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

IIC2133 — Estructuras de Datos y Algoritmos 1'2024

Interrogación 1

10 de mayo de 2024

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.

1. Algoritmos de ordenación, correctitud, análisis

Bob ha cocinado pancakes y los dejó apilados sobre la mesa, formando una pila $P[0..n-1]$ tal que $P[0]$ almacena el pancake que está en la cima de la pila, mientras que $P[n-1]$ representa el que está en la base. Los pancakes tienen un diámetro representado por un número real. Alice ve la pila y decide ordenarla de tal manera que el diámetro de los pancakes sea descendente desde la base de la pila hacia la cima. Para realizar la tarea, suponga que cuenta con las siguientes funciones:

- Diámetro($P[i]$): devuelve el diámetro del pancake $P[i]$, para $0 \leq i \leq n-1$. Asuma que todos los diámetros son distintos entre sí.
- Invertir(i): invierte $P[0..i]$, para $0 \leq i \leq n-1$. Esta función simula la acción de colocar una espátula debajo del pancake $P[i]$, para luego invertir el orden de los elementos de la pila $P[1..i]$ que queda por encima de la espátula. Asuma que este proceso es in-place. Por ejemplo, para la pila $P[0..5] = \langle 3.2, 4.1, 2.8, 7.3, 6.1, 1.3\rangle$, Invertir(3) produce como resultado la pila $\langle 7.3, 2.8, 4.1, 3.2, 6.1, 1.3\rangle$.

(a) (3 puntos) Escriba el algoritmo PancakeSort($P[0..n-1]$), que permita ordenar la pila de pancakes $P[0..n-1]$ como se indica. La única manera permitida de manipular los pancakes de la pila es a través de las funciones Diámetro e Invertir. Asuma que dichas funciones ya están implementadas. Si necesita emplear otra función distinta a esas, debe implementarla. Hay muy poco espacio disponible en la mesa, por lo que su algoritmo debe ser *in-place*.

(b) (2 puntos) Demuestre formalmente la correctitud de su algoritmo.

(c) (1 punto) Analice su algoritmo, indicando el tiempo de ejecución del mismo, contando sólamente la cantidad de veces que llama a la función Invertir. Repita su análisis, pero ahora contando la cantidad de veces que invoca a la función Diámetro.

Solución:

(a) El algoritmo es similar a Selectionsort, salvo por las restricciones que impone este ejercicio. En cada iteración se debe buscar el siguiente pancake de mayor diámetro, para trasladarlo hacia la parte inferior de la pila usando la función Invertir. El pseudocódigo de una posible solución es el siguiente:

```
input  : Una pila de pancakes P[0..n-1]
output: La pila de pancakes P, ordenada

PancakeSort(P[0..n-1])
for i ← n-1 downto 1 :
    m ← MÁXIMO(P[0..i])
    Invertir(m)
    Invertir(i)
return P
```

en donde el algoritmo MÁXIMO se define como a continuación:

input : Una pila de pancakes $P[0..i]$
output: La posición m tal que $0 \leq m \leq i$ y $P[m]$ es el pancake con el diámetro máximo.

```
MÁXIMO(P[0..i])
m ← 0
for i ← 1 to i :
    if Diámetro(i) > Diámetro(m) :
        m ← i
return m
```

(b) La demostración de correctitud del algoritmo debe mostrar que:

- PancakeSort termina en una cantidad finita de pasos (**0.5 pts.**).
- PancakeSort cumple su propósito (**1.5 pts.**).

Para demostrar la finitud del algoritmo, note que realiza siempre $n-1$ iteraciones. En cada iteración, busca el pancake de mayor diámetro, lo cual requiere la invocación al algoritmo MÁXIMO($P[0..i]$) (que finaliza en $i-1$ iteraciones), para luego invocar dos veces a la operación Invertir. En otras palabras, cada una de las operaciones que se realizan en cada iteración de PancakeSort son finitas, y por lo tanto el algoritmo finaliza en una cantidad finita de pasos.

La demostración de que PancakeSort cumple su propósito es por inducción, similar a la demostración respectiva hecha en clases para SelectionSort. La propiedad que demostraremos es la siguiente:

- $\mathbb{P}(i)$: al finalizar la iteración i (para $1 \leq i \leq n$) de PancakeSort, $P[n-i..n-1]$ está ordenada y contiene los i pancakes de mayor diámetro.

