CC4102 - Examen

Profs. Benjamin Bustos y Gonzalo Navarro

11 de Diciembre de 2023

P1 (1.5 pt) Reducción

Sean $A = [a_1, \ldots, a_n]$ y $B = [b_1, \ldots, b_n]$ dos secuencias desordenadas de números, cada una de ellas con n valores. El problema del *emparejamiento* consiste en emparejar un valor de A con un valor de B de forma que el menor valor de A se empareje con el menor valor de B, el segundo menor valor de A se empareje con el segundo menor valor de B, y así sucesivamente, hasta emparejar el mayor valor de A con el mayor valor de B. El output del problema de emparejamiento es la secuencia correspondiente de parejas (a_i, b_j), en algún orden.

Demuestre que el problema del emparejamiento requiere tiempo $\Omega(n \log n)$ si se procede por comparaciones. Hint: considere reducción usando un problema parecido.

Solución: # Arreglo Y esta ordenado [1, 2, ..., n], por tanto Y[j]=j se puede usar como índice para reconstruir el output y transformar el arreglo resultante al que se obtendría por ordenamiento.

Reducimos del problema de ordenar al de emparejar. Sea X el arreglo a ordenar. Generar un arreglo Y del mismo tamaño con los números $[1, \ldots, n]$, lo que toma tiempo $O(n)$. Resolver el problema del emparejamiento (donde $x_i = X[i]$ e $y_j = Y[j] = j$). Finalmente, recorrer las parejas (x_i, y_j) escribiendo x_i en la posición $y_j = j$ del arreglo output, lo que también toma tiempo $O(n)$.

Transformar desde el problema de ordenamiento al de emparejamiento y de vuelta toma tiempo $O(n) = o(n \log n)$. Si el algoritmo de emparejamiento tomara tiempo $o(n \log n)$, se podría ordenar en tiempo $o(n \log n)$, lo que rompería la cota inferior de ordenamiento basado en comparaciones. Por lo tanto, cualquier algoritmo basado en comparaciones que resuelva el problema de emparejamiento debe tomar tiempo $\Omega(n \log n)$ (no necesitan hacer este razonamiento para tener la nota completa, se supone que ya se sabe de clases).

Cuando se dice O(n)=o(n logn) significa que el tiempo O(n) crece estrictamente más lento que n logn.

P2 (1.5 pt) Memoria

Se tienen dos tablas, $R(x, y)$ y $S(y, z)$ y se desea calcular su join natural:

$$T \;=\; R \bowtie S \;=\; \{(x, y, z),\ (x, y) \in R \wedge (y, z) \in S\}.$$

Cada tabla se encuentra almacenada en disco, sin orden, y el resultado del join también se debe almacenar en disco, sin un orden particular. Llamaremos r, s y t, respectivamente, a la cantidad de enteros que forman las tablas R, S y T. Suponga que ninguna de las tablas cabe en memoria principal. Recuerde que la memoria principal puede almacenar M enteros y los bloques de disco B enteros.

1. Diseñe y analice un algoritmo que genere T en $O((r+s+t)/B + rs/(MB))$ I/Os. Hint: mejore la idea básica del nested-loop: for $(x_r, y_r) \in R$, for $(y_s, z_s) \in S$, if $y_r = y_s$ output (x_r, y_r, z_s).
2. Diseñe y analice otro algoritmo que ordene R y S, y luego haga $O((r + s + t)/B)$ I/Os.
3. Suponga que $r = s = n$ para simplificar, ¿qué criterio usaría para elegir entre el primer y el segundo algoritmo?

núm. grupos = techo(r/ (M/2)) = 2r/M => equivalente a O(r/M)
En este análisis asintótico O(...) incluso si r no es múltiplo exacto de M/2, puede quedar un grupo más pequeño adicional => O(1 + r/M)
#s: núm. de filas en la tabla S; B: tam. de un bloque de disco => lecturas para leer todo S = s/B

Solución:

En la primera (0.5pt), puedo ir leyendo R de a grupos de algo menos de $M/2$ filas (dejando $O(B)$ memoria libre para buffers). Por cada grupo, leo todo S de a buffers de $B/2$ filas. Comparo cada fila de cada buffer de S con todo el grupo de R, y voy escribiendo los resultados que pertenezcan a T en otro buffer, que voy escribiendo a disco cuando se va llenando. Por cada grupo de R, de los cuales hay $O(1 + r/M)$, leo todo S, que requiere s/B lecturas. Esto totaliza $O(s/B + rs/(MB))$. Además realizo r/B lecturas para traer los grupos de R a memoria, y t/B escrituras generando T.

