Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos 2'2022

Interrogación 1

9 de septiembre de 2022

Condiciones de entrega. Debe entregar solo 3 de las siguientes 4 preguntas.

Puntajes y nota. Cada pregunta tiene 6 puntos más un punto base. La nota de la interrogación será el promedio de las notas de las 3 preguntas entregadas.

Uso de algoritmos. De ser necesario, en sus diseños puede utilizar llamados a cualquiera de los algoritmos vistos en clase. No debe demostrar la correctitud o complejidad de estos llamados, salvo que se especifique lo contrario.

1. Análisis de algoritmos

Para ordenar una secuencia de datos implementada como arreglo A se propone el siguiente algoritmo `CocktailShakerSort` definido junto a sus subrutinas

```
input : Arreglo A[0, ..., n − 1]
CocktailShakerSort (A):
1   i ← 0
2   f ← n − 2
3   while i ≤ f :
4       k ← LeftRightShake(A, i, f)
5       f ← k − 1
6       t ← RightLeftShake(A, i, f)
7       i ← t + 1
```

```
input : Arreglo A[0, ..., n − 1], índices i, f
LeftRightShake (A, i, f):
1   j ← i
2   for m = i ... f :
3       if A[m] > A[m + 1] :
4           A[m] ⇄ A[m + 1]
5           j ← m
    return j

RightLeftShake (A, i, f):
1   j ← f
2   for m = f ... i :        ▷ loop decreciente
3       if A[m] > A[m + 1] :
4           A[m] ⇄ A[m + 1]
5           j ← m
    return j
```

(a) [1 pto.] Demuestre que los tres algoritmos mostrados terminan en tiempo finito.
Solución.
Las subrutinas son equivalentes, por lo que discutiremos solo `LeftRightShake`. Este método es un **for** que itera $f - i$ veces comparando y eventualmente intercambiando elementos, por lo que es finito. Como ambas subrutinas al menos mantienen el valor de j, `CocktailShakerSort` en cada iteración al menos reduce en 1 a f y aumenta en 1 a i. Luego, en alguna iteración se deja de cumplir $i \leq f$ y el loop termina. Como cada iteración del **while** es finita gracias a las subrutinas, `CocktailShakerSort`

termina.

Puntajes.
0.5 por argumentar respecto a los métodos auxiliares. Puede referirse a uno solo y mencionar que el otro es análogo.
0.5 por argumentar respecto al algoritmo principal

(b) Demuestre que el algoritmo `CocktailShakerSort` ordena.

- [2 ptos.] Pruebe primero que `LeftRightShake` cumple su objetivo. *Hint:* luego de la iteración de valor m del **for**, ¿qué satisface el tramo $A[j \dots m+1]$?
  Solución.
  Sea la propiedad

  $$P(m) := \text{Luego de la iteración de valor } m\text{, el tramo } A[j \dots m+1] \text{ está ordenado}$$

  Probaremos por inducción que $P(m)$ es cierta para cualquier m.

  - **Caso base.** $P(i)$: luego de la primera iteración, el tramo a considerar es $A[i \dots i+1]$. Si los elementos estaban ordenados, se mantuvieron. Si no lo estaban, se intercambiaron. En cualquier caso, estos dos primeros elementos están ordenados.
  - **Hipótesis inductiva.** Suponemos que se cumple $P(m)$.
  - **Tesis inductiva.** Supongamos que acaba de terminar la iteración $m+1$. Si $A[m+1] > A[m+2]$, entonces debemos intercambiar y luego $j \leftarrow m+1$, indicando que j fue el último elemento intercambiado. Con esto, el tramo $A[j \dots m+2] = A[m+1 \dots m+2]$, que gracias al intercambio quedó ordenado.

    El otro caso es si $A[m+1] \leq A[m+2]$, en cuyo caso, estos elementos están ordenados entre sí y j no cambia respecto a la iteración anterior. Dado que por **H.I.** el tramo $A[j \dots m+1]$ está ordenado y ahora sabemos que $A[m+1 \dots m+2]$ también lo está, concluímos que $A[m+1 \dots m+2]$ está ordenado.