Caso Base: el caso base es para $i = 1$ (es decir, la primera iteración). Al finalizar la primera iteración, $\mathbb{P}(1)$ se cumple ya que el elemento de mayor diámetro es movido a la base $P[n-1]$ de la pila.

Hipótesis Inductiva: Al finalizar la iteración i, se cumple $\mathbb{P}(i)$.

Paso Inductivo: Demostramos ahora que $\mathbb{P}(i+1)$ se cumple, dado que $\mathbb{P}(i)$ (la H.I.) se cumple. Esto último significa que $P[n-i..n-1]$ está ordenada y contiene los i pancakes de mayor diámetro, como hemos dicho, y por lo tanto $P[0..n-i-1]$ contiene los $n-i$ pancakes de menor diámetro. De entre esos pancakes de menor diámetro se selecciona aquel que tiene mayor diámetro, el cual es movido a la posición $P[n-(i+1)]$ de la pila. Eso significa que ahora $P[n-(i+1)..n-1]$ está ordenada y contiene los $i+1$ pancakes de mayor diámetro, por lo que $\mathbb{P}(i+1)$ se cumple.

(c) El tiempo de ejecución de PancakeSort es:

- Invocaciones a Invertir (**0.5 pts.**): la cantidad de invocaciones a Invertir es $2(n-1)$, ya que se realizan 2 invocaciones por cada una de las $n-1$ iteraciones que realiza.
- Invocaciones a Diámetro (**0.5 pts.**): las invocaciones a esta operación se hacen desde el algoritmo MÁXIMO. En particular, MÁXIMO($P[0..i]$) realiza $2i$ invocaciones a Diámetro, para $i = 1, \ldots, n-1$. Entonces, la cantidad de invocaciones a Diámetro es:

$$\sum_{i=1}^{n-1} 2i = 2\frac{n(n-1)}{2} = n^2 + n.$$

2. Modificación de algoritmos, dividir para conquistar

Considere dos arreglos $A[0 \ldots n-1]$ y $B[0 \ldots n-1]$ ordenados, ambos de largo n. Nos interesa el problema de determinar el valor de la mediana de los datos al considerar todos los elementos de A y B (incluyendo repetidos).

(a) (1 punto) Proponga el pseudocódigo de un algoritmo que resuelva el problema en tiempo $\Theta(n)$.

Solución.

Entrada: Arreglos ordenados $A[0, \ldots, n-1]$ y $B[0, \ldots, n-1]$
Salida : Valor de la mediana de A y B

Función `MergeMedian`(A, B)
  $C \leftarrow$ `Merge`(A, B)
  $n \leftarrow$ longitud de C
  if $(n \mod 2) = 0$ **:**
    return $(C[n/2-1] + C[n/2])/2$
  else:
    return $C[\lfloor n/2 \rfloor]$

Puntajes.
0.5 por utilizar estrategia con complejidad $\Theta(n)$ para ordenar
0.5 por obtener la mediana
Observación: es importante notar que el algoritmo del inciso (b) tiene complejidad $\mathcal{O}(n)$ y por lo tanto no responde a la pregunta planteada. La solución de este inciso debe tener complejidad $\Theta(n)$.

(b) (3 puntos) Proponga el pseudocódigo de un algoritmo que utilice la estrategia dividir para conquistar para resolver el problema en tiempo $\Theta(\log(n))$.

Solución.

Entrada: Arreglo ordenado $A[0, \ldots, n-1]$, índices $i < f$
Salida : Valor de la mediana de A

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

Entrada: Arreglos $A[0, \ldots, n-1]$ y $B[0, \ldots, n-1]$, índices i_A, f_A de A y i_B, f_B de B
Salida : Valor de la mediana al considerar los elementos de A y B

Función `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`$(A, B, i'_A, f_A, i_B, f'_B)$
  else:
    $f'_A \leftarrow \lfloor (i_A + f_A)/2 \rfloor - 1$
    $i'_B \leftarrow \lfloor (i_B + f_B)/2 \rfloor$
    return `Medians`$(A, B, i_A, f'_A, i'_B, f_B)$

Puntajes.
0.5 por utilizar llamados recursivos a instancias más pequeñas
0.5 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 $\Theta(\log(n))$ como el propuesto.