En la segunda (0.5pt), luego de ordenar R y S en tiempos $sort(r) + sort(s)$, paso por ambos archivos secuencialmente (usando en total $O(r/B + s/B)$ I/Os), y cuando detecto valores iguales de y en ambos archivos, debo enviar a T el producto cartesiano de cada x y cada z asociados a ese y. Estos grupos pueden no caber en memoria, sin embargo. Supongamos que hay n_x valores de x y n_z valores de z. Hago algo parecido a la primera parte, leyendo los n_x valores de x en grupos de $M/2$ y pasando por todos los valores de n_y por grupo. Esto cuesta $O(n_x/B + n_y/B + n_x n_y/(MB))$, más los $O(n_x n_y/B)$ I/Os para escribir los resultados. Los n_x suman r, los n_y suman s, y los $n_x n_y$ suman t, por lo que el total es $O((r + s + t)/B)$. # El M de nxny/MB se simplifica en los costos asintóticos

Para compararlos (0.5pt), el punto es que el segundo algoritmo tiene mejor costo asintótico: si eliminamos el costo imprescindible $O((r + s + t)/B)$, nos queda comparar $O(rs/(MB)) = O(n^2/(MB))$ con $sort(r) + sort(s) = O((n/B) \log_{M/B}(n/M))$, es decir, n/M versus $\log_{M/B}(n/M)$: el primero siempre es mayor. Basta con que se den cuenta de eso para tener toda la nota. También podrían argumentar que, si n no es tanto mayor que M, preferirían el primer método, lo cual también está bien. No necesitan hacer un análisis formal, basta con que puedan argumentar que el primero es mejor para n/M no tan grande y el segundo para n/M mayor.

P3 (1.5 pt) Costo Amortizado

Cuando vimos cómo incrementar un número binario de k bits a costo amortizado $O(1)$, no consideramos la operación de decrementar. Para permitir esta operación, vamos a usar *trits* en vez de bits, donde un trit tiene tres valores posibles: -1, 0, ó 1. La secuencia de k trits $t_{k-1} \cdots t_0$ representa el número $\sum_{i=0}^{k-1} t_i \cdot 2^i$. Los números se pueden representar de muchas formas distintas usando trits, por ejemplo $7 = 111$, pero también $7 = 100(-1)$. También se pueden representar números negativos, como $-1 = 000(-1)$ ó $-1 = (-1)111$.

1. (0.5pt) Demuestre que, usando bits, no es posible obtener costo amortizado constante si se permiten incrementos y decrementos.

2. (1.0pt) Diseñe un algoritmo para expresar incrementos y decrementos usando trits, y demuestre que permite costo amortizado constante para ambas operaciones si se parte del número cero. Como hint, piense cómo sumar 1 a las dos representaciones de -1 vistas para obtener 0000, cómo restar 1 a $-7 = 0(-1)(-1)(-1)$ para obtener $-8 = (-1)000$, y cómo restar 1 a $1 = 1(-1)(-1)(-1)$ para obtener 0000.

Solución:

Para el primer punto, basta con que muestren que, por ejemplo, si llegamos al número $2^{k-1} - 1$, una secuencia alternada de incrementos y decrementos exige modificar los k bits del número, resultando en costo amortizado $\Theta(k)$: la representación *debe* variar entre 01^{k-1} y 10^{k-1}.

El núm. más grande representable con k bits es 2^k - 1. Este se presenta como 11...1 (k veces 1)
Incrementar y decrementar implica que:
Partimos en 2^k - 1 = 11...1
Incrementamos: 11...1 -> 10...0 (k cambios)
Decrementamos: 10...0 -> 01...1 (k cambios)
Por ende, cada incremento/decremento requiere cambiar los k bits.
Costo amortizado: T(n)/n = Costo total/ núm. operaciones = k*n/n = k y por tanto el costo amortizado es O(k), que no es constante.

Para el segundo, se puede incrementar de la siguiente manera: de derecha a izquierda, mientras el dígito sea 1, se convierte a 0 y se continúa. Al llegar a un $d \neq 1$, se lo reemplaza por $d+1$ (es decir, un 0 se convierte en 1, y un -1 se convierte en 0). Para decrementar es análogo: mientras el dígito sea -1, se convierte a 0 y se continúa. Al llegar a un $d \neq -1$, se lo reemplaza por $d-1$. (0.5 pt por diseñar algo que funcione).