  Puntajes.
  0.5 por plantear la propiedad a demostrar
  0.5 por demostrar el caso base
  0.5 por el caso $A[m+1] > [m+2]$ del paso inductivo
  0.5 por el caso $A[m+1] \leq [m+2]$ del paso inductivo
  Observación: pueden usarse otras formas de demostración que argumenten correctamente la correctitud del algoritmo
- [1 pto.] Pruebe que `CocktailShakerSort` ordena, dado que `LeftRightShake` y `RightLeftShake` cumplen su objetivo.
  Solución.
  Consideremos una iteración cualquiera de `CocktailShakerSort`. Sabemos que al terminar `LeftRightShake`(A, i, f) con valor k, el tramo $A[k \dots f]$ está ordenado, i.e. el final de la secuencia a partir de k. De forma análoga, `RightLeftShake`(A, i, f) resultando en valor t deja ordenado el tramo $A[i \dots t]$. Dado que los índices f, i se actualizan achicando el rango al menos en 1 para cada subrutina, la siguiente iteración ordenará tramos que tienen un elemento más en ambos extremos. Luego, como eventualmente $i > f$, ambos extremos se tocan y la secuencia completa está ordenada.

  Puntajes.
  0.5 por utilizar que los algoritmos auxiliares ordenan los extremos en una iteración cualquiera
  0.5 por argumentar que esos extremos se acercan en cada iteración, encontrándose

(c) [1 pto.] Determine el mejor caso de `CocktailShakerSort`. Explique brevemente su respuesta.
Solución.
Tal como `InsertionSort`, `CocktailShakerSort` logra detectar si la secuencia está ordenada en la línea 3 de sus subrutinas. Su mejor caso es que esta esté ordenada (no decreciente).

Puntajes.
0.5 por mencionar el mejor caso
0.5 por justificar que el algoritmo detecta elementos ordenados (o similar)

(d) [1 pto.] Determine su complejidad en el mejor caso.
Solución.
En el mejor caso, en la primera iteración del **while**, cada subrutina solo itera por el arreglo sin aumentar su variable j. Luego, los índices principales cumplen $f = i - 1$ y $i = f + 1$ al término de esta primera iteración, por lo que el **while** termina. La complejidad de las subrutinas es $\mathcal{O}(n)$ por tratarse de loops **for**. La complejidad de mejor caso es por lo tanto $\mathcal{O}(n)$.

Puntajes.
0.5 por argumentar que se ejecuta solo una iteración del **while**
0.5 por plantear complejidad del **for** que se ejecuta y la complejidad final

2. Diseño de algoritmos

Un archivo contiene datos de todos los estudiantes que han rendido cursos del DCC desde 1980 hasta la fecha. El formato cada registro en el archivo es

`RUT`	`primer_apellido`	`segundo_apellido`	`nombre`

y los registros se encuentran ordenados por `RUT`.

(a) [3 ptos.] Proponga el pseudocódigo de un algoritmo para ordenar los registros alfabéticamente, i.e. según (`primer_apellido`, `segundo_apellido`, `nombre`). Especifique qué estructura básica usará (listas o arreglos). Si p es un registro, puede acceder a sus atributos con `p.primer_apellido`, `p.nombre`, etc. Además, puede asumir que todo algoritmo de ordenación visto en clase puede ordenar respecto a un atributo específico.

Solución.

input : Arreglo $A[0, \ldots, n-1]$ e índices i, f
output: Lista de pares de índices L

`FirstLastNameReps` (A, i, f)**:**
    $L \leftarrow$ lista vacía
    $k \leftarrow i$
    $j \leftarrow i$
    for $m = 1 \ldots f$ **:**
        if $A[m].$*primer_apellido* $= A[k].$*primer_apellido* **:**
            $j \leftarrow m$
        else:
            if $k < j$ **:**
                añadir a L el par (k, j)
            $k \leftarrow m$
            $j \leftarrow m$
    return L

Es decir, `FirstLastNameReps` (A, i, f) entrega una lista con los rangos entre los cuales hay repeticiones de primer apellido entre los índices i y f. De forma similar se define la rutina `SecondLastNameReps` que entrega rangos de repetidos de segundo apellido. El algoritmo principal es el siguiente

input : Arreglo $A[0, \ldots, n-1]$
output: Arreglo ordenado alfabéticamente

```
AlphaSort (A):
1   MergeSort(A, 0, n − 1, primer_apellido)      ▷ ordenamos A según primer apellido
2   F ← FirstLastNameReps(A, 0, n − 1)
3   for (k, j) ∈ F :
4       MergeSort(A, k, j, segundo_apellido)
5       S ← SecondLastNameReps(A, k, j)
6       for (s, t) ∈ S :
7           MergeSort(A, s, t, nombre)
```

Puntajes.
1.0 por el algoritmo que determina los rangos que deben ser ordenados según el siguiente atributo
2.0 por el algoritmo principal (puede asumir que existe un algoritmo que entrega los rangos)
Observación: el algoritmo propuesto puede diferir del presentado en esta pauta. Lo importante es que cumpla el objetivo de ordenar por los tres atributos.