(c) (1 punto) Justifique la complejidad de tiempo en el peor caso para su algoritmo de (b).
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 simplemente 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

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

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.

(d) (1 punto) ¿Existe alguna instancia de A y B en que su algoritmo de (a) tenga mejor desempeño práctico que el de (b)? Justifique.
Solución.
Debido a que el algoritmo utilizado en (a) siempre tiene complejidad lineal en n, sabemos que no es posible que obtenga la mediana en tiempo logarítmico. Para esto, sería necesario que el algoritmo no fuera $\Theta(n)$. Luego, en la práctica no hay instancia donde (a) opere más rápido que (b).
Puntajes.
1.0 por argumentar aludiendo a que el algoritmo de (a) tiene una complejidad lineal que siempre es peor en términos de eficiencia asintótica que el algoritmo de (b).

3. Heaps

Un sistema de manejo de residuos contaminantes utiliza una cola priorizada para elegir el siguiente caso a analizar en función del índice de contaminación del caso. Para ésto utiliza un Max Heap H.

(a) (1 punto) Muestre paso a paso el contenido del heap H, inicialmente vacío, al insertar los valores: 11, 13, 7, 19, 17, 5, 23, 29, 3.
Solución.
Comenzamos con un arreglo vacío.

[11]

[11,13] // insert 11 al final, then siftup x 1

[13,11]

[13,11,7] // insert 7 al final, siftup x 0

[13,11,7,19] // insert 19 al final, siftup hasta la raiz

[19,13,7,11]

[19,13,7,11,17] // insert 17 al final, siftup x 1

[19,17,7,11,13]

[19,17,7,11,13,5] // insert 5 al final, siftup x 0

[19,17,7,11,13,5,25] // insert 25 al final, siftup hasta la raiz

[25,17,19,11,13,5,7]

[25,17,19,11,13,5,7] // insert 29 al final, siftup hasta la raiz

[29,25,19,17,13,5,7,11]

[29,25,19,17,13,5,7,11,3] // insert 5 al final, siftup x 0 // FIN

Interesa especialmente que entiendan el concepto de Heap, pueden en hacer las operaciones en el "árbol" conceptual o en arreglo.

Puntajes.

Se descuenta 0.1 por cada elemento insertado de forma incorrecta (ya sea no respetando la propiedad de heap o incumpliendo los algoritmos vistos para el caso compacto.)

Observación: se considera correcto mostrar el desarrollo en un heap como arreglo compacto, o como árbol. En el caso compacto, se debe verificar el buen uso de los algoritmos de sift según corresponda. En el árbol, que la propiedad de heap sea satisfecha correctamente al término de una inserción.

(b) (2 puntos) Un cambio en la norma de cálculo de los índices de contaminación implica que la prioridad más alta es ahora el número más negativo. Para adaptar el sistema escriba un algoritmo en pseudo código para convertir el Max Heap H en un Min Heap h, con complejidad menor que $O(n \log n)$.

Solución.

(1 punto) BuildHeap(A) permite construir un Heap (max o min) a partir de un arreglo cualquiera, en particular un MaxHeap, con complejidad O(n), lo que cumple lo solicitado. El pseudo código de BuildHeap(A) es el mismo visto en clases, por lo que no es necesario que lo detallen.

(1 punto) Deben explicitar la modificación necesaria en SiftDown(A,i) para ajustarla a un MinHeap (en clases se vió para MaxHeap)

SiftDown(H , i):

if i tiene hijos :

j ← hijo de i con menor prioridad

if H[j] ¡H[i] :

H[j] intercambiar H[i]

SiftDown(H , j)

(c) (2 puntos) Como parte del sistema de administración se le pide a Ud. escribir un algoritmo en pseudo código para modificar la prioridad P de un elemento del Min Heap h.

Solución.

(1 punto)

modKeyMinHeap(h,P,Q) // Q es la nueva prioridad

i = findKey(h,P) // retorna el indice de la llave P en h, -1 si no está

if i != -1 : // existe P en el MinHeap

if P ¡Q :

h[i] ← Q

SiftDown(h,i) // nueva prioridad mayor, mantener heap

else // P ¿Q

h[i] ← Q

SiftpUp(h,i) // nueva prioridad menor, mantener heap

endif

endif

return

(1 punto)

findKey(h,P) // busca en el heap la posición de la llave P

// El heap no es un ABB, luego la búsqueda implica full scan

// Lo eficiente (más que tratar de recorrer el "arbol-heap" con

// un DFS o similar es realizar una búsqueda linear directamente

// en el arreglo

found = -1

for i = 0 to n -1 // recorro todo el heap

if h[i] == P

found = i

endif

endfor

return found

No es necesario indicar las modificaciones a SiftUp y SiftDown

Observación: Si consideran que se dispone del elemento a modificar, el código de búsqueda no es necesario.

