IIC2133 – Estructuras de Datos y Algoritmos

Interrogación 1

Hora inicio: 14:00 del 8 de abril del 2020

Hora máxima de entrega: 13:00 del 9 de abril del 2020

0. Formalidad. Responde esta pregunta en papel y lápiz, incluyendo tu firma al final.

    a. ¿Cuál es tu nombre completo?
    b. ¿Te comprometes a no preguntar ni responder dudas de la prueba a nadie que no sea parte del cuerpo docente del curso, ya sea de manera directa o indirecta?

1. A la hora de ordenar usando $HeapSort$, es necesario primero que el arreglo esté estructurado como un heap. Considera la siguiente función que recibe un arreglo A de n elementos:

    $preparar\ para\ HeapSort(A, n)$:

        $for\ i = \lfloor \frac{n}{2} \rfloor .. 0$:

            $sift\ down(A, i)$

    a. Demuestra que $preparar\ para\ HeapSort$ deja el arreglo A estructurado como un heap.
    b. Justifica, mediante cálculos, que la complejidad de este algoritmo es $O(n)$.

2. En ocasiones, como vimos con $QuickSort$, los algoritmos pueden tener una componente aleatoria. Esto puede ser muy útil, ya que en el caso de $QuickSort$ significa que no podemos forzar el peor caso intencionalmente. En otros algoritmos, por ejemplo, puede ser un problema. Considera el siguiente algoritmo de ordenación que recibe un conjunto A de n elementos:

    $BogoSort(A, n)$:

        $while(true)$:

            Sea B un arreglo vacío

            $while$ A aún tenga elementos:

                Extraer aleatoriamente un elemento de A y ponerlo al final de B

            if B está ordenado:

                $return$ B

            Devolver todos los elementos de B a A

    a. Demuestra que $BogoSort$ no es correcto según la definición vista en clases. ¿Cuál es la lógica detrás de este algoritmo? ¿Qué habría que modificar en el algoritmo para que este sea correcto, manteniendo esta lógica?

    b. ¿Qué hace que $QuickSort$ sea correcto, pese a que toma decisiones aleatorias?

3. $\boldsymbol{MergeSort}$ utiliza la estrategia "dividir para conquistar" dividiendo los datos en 2 y luego resolviendo el problema recursivamente. Considera una variante de $\boldsymbol{MergeSort}$ que divide los datos en 3 y los ordena recursivamente, para luego combinar todo en un arreglo ordenado usando una variante de $\boldsymbol{Merge}$ que recibe 3 listas.

   a. Escribe la recurrencia $\boldsymbol{T(n)}$ del tiempo que toma este nuevo algoritmo para un arreglo de $\boldsymbol{n}$ datos. ¿Cuál es su complejidad, en notación asintótica?

   b. Generaliza esta recurrencia a $\boldsymbol{T(n,k)}$ para la variante de $\boldsymbol{MergeSort}$ que divida los datos en $\boldsymbol{k}$. ¿Cuál es la complejidad de este algoritmo en función de $\boldsymbol{n}$ y $\boldsymbol{k}$? Considera que la cantidad de pasos que toma $\boldsymbol{Merge}$ para $\boldsymbol{k}$ listas ordenadas, de $\boldsymbol{n}$ elementos en su totalidad, es $\boldsymbol{n \cdot \log_2(k)}$. Por ejemplo, si $\boldsymbol{k = 2}$, $\boldsymbol{Merge}$ toma $\boldsymbol{n}$ pasos, ya que $\boldsymbol{\log_2(2) = 1}$.

      Finalmente, ¿Qué sucede con la complejidad del algoritmo cuando $\boldsymbol{k}$ tiende a $\boldsymbol{n}$?

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Computación
IIC2133 – Estructuras de datos y algoritmos
Primer Semestre 2020

Pauta Interrogación 1

Problema 1

a)

Idea central: El algoritmo comienza desde el ultimo padre escalando hacia arriba del árbol ordenando cada subárbol como un heap (las hojas de estos son trivialmente heaps de un solo elemento). Esta propiedad es utilizada hasta llegar al primer elemento completando lo pedido en el enunciado. **(0.2 puntos)**

Demostración formal:

La presente demostración es parecida a una inducción. Se asume una proposición B. Se expone su caso base, y se muestra que si en un caso general (ciclo n) si la proposición B es verdadera en el siguiente caso (ciclo n+1) también lo será.

Proposición B: Todos los elementos mayores a i son la raíz un heap.

