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

IIC2133 — Estructuras de Datos y Algoritmos 1'2023

Interrogación 1

3 de abril de 2023

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

Nota. Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas.

Uso de algoritmos. 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 algoritmo `GnomeSort`, apodado *stupid sort* debido a que para ciertos inputs puede realizar una cantidad de iteraciones mayor a n.

input : Arreglo $A[0,\dots,n-1]$, largo $n \geq 2$

`GnomeSort`(A, n)**:**
  $p \leftarrow 0$
  while $p < n$ **:**
    if $p = 0 \vee A[p] \geq A[p-1]$ **:**
      $p \leftarrow p + 1$
    else:
      $A[p] \rightleftharpoons A[p-1]$
      $p \leftarrow p - 1$

(a) [3 ptos.] Demuestre que luego de la k-ésima iteración del **while** de `GnomeSort`, $A[0 \dots p-1]$ está ordenada. Note que p no necesariamente coincide con el número de iteraciones que se han ejecutado hasta el momento. *Pista:* use inducción sobre k.
Solución.
Definimos la propiedad

$$P(k) := \text{al término de la } k\text{-ésima iteración, } A[0 \dots p-1] \text{ está ordenado}$$

notando que el valor p no está directamente relacionado con k. Luego, demostramos por inducción sobre k

- **C.B.** Para $k = 1$, al término de la primera iteración $p = 1$ y $A[0 \dots p-1] = A[0]$ tiene un solo elemento, que está trivialmente ordenado.
- **H.I.** Suponemos que luego de la iteración k el tramo $A[0 \dots p-1]$ está ordenado.
- **T.I.** Partimos de la iteración k y ejecutamos la siguiente iteración.
  - Si el elemento $A[p] \geq A[p-1]$, entonces está bien ubicado y al aumentar p, al término de la iteración $k+1$ se tiene que el tramo $A[0 \dots (p+1)-1]$ está ordenado.
  - Si $A[p] < A[p-1]$, se intercambian y se reduce p. Es decir, corresponde analizar $A[0 \dots (p-1)-1]$, que por **H.I.** está ordenado.

  Concluimos que en ambos casos, al término de la iteración $k+1$ la secuencia $A[0 \dots p-1]$ está ordenado. Esto demuestra que el algoritmo en cada iteración ordena.

Puntajes.
0.5 por plantear la propiedad a demostrar
0.5 por demostrar el caso base
1.0 por el caso $A[p] \geq A[p-1]$ del paso inductivo
1.0 por el caso $A[p] < A[p-1]$ del paso inductivo
Observación: pueden usarse otras formas de demostración que argumenten correctamente la correctitud del algoritmo

(b) [1 pto.] Demuestre que `GnomeSort` termina, es decir, que se logra $p = n$ al término de alguna iteración del **while**.
Solución.
Notemos que el bloque **if** detecta si hay elementos consecutivos invertidos. En caso de encontrar inversión, esta se intercambia hacia el inicio del arreglo hasta ubicarlo correctamente. Como este proceso no se repite para elementos ya intercambiados, sabemos que a lo más para cada elemento se hacen n intercambios. Es decir, una vez que se ubica correctamente un elemento, p incrementa hasta volver a los elementos no revisados, eventualmente llegando a $p = n$. Luego, el bloque **while** se ejecuta una cantidad finita de veces.
Puntajes.
0.5 por justificar que el algoritmo efectivamente *avanza* en su proceso de reubicar elementos.
0.5 por justificar que cada proceso de reubicación a lo más toma n pasos (o similar)
Observación: se aceptan respuestas diferentes. Lo importante es convencer justificadamente de que efectivamente se alcanza $p = n$ y que no se queda en un loop de aumentar y reducir p sin llegar a la condición de término.

(c) [1 pto.] Determine la complejidad de tiempo y espacio de `GnomeSort` en el peor caso.
Solución.
En el peor caso, el arreglo está totalmente invertido y para cada elemento $A[i]$ se gatilla el bloque **else** hasta ubicar correctamente el elemento. Esto significa una cantidad de intercambios $\mathcal{O}(n)$ para cada elemento, de los cuales hay n. Por lo tanto, el algoritmo toma tiempo $\mathcal{O}(n^2)$ en el peor caso. En términos de memoria, como los intercambios se hacen sobre el mismo arreglo de input y solo se utiliza una cantidad fija de índices y variables adicionales, usa memoria $\mathcal{O}(1)$ adicional.
Puntajes.
0.5 por justificar tiempo $\mathcal{O}(n^2)$. No es necesario hacer referencia explícita al peor caso, pero sí a una cantidad lineal de intercambios **por elemento.**
0.5 por justificar memoria $\mathcal{O}(1)$.