(d) (1 punto) Finalmente, escriba un algoritmo en pseudo código para eliminar del Min Heap h el elemento con prioridad P.
Solución.
deleteKeyMinHeap(h,P)

i = findKey(h,P) // retorna el indice de la llave P en h, -1 si no está

if i != -1 : // existe P en el MinHeap

Q ← h[n-1] // prioridad del último elemento del heap

h[n-1] ← null // reduce el tamaño del heap en 1

modKeyMinHeap(h,P,Q) // cambia la prioridad de P con la Q

// y mantiene la propiedad de Heap

Se usan las funciones de la pregunta anterior, puede asumir que el heap solo tiene las prioridades (no hay "valores" que llevar de un nodo a otro).

4. Árboles B+

Suponga que tiene que almacenar un conjunto de registros, cada uno de los cuales representa datos de una persona. En particular, por cada persona se almacena su RUT, teléfono, email y nombre. Suponga que sobre un conjunto de datos de ese tipo se tienen los dos siguientes árboles de búsqueda:

- **Rut-Index**: un árbol B+ como el visto en clases, cuyo directorio (nodos internos) están ordenados por el RUT de las personas y sus nodos externos (hojas) almacenan los registros de forma ordenada.
- **Fono-Index**: un árbol cuyo directorio (nodos internos) son como el de un árbol B+ y que está ordenado por teléfono, y en cuyos nodos externos los registros se almacenan desordenados.

(a) (3 puntos) Escriba los algoritmos:

- RangoRUT(i, f), que trabaja sobre Rut-Index, devolviendo un arreglo (o lista) con todos los registros tal que el RUT se encuentra en el rango $[i, f]$.
- RangoFono(i, f) que trabaja sobre Fono-Index, devolviendo un arreglo (o lista) con todos los registros tal que el teléfono se encuentra en el rango $[i, f]$.

Se recomienda modificar el algoritmo correspondiente de búsqueda en rango de los árboles B+.

(b) (3 puntos) Escriba el algoritmo Insert(r), el cual sea una modificación del algoritmo de inserción en árboles B+ para la inserción de un nuevo registro r representando los datos de una persona. Dicho algoritmo debe actualizar ambos índices mencionados.

Solución Parte A

Función `GetHoja(`T, i`)`
  if T es hoja **:**
    return T
  $s \leftarrow$ hijo entre llaves k_1, k_2 tal que $k_1 \leq i \leq k_2$
  return `GetHoja(`s, i`)`

(1pt) por recorrer usando el orden interno de las hojas

Función `RangoRUT(`i, f`)`
  $T \leftarrow$ `GetHoja(`*RutIndex,* i`)`
  $m \leftarrow T$.start el menor de la hoja
  $A \leftarrow$ lista vacía
  while $m < i$ **:**
    $m \leftarrow m$.next
    $M \leftarrow m$
  while $M \leq f$ **:**
    A.add(M)
    $M \leftarrow M$.next
  return A

(1pt) por recorrer usando que las hojas están ligadas pero internamente desordenadas

Función `RangoFono(`i, f`)`
  $T1 \leftarrow$ `GetHoja(`*fono-index,* i`)`
  $T2 \leftarrow$ `GetHoja(`*fono-index,* f`)`
  $A \leftarrow$ lista vacía
  foreach T in lista $T1 \rightarrow \ldots \rightarrow T2$ **do**
    foreach m in T **do**
      if $i \leq m \leq f$ **:**
        A.add(m)
  return A

Solución Parte B

(1,5 pt) por distinguir entre llave y dato

Función `Insert(`r`)`
  `InsertClasico(`*Rut-index, r.rut,* r`)`
  `InsertDesordenado(`*r.fono,* r`)`

,donde Insertdesordenado opera igual que insertclasico pero ubica el dato en la última posición libre dentro de la hoja que le correspónde, los split operan igual.
(1,5) descripción general de la diferencia