Concepto de termino y correctitud: Al llegar al primer elemento, como sus hijos son heaps el algoritmo dejará la totalidad del array estructurado como uno. **(proposición + concepto de termino y correctitud = 0.2 puntos)**

Caso base: Todos los elementos mayores a $\lfloor \frac{n}{2} \rfloor$ son hojas y trivialmente heaps. **(0,2 puntos)**

Mantenimiento: Como todos los hijos de i tienen una numeración mayor, debido a la proposición B tienen que ser raíces de heaps. Esta es precisamente la condición necesaria para que SiftDown cree un heap. Preservando la proposición B, para el siguiente ciclo: Todos los elementos mayores a $(i-1)$ son la raíz un heap. **(0.4 puntos)**

b)

La idea principal de por qué toma tiempo lineal (i.e. $\mathcal{O}(n)$) se basa en que la cantidad de pasos que toma la operación `siftDown` depende del nivel del árbol en el que estamos situados. El algoritmo toma $\mathcal{O}(log(n))$ cuando estamos en la raíz y $\mathcal{O}(1)$ cuando estamos en la penúltima hoja del árbol.

Sabemos que la cantidad de nodos por nivel está dado por la siguiente fórmula

$$nodos_h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

donde h es la altura del árbol ($log(n)$ para la raíz). Además, sabemos que la cantidad de pasos que tomará `siftDown` en el nivel de altura h será, en el peor caso, de h pasos. **(0.3 puntos)**

Teniendo lo anterior, podemos demostrar que la complejidad de nuestro algoritmo está dada por

$$\sum_{h=0}^{\lfloor log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h)$$

Es decir, la cantidad de nodos por nivel, multiplicados por la complejidad de aplicar el método `siftDown` a esos nodos nos dará la complejidad total de nuestro algoritmo. Es importante notar que podemos partir desde $h = 0$ dado que, si bien comenzamos desde la penúltima hoja, en la última hoja la expresión se multiplica por $h = 0$ (la cantidad de pasos de `siftDown` son 0). Resolviendo obtenemos

$$= \mathcal{O}\left(n \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^{h+1}} \right)$$

(0.3 puntos)

$$= \mathcal{O}\left(\frac{n}{2} \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^{h}} \right)$$

Dado que estamos en notación asintótica, podemos escribir la expresión anterior como

$$= \mathcal{O}\left(\frac{n}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^{h}} \right)$$

La suma converge a 2, por lo que nos queda

$$= \mathcal{O}\left(\frac{n}{2} \cdot 2 \right) = \mathcal{O}(n)$$

(0.4 puntos)

Problema 2

a)

Finitud: Como se extraen elementos de forma aleatoria de A a B, aun que un orden esté mal, esto no asegura que en un intento futuro no se valla a repetir el mismo orden. Por lo tanto no hay manera de demostrar que hay un avance en dirección al final. Por lo tanto el algoritmo falla en este ámbito.**(Max 0.4 puntos)**

(0.2 puntos) Menciona que puede no terminar.
(0 puntos) No menciona que puede no terminar.

(0.2 puntos) Menciona que ordenes incorrectos se pueden repetir.
(0.1 puntos) Menciona probabilidades(correcta o incorrectamente) y/o loops infinitos.
(0 puntos) Sin explicación.

Lógica: La lógica detrás de este algoritmo es generar permutaciones aleatorias de los datos, con la esperanza de obtener un arreglo ordenado. **Max 0.3 puntos**

(0.3 puntos) Menciona "permutaciones" explicitamente, y el probarlas.
(0.2 puntos) No menciona "permutaciones" pero tiene la idea correcta.
(0.1 puntos) Habla de aleatoriedad.
(0 puntos) Si no habla de la lógica del algoritmo.

Arreglo: La manera de mantener la aleatoriedad pero arreglar el problema de la finitud sería asegurar de alguna forma que una misma permutación, de los elementos del conjunto A, no se genere genere dos veces. En otras palabras, que no se repita la misma permutación más de una vez.**Max 0.3 puntos**

(0.3 puntos) Habla de que se use cada permutación solo una vez. (Sigue lógica de Bogosort)
(0.2 puntos) Usa otras lógicas, pero logra resolver el problema.
(0 puntos) Si no arregla el problema de bogosort.
(0 puntos) Si usa lógicas de ordenación conocidas (Inserionsort, Selectionsort) randomizadas.