(d) [1 pto.] ¿Tiene un mejor caso? Justifique y en caso afirmativo, determine su complejidad.
Solución.
Tal como `InsertionSort`, este algoritmo se *da cuenta* cuando los elementos vienen ordenados, específicamente en el bloque **if**. El arreglo ordenado siempre hace avanzar p y por lo tanto termina en $\mathcal{O}(n)$ iteraciones. La complejidad de mejor caso es tiempo $\mathcal{O}(n)$.
Puntajes.
0.5 por describir el mejor caso.
0.5 por entregar justificadamente la complejidad $\mathcal{O}(n)$.

2. Diseño de algoritmos

Su compañía empleadora ha sido contratada para actualizar el sistema de control de un aeropuerto. Le corresponde modificar el sistema que selecciona el próximo avión que debe aterrizar, que actualmente utiliza una cola FIFO implementada en un arreglo $A[0..n-1]$ en que cada avión P_i que llega a la cola trae asociado un número de secuencia S_i que lo identifica. El valor n es adecuado para el tráfico del aeropuerto.

Además de S_i, cada avión en la cola Q cuenta con su autonomía de vuelo F_i (una medida de cuánto tiempo puede permanecer en el aire sin caer) y el número de pasajeros que transporta T_i. La autonomía F_i se actualiza para todos los aviones P_i de la cola Q cada un minuto y esta autonomía cambia en tasas diferentes para cada avión P_i. Los aviones nuevos que llegan a la cola Q ingresan con su secuencia S_i, autonomía F_i y número de pasajeros T_i definidas.

Para los siguientes escenarios, proponga una algoritmo que resuelva lo solicitado.

(a) [3 ptos.] Se busca disminuir el riesgo de caída, por lo que se requiere ordenar la cola Q primero por la autonomía F_i de cada avión P_i en la cola Q. Para autonomías iguales se debe ordenar por número de pasajeros T_i y para los casos con igual autonomía y número de pasajeros se debe ordenar por la secuencia S_i del avión. El siguiente avión a aterrizar debe quedar en la cabeza de la cola Q, i.e. en la posición $Q[0]$. Escriba en pseudo código el algoritmo `sortTrafico`(Q, i, f) para ordenar la cola Q según lo indicado.

Solución.

input : Arreglo $Q[0, \dots, n-1]$ e índices i, f
output: Lista de pares de índices L que comparten autonomía

`RepeatedRanges` (Q, i, f)**:**
  $L \leftarrow$ lista vacía
  $k \leftarrow i$
  $j \leftarrow i$
  for $m = 1 \dots f$ **:**
    if $Q[m].\mathit{autonomía} = Q[k].\mathit{autonomía}$ **:**
      $j \leftarrow m$
    else:
      if $k < j$ **:**
        añadir a L el par (k, j)
      $k \leftarrow m$
      $j \leftarrow m$
  return L

Es decir, `RepeatedRanges` (A, i, f) entrega una lista con los rangos entre los cuales hay repeticiones de autonomía entre los índices i y f. De forma similar se define la rutina `RepeatedRanges2` que entrega rangos de repetidos de capacidad entre los índices específicados. El algoritmo principal es el siguiente

input : Arreglo $Q[0, \dots, n-1]$, índices i, f
output: Arreglo ordenado por autonomía, capacidad y número de secuencia

`SortTrafico` (Q, i, f)**:**
  `MergeSort`$(Q, 0, n-1,$ `autonomía`$)$ ▷ ordenamos A crecientemente según autonomía
  $F \leftarrow$ `RepeatedRanges`(Q, i, f)
  for $(k, j) \in F$ **:**
    `MergeSort`$(Q, k, j,$ `capacidad`$)$
    $S \leftarrow$ `RepeatedRanges2`(Q, k, j)
    for $(s, t) \in S$ **:**
      `MergeSort`$(Q, s, t,$ `secuencia`$)$

Puntajes.
1.0 por el algoritmo que determina los rangos que deben ser ordenados según el siguiente atributo. Puede estar explicado a alto nivel.
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 y que quede de forma creciente según autonomía.

(b) [3 ptos.] Para permitir una forma eficiente de desviar tráfico aéreo a otros terminales se requiere encontrar en la cola Q el avión P_i con la menor autonomía mayor que un valor D para desviarlo a otro aeropuerto. Escriba en pseudo código el algoritmo `desviaTrafico`(Q, i, f, D) que retorne la posición en la cola Q del avión P_i que cumple lo solicitado.