(b) [2 ptos.] Determine la complejidad de peor caso de su algoritmo en función del número de estudiantes en el archivo.
Solución.
El peor caso corresponde a una cantidad $\mathcal{O}(n)$ de repetidos en primer y segundo apellido. Luego, el análisis de complejidad puede resumirse en

- Línea 1 en $\mathcal{O}(n \log(n))$
- Línea 2 en $\mathcal{O}(n)$
- **for** de línea 3 se ejecuta $\mathcal{O}(n)$ veces
  - Línea 4 en $\mathcal{O}(n \log(n))$
  - Línea 5 en $\mathcal{O}(n)$
  - **for** de línea 6 se ejecuta $\mathcal{O}(n)$ veces
    - Línea 7 en $\mathcal{O}(n \log(n))$

Con esto, la complejidad sería

$$\mathcal{O}(n \log(n) + n + n \cdot [n \log(n) + n + n \cdot (n \log(n))]) \quad \Rightarrow \quad \mathcal{O}(n^3 \log(n))$$

Puntajes.
1.0 por plantear complejidad de etapas/pasos
1.0 por concluir complejidad del algoritmo principal
Observación: lo importante es que el análisis sea acorde al algoritmo propuesto. No se pedían requisitos de complejidad.

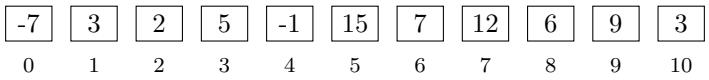
(c) [1 pto.] Le informan que, si bien en el archivo hay primeros apellidos repetidos, la cantidad de repeticiones por apellido es muy baja (#repeticiones < 20). ¿Puede proponer alguna mejora a su algoritmo utilizando esta información? Justifique.
Solución.
Una posible mejora sería utilizar `InsertionSort` para ordenar las secuencias de repetidos, a fin de evitar la recursión y aprovechar el mejor desempeño práctico de `InsertionSort` en instancias pequeñas.

Puntajes.

1.0 por proponer y justificar
Observación: depende mucho de la solución propuesta. Si el algoritmo ya incluye mejoras, es válido mencionar que no hay mejoras obvias.

3. Estrategias algorítmicas

Dada una secuencia $A[0 \dots n-1]$ de números enteros, se define un **índice mágico** como un índice $0 \leq i \leq n-1$ tal que $A[i] = i$. Por ejemplo, en la siguiente secuencia



existen dos índices mágicos: el 2 y el 9.

Dada una secuencia $A[0 \dots n-1]$ **ordenada**, sin elementos repetidos e implementada como arreglo,

(a) [4 ptos.] Proponga el pseudocódigo de un algoritmo que retorne un índice mágico en A si existe y que retorne `null` en caso contrario. Su algoritmo debe ser más eficiente que simplemente revisar el arreglo elemento por elemento, i.e. mejor que $\mathcal{O}(n)$. *Hint:* utilice la estrategia dividir para conquistar.

Solución.

input : Arreglo $A[0, \dots, n-1]$, índices i, f
output: Índice mágico o `null`

`Magic` (A, i, f)**:**

```
1   if (f − i) = 0 :
2       if A[i] = i :
3           return i
4       return null
5   p ← ⌊(f − i)/2⌋
6   if A[p] = p :
7       return p
8   if A[p] > p :
9       return Magic(A, i, p − 1)
10  return Magic(A, p + 1, f)
```

Puntajes.
1.0 por utilizar llamados recursivos a instancias más pequeñas
1.0 por caso base que determina si el elemento es índice mágico
2.0 por rangos correctos al hacer los llamados recursivos (que se escoja correctamente qué tramo contiene el posible índice mágico)
Observación: puede proponerse otra solución, pero debe tener una complejidad que en cualquier caso sea mejor que $\mathcal{O}(n)$. Notar que la solución de esta pauta es $\mathcal{O}(\log(n))$ en el caso promedio y peor caso, mientras que es $\mathcal{O}(1)$ en el mejor caso.

(b) [2 ptos.] Determine la complejidad de peor caso de su algoritmo.
Solución.
Definimos como $T(n)$ el número de pasos necesarios para encontrar un índice mágico (o responder `null`) para un arreglo de tamaño n. Con esto, la ecuación de recurrencia de este problema es

$$T(1) = 1, \quad T(n) = T(n/2) + c, \text{ para } c \text{ constante}$$

Luego, usando una estrategia similar a la empleada en clases, resolvemos la recurrencia como sigue

$$\begin{aligned} T(n) &= T(n/2) + c \\ T(n/2) &= T(n/4) + c \\ T(n/4) &= T(n/8) + c \\ &\vdots \\ T(2) &= T(1) + c \end{aligned}$$

Viendo que $1 = n/n = n/2^k$, deducimos que tenemos $k = \log(n)$ ecuaciones. Sumándolas, queda $T(n) = 1 + c\log(n)$ y eliminando constantes obtenemos $T(n) \in \mathcal{O}(\log(n))$. Se verifica que el algoritmo tiene mejor complejidad que $\mathcal{O}(n)$.