Si el alumno le falta puntaje en esta pregunta puede conseguir estos puntos extra, mencionando ambos criterios(0.1 pto.) y explicando por que si cumple su fucnión cuando termina(0.1 pto.):

Criterios: Para que un algoritmo de ordenación sea correcto necesitamos ver que se cumplan dos cosas:

- ◇ Finitud: Termina en una cantidad finita de pasos.
- ◇ Correctitud: Cumple con su función al terminar. Como es un algoritmo de ordenación: Que al retornar B, este esté ordenado. **(0.1 puntos)**

Correctitud: Como sabemos que para retornar B se tiene que cumplir que "B está ordenado", entonces en esta parte el algoritmo revisa si B está ordenado. Si está ordenado, retorna B ordenado, por lo que cumple con su función. Si no está ordenado, no retorna, devuelve los elementos de B a A y vuelve a correr el while. Por lo tanto sabemos que si B es retornado, estará ordenado.**(0.1 puntos)**

b)

La aleatoriedad de Quicksort se encuentra en la elección del pivote para realizar la partición. Una vez que un elemento es elegido como pivote, éste queda en su posición correcta y no puede volver a ser elegido como pivote. Lo anterior implica que en cada iteración el número de elementos a ordenar va disminuyendo, por lo tanto, con un número de iteraciones finitas, el algoritmo termina para cualquier caso, lo que no sucede con Bogosort. **(1 punto)**

Problema 3

a)

Sabemos que $Merge$ funciona en $O(n)$, y que *MergeSort* funciona en $O(1)$ para un solo elemento, y que para un input n, esta variable llamará recursivamente a *MergeSort* tres veces, con inputs $\lceil \frac{n}{3} \rceil$, $\lfloor \frac{n}{3} \rfloor$ y $n - \lfloor \frac{n}{3} \rfloor - \lceil \frac{n}{3} \rceil$para después unir las 3 con *Merge*. Por lo tanto, la ecuación de recurrencia quedaría:

$$T(n) = \begin{cases} 1 & \text{if n} = 1 \\ T\left(\lceil \frac{n}{3} \rceil\right) + T\left(\lfloor \frac{n}{3} \rfloor\right) + T\left(n - \lceil \frac{n}{3} \rceil - \lfloor \frac{n}{3} \rfloor\right) + n & \text{if n} > 1 \end{cases}$$

Alternativamente:

$$T(n) \leq \begin{cases} 1 & \text{if n} = 1 \\ 3 * T\left(\lceil \frac{n}{3} \rceil\right) + n & \text{if n} > 1 \end{cases}$$

(0.5 pts)
Se descuenta 0.1 si solo indica $\frac{n}{3}$, sin resolver la divisón entera, si indica mal la recursión o si indica la cota superior como igualdad.

Para la complejidad asintótica tenemos dos opciones, utilizar el teorema maestro, o resolver la recurrencia reemplazando recursivamente.
El teorema maestro resuelve recurrencias de la forma:

$$T(n) = a \cdot T\Big(\frac{n}{b}\Big) + f(n)$$

Donde:

- $\diamond$ n es el tamaño del problema.
- $\diamond$ a es el número de subproblemas en la recursión.
- $\diamond$ $\frac{n}{b}$ el tamaño de cada subproblema.
- $\diamond$ $f(n)$ es el costo de dividir el problema y luego volver a unirlo.

En este caso, podemos acotar la recurrencia por arriba, sabiendo que cada subllamada tendrá a lo más $\lceil \frac{n}{3} \rceil$ elementos, por lo que podemos decir que:

$$T(n) \leq 3 * T\Big(\lceil \frac{n}{3} \rceil\Big) + n$$

Aquí tenemos que $a = b = 3$, y $f(n) = n$, y tenemos que $f(n) \in \Theta(n^{log_b a}) = \Theta(n^{log_3 3}) = \Theta(n)$, por lo tanto, según el teorema maestro (caso 2),

$$T(n) \in O\big(n \cdot log(n)\big)$$

(Opción 1: 0.5 pts)

Para resolver esta recurrencia reemplazando recursivamente buscamos un k tal que $n \leq 3^k < 3n$. Se cumple que $T(n) \leq T(3^k)$. Como $\lceil \frac{3^k}{3} \rceil = \lfloor \frac{3^k}{3} \rfloor = 3^k - \lceil \frac{3^k}{3} \rceil - \lfloor \frac{3^k}{3} \rfloor$, podemos entonces, reescribir la recurrencia de la siguiente forma:

$$T(n) \leq T(3^k) = \begin{cases} 1 & \text{if k} = 0 \\ 3^k + 3 \cdot T(3^{k-1}) & \text{if k} > 0 \end{cases}$$