Solución.

input : Arreglo $Q[0,\ldots,n-1]$, índices i, f, valor umbral D
output: índice de elemento con la menor autonomía mayor que D

`desviaTrafico` (Q, i, f, D):
  if $f < i$ **: return** -1
  if $Q[i] > D$ **: return** i
  $m \leftarrow \left\lfloor \dfrac{i+f}{2} \right\rfloor$
  if $Q[m] \leq D$ **:**
    return `desviaTrafico`$(A, m+1, f, D)$
  if $Q[m] > D$ **:**
    $a \leftarrow$ `desviaTrafico`$(A, i, m-1, D)$
    if $a = -1 \vee a \geq Q[m]$ **:**
      return $Q[m]$
    return a

Es decir, `desviaTrafico` (A, i, f) es una versión modificada de búsqueda binaria, que detecta en Q ordenada la zona que debe revisar y escoge según el umbral dado. El valor de retorno -1 indica que no existe tal avión.
Puntajes.
1.0 por la idea de usar búsqueda binaria sobre la secuencia ordenada.
1.0 por caso base y caso $\leq$
1.0 por caso $>$, que requiere comparar
Observación: el algoritmo propuesto puede diferir del presentado en esta pauta.

3. Estrategias algorítmicas

Considere dos arreglos $A[0 \ldots n-1]$ y $B[0 \ldots n-1]$ ordenados y del mismo tamaño.

(a) [3 ptos.] Proponga el pseudocódigo de un algoritmo que utilice la estrategia dividir para conquistar que retorne la mediana del arreglo $C[0 \ldots 2n-1]$ que se obtendría al combinar los arreglos A y B. Su algoritmo debe tener una complejidad asintótica mejor que lineal en el peor caso.

Solución.

input : Arreglo ordenado $A[0,\ldots,n-1]$, índices $i < f$
output: valor de la mediana de A

`GetSortedMedian` (A, i, f):
  $n \leftarrow f - i$
  if $(n \,\%\, 2) = 0$ **:**
    return $(A[n/2] + A[n/2+1])/2$
  return $A[\lfloor n/2 \rfloor]$

input : Arreglos $A[0,\dots,n-1]$ y $A[0,\dots,n-1]$, índices i_A, f_A de A y i_B, f_B de B
output: valor de la mediana al considerar los elementos de A y B

`Medians` $(A, B, i_A, f_A, i_B, f_B)$:
  if $i_A = f_A$ **:**
    return $A[i_A]$
  $m_A \leftarrow$ `GetSortedMedian`(A, i_A, f_A)
  $m_B \leftarrow$ `GetSortedMedian`(B, i_B, f_B)
  if $m_A = m_B$ **:**
    return m_A
  if $m_A < m_B$ **:**
    $i'_A \leftarrow \lfloor (i_A + f_A)/2 \rfloor$
    $f'_B \leftarrow \lfloor (i_B + f_B)/2 \rfloor - 1$
    return `Medians`$(A, B, i'_A, f_A, i_B, f'_B)$
  $f'_A \leftarrow \lfloor (i_A + f_A)/2 \rfloor - 1$
  $i'_B \leftarrow \lfloor (i_B + f_B)/2 \rfloor$
  return `Medians`$(A, B, i_A, f'_A, i'_B, f_B)$

Puntajes.
1.0 por utilizar llamados recursivos a instancias más pequeñas
1.0 por caso base que determina si el largo de las secuencias es 1
2.0 por rangos correctos al hacer los llamados recursivos (que se escoja correctamente qué tramo contiene el posible índice mágico)
Observación: una buena propiedad del problema es que en todo momento, los tramos de A y B que se consideran son del mismo tamaño. Se aceptan enfoques distintos, pero que sean $\mathcal{O}(\log(n))$ como el propuesto.

(b) [2 ptos.] Determine justificadamente la complejidad de tiempo en el peor caso para su algoritmo.
Solución.
Definimos como $T(n)$ el número de comparaciones = necesarias para encontrar la mediana para dos arreglos de tamaño n en el peor caso (cuando la mediana nunca coincide salvo en el llamado más profundo). Con esto, la ecuación de recurrencia de este problema es

$$T(1) = 1, \quad T(n) = T(n/2) + 3,$$

Notar que `GetSortedMedian()` es simplemento un acceso por índice dado que las secuencias están ordenadas, por lo que aporta solo una comparación.
Luego, usando una estrategia similar a la empleada en clases, resolvemos la recurrencia como sigue

$$\begin{array}{rcl} T(n) & = & T(n/2) + 3 \\ T(n/2) & = & T(n/4) + 3 \\ T(n/4) & = & T(n/8) + 3 \\ & \vdots & \\ T(2) & = & T(1) + 3 \end{array}$$

Viendo que $1 = n/n = n/2^k$, deducimos que tenemos $k = \log(n)$ ecuaciones. Sumándolas, queda $T(n) = 1 + 3\log(n)$ y eliminando constantes obtenemos $T(n) \in \mathcal{O}(\log(n))$.
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.