Puntajes.
1.0 por plantear una ecuación de recurrencia adecuada
1.0 por resolverla y concluir correctamente
Observación: se pueden usar otras técnicas como el teorema maestro. También se puede argumentar su similitud con búsqueda binaria y deducir que tiene la misma complejidad.

4. Modificación de algoritmos

(a) [2 ptos.] Considere el algoritmo `MergeThree` que toma como argumento 3 secuencias ordenadas, en lugar de 2 como el algoritmo `Merge` visto en clases. Su output sigue siendo una nueva secuencia ordenada con los elementos de las subsecuencias. Si diseñamos una versión de `MergeSort` que utilice `MergeThree`, ¿cuál es la complejidad asintótica de esta nueva versión en función de la cantidad n de elementos en la secuencia inicial? *Hint:* Puede utilizar los supuestos que necesite sobre n para simplificar el análisis.
Solución.
Dadas 3 secuencias, `Merge` sigue encontrando el mínimo entre las tres en tiempo $\mathcal{O}(1)$. Es decir, `Merge` mantiene su tiempo $\mathcal{O}(n)$. Luego, la única diferencia en `MergeSort` será que en lugar de de dividir la secuencia actual en 2 partes, se divide en 3 y se hacen 3 llamados recursivos antes de `Merge`. Luego, suponiendo que n es potencia de 3, la ecuación de recurrencia se modifica como sigue

$$T(1) = 1, \quad T(n) = 3T(n/3) + n$$

lo que entrega como resultado $T(n) = n\log_3(n) + n$. La complejidad asintótica sigue siendo $\mathcal{O}(\log(n))$.

Puntajes.
1.0 por plantear diferencias en el análisis con la versión vista en clase
1.0 por concluir que sigue siendo logarítmico
Observación: dado que no se especifica, se permite dar una interpretación distinta al `MergeSort` modificado, pudiendo tener una complejidad distinta. Lo importante es que el análisis sea coherente con la solución propuesta.

(b) [4 ptos.] Recuerde que dado un arreglo A, una inversión es un par de índices (i, j) tales que $i < j$ y $A[i] > A[j]$. Proponga el pseudocódigo de un algoritmo que determine el número de inversiones en un arreglo de n elementos en tiempo $\mathcal{O}(n\log(n))$ en el peor caso. *Hint:* modifique la versión de `Merge`/`MergeSort` vista en clases. No necesita demostrar su complejidad.

Solución.

input : Arreglo A
output: Arreglo A ordenado y d conteo de inversiones en A

`RecMergeCount` (A)**:**
  if $|A| = 1$ **:**
    return $(A,\ 0)$
  dividir A en mitades A_1, A_2
  $B_1, d_1 \leftarrow$ `RecMergeCount`(A_1)
  $B_2, d_2 \leftarrow$ `RecMergeCount`(A_2)
  $B, d \leftarrow$ `MergeCount`(B_1, B_2)
  return $(B,\ d_1 + d_2 + d)$

input : Arreglos A, B ordenados
output: Arreglo C ordenado y d conteo de inversiones en AB

`MergeCount` (A, B)**:**
  $d \leftarrow 0$
  $i \leftarrow 0$
  $j \leftarrow 0$
  $C \leftarrow$ arreglo vacío
  while $i < |A| \wedge j < |B|$ **:**
    $m \leftarrow$ mín$(A[i], B[j])$
    if $A[i] > B[j]$ **:**
      $d \leftarrow d + |A| - i$ ▷ `se suma el tamaño restante de` $|A|$
      agregar $B[j]$ a C
      $j \leftarrow j + 1$
    else:
      agregar $A[i]$ a C
      $i \leftarrow i + 1$
  agregar los restantes a C
  return C, d

Puntajes.
1.0 por modificar `Merge` para que agregue inversiones al encontrar el mínimo en el segundo arreglo
1.0 por modificar `Merge` para que retorne el número de inversiones así como el arreglo
1.0 por el caso base con el número de inversiones correcto en `MergeSort`
1.0 por modificar `MergeSort` para combinar correctamente los valores retornados
Observación: se aceptan algoritmos que no retornen el arreglo y el número de inversiones, pero debe cumplirse el objetivo de contar correctamente y que la complejidad sea la deseada.