Men) // DESCENDENTE POR LIKES O(N)

Array res = Combinar (May, Men) // O(N)

RADIX SORT (res) // DESCENDENTE POR PUNTAJE O(N)

return res;

function $\text{list_to_array}(L: \text{list})$: Array

$\text{res} = \text{new Array}(L.\text{length})$

 for ($i=0, i < \text{res.length}, i++$)

$\text{res}[i] = L[i]$

 end

 return res

function $\text{concatenate}(a, b: \text{Array})$: Array

$\text{array_rer} = \text{new Array}(a.\text{length} + b.\text{length})$

 for ($i=0, i < a.\text{length}, i++$)

$\text{rer}[i] = a[i]$

 end

 for ($i=a.\text{length}, i < a.\text{length} + b.\text{length}, i++$)

$\text{rer}[i] = b[i]$

$j++$

 end

 return rer

E4) Diseño

IDEAS: TENEMOS AÑO ACTUAL Y DEBEMOS DEVOLVER EN $O(1)$ LAS PIEZAS POR AÑO,
POR LO QUE AÑO UN DICCIONARIO DIGITAL \langle AÑO, CONJUNTO LOG \langle PIEZAS \rangle \rangle

VIENDO LA COMPLEJIDAD DE K PIEZAS POR COLECCIÓN, TENEMOS QUE ARMARLO UN DICC LOG
DE COLECCIÓN A GRÁFICA DE PRIORIDAD MAX \langle PIEZA, AÑO \rangle , DESDEMANDANDO K VECES NOS DA
 $O(\log c + k \cdot \log p)$

\Rightarrow YA TENEMOS ANTIGUAS POR COLECCIÓN: DICCIONARIO LOG \langle COLECCIÓN, GRÁFICA DE PRIORIDAD MAX \langle PIEZA, AÑO \rangle \rangle
PIEZAS POR AÑO: DICCIONARIO DIGITAL \langle AÑO, CONJUNTO LOG \langle PIEZAS \rangle \rangle

Luego, en registrar PIEZA ~~en la colección~~ creamos una tupla con el año.

Nos cuesta $O(1)$ crearla, y la engañamos EN UN HÉAP QUE NOS CUESTA $O(p)$

PARA ESTO NECESITAMOS UNA VARIABLE PIEZAS POR COLECCIÓN: DICCIONARIO LOG \langle COLECCIÓN, CONJUNTO LOG \langle PIEZAS \rangle

TENEMOS ENTONCES: VAR ANTIGUAS POR COLECCIÓN \langle COLECCIÓN, GRÁFICA DE PRIORIDAD MAX \langle PIEZA, AÑO \rangle // POR AÑO
~~PIEZAS POR COLECCIÓN~~ : DICC LOG

VAR PIEZAS POR AÑO: DICCIONARIO DIGITAL \langle AÑO, CONJUNTO LOG \langle PIEZA \rangle \rangle

VAR PIEZAS POR COLECCIÓN: DICCIONARIO LOG \langle COLECCIÓN, CONJUNTO LOG \langle PIEZA \rangle \rangle

Abrir Museo será EN $O(1)$ PUES úNICAMENTE INICIALIZA ESTRUCTURA

EN PREPARAR COLECCIÓN, MIGARÁ EL CONJUNTO LOG EN $O(c)$ POR EL DICCIONARIO Y PARA CADA ELEMENTO LO MIGRAMOS.

b) función registrar PIEZA (INOUT M: Museo, IN C: Colección, IN P: Pieza, IN A: Año)

TUPLA <INT, INT> PIEZA AÑO = <PIEZA, AÑO> // $O(1)$

M. ANTIGUAS POR COLECCIÓN. OBTENER(C). ENGAÑAR(PIEZA AÑO) // $O(\log c) + O(\log p)$

M. PIEZAS POR AÑO. OBTENER(AÑO) = M. PIEZAS POR AÑO. OBTENER(AÑO). AGREGAR(P) // $O(1)$ OBTENER
 $O(p)$ ESCRIBIR

Falta explicar cómo se cumplen los complejidades para
PREPARAR COLECCIÓN, PIEZAS POR AÑO Y LA PIEZAS + ANTIGUAS