(c) [1 pto.] ¿Su algoritmo tiene un mejor caso? Justifique, y en caso afirmativo, entregue la complejidad de tiempo en el mejor caso.

Solución.
El algoritmo tiene como mejor caso aquel en que la mediana de ambos arreglos coincide en el primer llamado. En tal caso, la línea 6 retorna exitosamente sin realizar llamados recursivos. Esto significa que el tiempo de mejor caso es $\mathcal{O}(1)$.
Puntajes.
0.5 por indicar cuál es el mejor caso
0.5 por especificar la complejidad de tiempo en el mejor caso

4. Modificación de algoritmos

Se tiene un arreglo $A[0 \dots n-1]$ originalmente ordenado, tal que varios de sus elementos fueron desordenados. Se sabe que en este minuto, al menos un 80 % de sus elementos están en su posición correcta ordenada. En su empresa se piensa usar `Quicksort` para ordenar nuevamente A.

(a) [2 ptos.] Un(a) ingeniero(a) de software (SI) propone que la elección del pivote sea la mediana entre los valores en los índices 0, $\lfloor n/2 \rfloor$ y $n-1$ del arreglo A. Indique en qué líneas o zona debe hacer la modificación en la versión de `Quicksort` vista en clases y especifique el fragmento de pseudocódigo nuevo para incorporar el cambio propuesto.

Solución.

El siguiente método se llama en lugar de aquel que genera el pivote aleatorio en `Partition` (línea 1). El llamado se hace con `GetPivot`(A, i, f) cuando se hace `Partition`(A, i, f).

input : Secuencia $A[0, \dots, n-1]$, índices i, f
output: Índice del pivote aleatorio en la secuencia ordenada

`GetPivot` (A, i, f)**:**
  $p_1 \leftarrow i,\ p_2 \leftarrow \lfloor (i+f)/2 \rfloor,\ p_3 \leftarrow f$
  if $A[p_1] \leq A[p_2] \leq A[p_3] \lor A[p_3] \leq A[p_2] \leq A[p_1]$ **:**
    $p \leftarrow p_2$
  elif $A[p_2] \leq A[p_1] \leq A[p_3] \lor A[p_3] \leq A[p_1] \leq A[p_2]$ **:**
    $p \leftarrow p_1$
  else:
    $p \leftarrow p_3$
  return p

Puntajes.
1.0 por plantear determinar mediana
1.0 por reemplazar solo la asignación de pivote.
Observación. Es importante que se sigue llamando a `Partition` y este usa el pivote nuevo. No basta con reemplazar el llamado de `Partition` por `GetPivot`.

(b) [2 ptos.] Un segundo SI propone que cuando el subarreglo a ordenar sea de tamaño 20 o menos se utilice `InsertionSort`, ya que si está ordenado, este es su mejor caso. Indique en qué líneas o zona debe hacerse la modificación en la versión de `Quicksort` vista en clases y especifique el fragmento de pseudocódigo nuevo para incorporar el cambio propuesto.

Solución.

Se modifica el caso base de `Quicksort` para que en lugar de revisar el caso base con $f \leq i$, se verifique si $(f-i) > 20$. En tal caso, se ejecuta `Quicksort` de forma usual. En caso contrario, se hace un llamado a `InsertionSort`(A, i, f).
Puntajes.
1.0 por indicar que se modifica la condición de caso base
1.0 por llamar a `InsertionSort` para la subsecuencia corresondiente.

(c) [2 ptos.] Un tercer SI propone reemplazar `Quicksort` por `InsertionSort` para todo el proceso, afirmando que si el arreglo A está 80 % ordenado, el desempeño de `InsertionSort` se debe parecer más a su mejor caso, que es mejor que `Quicksort`. ¿Es correcto lo propuesto? Justifique.

Solución.

Si se reemplaza por `InsertionSort` es importante notar que la fracción de elementos desordenados (invertidos) es proporcional a n, a saber, $\approx 0{,}2 \cdot n$. De esta forma, para aquellos elementos, `InsertionSort` ejecuta una cantidad lineal de veces y en consecuencia, tiene una ejecución $\mathcal{O}(n^2)$ de forma global. Dado que esta complejidad se corresponde con la de peor caso de `Quicksort`, no presenta una ventaja del punto de vista asintótico. Pero cabe notar que en la práctica, dado que la posición final de los pivotes en `Quicksort` no se puede anticipar, `InsertionSort` puede presentar una ventaja en los tiempos empíricos.

Puntajes.

1.0 por indicar que hay una cantidad $\mathcal{O}(n)$ de elementos mal ordenados
1.0 por por argumentar que `InsertionSort` es $\mathcal{O}(n^2)$ en este caso.