Para demostrar que este esquema funciona (0.5 pt), se puede extender el método de contabilidad de costos o el de función potencial. Para el primero: como todo 1 o -1 fue un 0 alguna vez, las transformaciones de 0 a 1 o a -1 cuestan 2, las opuestas cuestan cero. Como el incremento o el decremento tienen a lo sumo una operación de costo 2, su costo amortizado es ≤ 2.

En el caso de usar función potencial, pueden definir ϕ como la suma de los dígitos 1 y -1. Al incrementar un número terminado en c 1s, el costo real es $c+1$, y la diferencia de potencial puede ser $-c+1$ (si el dígito final es un 0) ó $-c-1$ (si el dígito final es un -1). Así, el costo amortizado es 2 ó 0. El caso de decrementar es similar, considerando un número terminado en c (-1)s.

P4 (1.5 pt)

Suponga que dispone de n objetos $\{x_1, \ldots, x_n\}$, cada uno de ellos de tamaño t_i, con $0 < t_i < 1$. Suponga también que dispone de cajas de tamaño unitario. Cada caja puede contener un número arbitrario de objetos, siempre que su capacidad no sea excedida (es decir, la suma de los tamaños de los objetos que contiene la caja no puede exceder 1). Se desea encontrar el mínimo número de cajas necesarias para poder guardar los n objetos.

Incluso la versión de decisión de este problema es NP-completa. Se propone entonces el siguiente algoritmo para decidir cómo repartir los objetos en las cajas (que no necesariamente retorna la respuesta óptima):

```
Numero de cajas utilizadas = 0
for i = 1 to n
    Poner objeto x_i en la primera caja en donde quepa
    Si no cabe en ninguna
        Agregar una nueva caja (incrementando numero de cajas utilizadas)
        Colocar x_i en esta nueva caja
return numero de cajas utilizadas
```

Sea $T = \sum_{i=1}^{n} t_i$ y suponga por simplicidad que $T > 1$ (es decir, no caben todos los elementos en una sola caja, en que el problema es trivial).

i. Muestre que el número óptimo de cajas necesarias es al menos $\lceil T \rceil$.

ii. Muestre que al utilizar el algoritmo propuesto a lo más hay una caja que esté ocupada en menos que la mitad de su capacidad.

iii. Muestre que si el algoritmo propuesto ocupa C cajas, entonces $T \geq C/2$. Note que esto sería trivial por ii si no fuera por la caja posiblemente ocupada a menos de la mitad.

iv. Concluya que el algoritmo propuesto es una 2-aproximación.

Solución:

i. Supongamos que disponemos de sólo $\lceil T \rceil - 1$ cajas. Notar que la capacidad total de estas cajas es $\lceil T \rceil - 1 < T$. En el mejor caso, se llenarán completamente dichas cajas, pero dado que la capacidad es estrictamente menor que T, al menos quedará un objeto sin poder guardarse en una caja. Por lo que se requiere de al menos una caja extra para poder guardar todos los objetos, es decir, $\lceil T \rceil - 1 + 1 = \lceil T \rceil$ cajas. [Nota: esta respuesta es bien exhaustiva, pero para asignar todo el puntaje basta con cualquier explicación intuitiva "razonable"].

ii. Sean dos cajas, X e Y, utilizadas en $\leq 1/2$. Supongamos que Y se creó después de X. Entonces, Y se creó para agregar un elemento con $t_i \leq 1/2$. Pero el algoritmo no puede haber hecho esto, dado que existía la caja X, ya en ese momento utilizada en $\leq 1/2$, por lo que t_i habría cabido en X. [Hay distintas explicaciones válidas, esta es posiblemente la más clara, pero otras pueden tener todo el puntaje aunque sean más enredadas.]

iii. Por ii, cada caja usada por el algoritmo está llena al menos hasta $1/2$, con la posible excepción de una, que podría estar llena hasta $0 < \epsilon < 1/2$. Pero en ese caso, por un argumento similar a ii, todas las otras cajas deben estar llenas a más de $1 - \epsilon$, por lo cual la caja llena a ϵ y cualquier otra caja juntas suman una capacidad de al menos $1 - \epsilon + \epsilon = 1$. Las otras $C - 2$ cajas suman al menos $(C - 2)/2$, totalizando entre todas al menos $(C - 2)/2 + 1 = C/2$. Por lo tanto debe ser que $T \geq C/2$.

iv. Por iii, la cantidad de cajas usadas por el algoritmo propuesto es $C \leq 2T$, mientras que por i, el algoritmo óptimo necesita $\lceil T \rceil \geq T$ cajas. Por definición de ρ-aproximación, el algoritmo propuesto es una $(2T/T) = 2$-aproximación.

Tiempo: 3 horas Con tres hojas de apuntes