Expandiendo la recursión:

$$\begin{aligned} T(n) \leq T(3^k) &= 3^k + 3 \cdot [3^{(k-1)} + 3 \cdot T(3^{k-2})] && (1) \\ &= 3^k + [3^k + 3^2 \cdot T(3^{k-2})] && (2) \\ &= 3^k + 3^k + 3^2 \cdot [3^{k-2} + 3 \cdot T(3^{k-3}) && (3) \\ &= 3^k + 3^k + 3^k + 3^3 \cdot T(3^{k-3}) && (4) \\ &... && (5) \\ &= i \cdot 3^k + 3^i \cdot T(3^{k-i}) && (6) \end{aligned}$$

cuando $i = k$, por el caso base tenemos que $T(3^{k-i}) = 1$, con lo que nos queda

$$T(n) \leq k \cdot 3^k + 3^k \cdot 1$$

Ahora, tenemos que volver a nuestra variable inicial n. Por construcción de k:

$$3^k < 3n$$

Tenemos entonces que

$$T(n) \leq k \cdot 3^k + 3^k < \log_3(3n) \cdot 3n + 3n$$

Por lo tanto

$$T(n) \in \mathcal{O}\big(n \cdot \log_3(n)\big) = \mathcal{O}\big(n \cdot \log(n)\big)$$

(Opción 2: 0.5 pts)
Se descuenta 0.1 si no utiliza la base del *log* en el reemplazo o el desarrollo, ya que es la diferencia de profundidad con MergeSort normal.

b)

Para esta pregunta hay mas de una solución ya que no era necesario realizar una demostración formal, igualmente en esta solución se incluye una explicación mas formal.

Primera solución

Una de las soluciones para generalizar la recurrencia de $\mathbf{T(n,k)}$ seria indicar en primer lugar que la función que modela la recurrencia para este caso sería para n > 1

Se divide el arreglo en k arreglos de al menos $\lceil \frac{n}{k} \rceil$ elementos.

$$T(n,k) \leq \underbrace{log_2(k) \cdot n}_{\text{Costo de realizar merge para k arreglos ordenados}} + \overbrace{T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + ... + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)}$$

Y para n= 1

$$T(1,k) = 1$$

Ahora bien, esto es equivalente a decir

$$T(n,k) \leq log_2(k) \cdot n + k \cdot T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)$$

Si se reemplaza n por $n \leq k^y < k \cdot n$ quedara

$$T(n,k) \leq T(k^y,k) = log_2(k) \cdot k^y + k \cdot T\left(k^{y-1}, k\right)$$

Y de manera recursiva quedara

$$T(k^y,k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot T\left(k^{y-2}, k\right))$$

Quedando finalmente

$$T(k^y,k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot (log_2(k) \cdot k^{y-2} + ... + (k^{y-y} \cdot log_2(k) + k \cdot T\left(1,k\right))))$$

Que en otras palabras es

$$T(k^y,k) = y \cdot k^y \cdot log_2(k) + k^y$$

Y por la condicion que se establecio en la definición de k^y, notar que

$$k^y < k \cdot n / log_k$$

$$y < log_k(k \cdot n) = \frac{log(kn)}{log(k)}$$

Por tanto quedara

$$T(n,k) \leq T(k^y) < \left(\frac{log_2(n)}{log_2(k)} + 1\right) \cdot n \cdot k \cdot log_2(k) + n \cdot k$$

Reordenando

$$T(n,k) < log_2(n) \cdot n \cdot k + n \cdot k \cdot (log_2(k) + 1)$$

A partir de esto se puede concluir que

$$\therefore T(n,k) \in O(k \cdot n \cdot log(n))$$

Tambien es valido haber indicado como posible orden

$$T(n,k) \in O(k \cdot n \cdot log(n \cdot k))$$

- **0.75 pts** por explicación y/o mostrar de manera correcta el orden de complejidad
- **0.6 pts** Por explicación y/o mostración correcta pero orden de complejidad incorrecto.
- **0.3 pts** Por explicación y/o mostración con errores mayores
- **0 pts** Por explicación y/o mostración incorrecta

Segunda solución

Se explica a traves de un desarrollo correcto que el orden de complejidad es $O(n * log(n))$
Como por ejemplo

$$\sum_{i=0}^{log_k(n)} log_2(k) \cdot \frac{n}{k^i} + k^{i+1} T(\frac{n}{k^{i+1}}, k)$$

$$\frac{log(k)}{log(2)} n \frac{log(n)}{log(k)} + k^{log_k(n)+1}$$

$$n * log_2(n) + n * k$$

.

- **0.75 pts** por explicación y/o mostrar de manera correcta el orden de complejidad
- **0.6 pts** Por explicación y/o mostración correcta pero orden de complejidad incorrecto.
- **0.3 pts** Por explicación y/o mostración con errores mayores
- **0 pts** Por explicación y/o mostración incorrecta

Para el caso de la complejidad del algoritmo para el caso que k tienda a n, es claro que la complejidad tendera a converger a $O(n \cdot log(n))$. Es claro si se reemplaza en la ecuación de recursión T(n,n).

- **0.25 pts** Por explicación correcta.
- **0.1 pts** Por explicación con intuición correcta pero equivocada respecto al orden de complejidad.
- **0 pts** Por explicación incorrecta.