Universidad de Chile
Departamento de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Auxiliar 2 - "Cotas inferiores: Teoría de la Información y Reducciones"

Profesores: Pablo Barceló
Gonzalo Navarro
Auxiliar: Dustin Cobas

P1. Arreglo *k*-ordenado

Decimos que un arreglo $A[1 \dots n]$ está **k-ordenado** si puede ser dividido en k bloques, cada uno de tamaño $\frac{n}{k}$ (asumimos que $\frac{n}{k}$ es un entero), de modo que los elementos de cada bloque son mayores que elementos de bloques anteriores. Los elementos en cada bloque no necesitan estar ordenados.

a) Describa un algoritmo que "k-ordene" un arreglo arbitrario en tiempo $\mathcal{O}(n \log k)$.

b) Muestre que cualquier algoritmo basado en comparaciones que k-ordene, requiere $\Omega(n \log k)$ comparaciones en el peor caso.

c) Describa un algoritmo que ordene un arreglo k-ordenado en tiempo $\mathcal{O}\left(n \log\left(\frac{n}{k}\right)\right)$.

d) Muestre que cualquier algoritmo basado en comparaciones que ordene un arreglo k-ordenado requiere $\Omega\left(n \log\left(\frac{n}{k}\right)\right)$ comparaciones.

P2. Tuercas y Tornillos

Nos dan una fila de n tuercas y otra de los n tornillos que les corresponden, pero en cualquier orden. Visualmente no podemos comparar dos tuercas o dos tornillos, pero sí podemos probar una tuerca con un tornillo y decidir cuál es mayor que la otra, o si se corresponden.

a) Demuestre que el problema de hacer corresponder a cada tuerca con su tornillo es $\Omega(n \log n)$ en comparaciones tuerca-tornillo incluso en promedio.

b) Muestre un algoritmo que realice $\mathcal{O}(n \log n)$ comparaciones en promedio.

P3. Distinción de Elementos

Utilice un argumento similar a una reducción para dar una cota inferior de $\Omega(n \log n)$ (en el modelo de comparaciones) para el problema de determinar si n enteros son diferentes.

Soluciones

P1. Arreglo *k*-ordenado

a) Supongamos para simplificar que $k = 2^r$. Para resolver el problema utilizaremos una modificación del algoritmo de quicksort, y un algoritmo que encuentra la mediana en tiempo lineal [1]. Para esto utilizaremos:

- `particionar(A, i, j, p)`: *particionar* $A[i:j]$ usando el pivote en p.
- `posicionMediana(A, i, j)` : devuelve el índice de la mediana en $A[i:j]$.

Con esto construimos el algoritmo *kSort(A, i, j, k)* que k-ordena el arreglo $A[i,j]$:

```
Funcion kSort (A, i, j, k)
    if k = 1 then
        return
    end
    particionar(A, i, j, posicionMediana(A, i, j))
    kSort(A, i, (i+j)/2, k/2)
    kSort(A, (i+j+2)/2, j, k/2)
```

Finalmente, la profundidad de la recursión generada a partir de `kSort(A, 1, n, k)` es $\log k$ (pues el parámetro k se divide a la mitad cada vez) y en cada nivel se hace trabajo lineal en el tamaño del arreglo, por lo que este algoritmo toma tiempo $\mathcal{O}(n \log k)$.

b) Las hojas del árbol de decisión respectivo contienen conjuntos de permutaciones que tienen los mismos elementos en los mismos bloques. Podemos contar esto considerando todas las permutaciones posibles y descontando los órdenes dentro de los grupos de tamaño $\frac{n}{k}$. Así, el árbol de decisión correspondiente tendrá $\frac{n!}{\left(\left(\frac{n}{k}\right)!\right)^k}$ hojas y su altura debe ser $\geq \log \frac{n!}{\left(\left(\frac{n}{k}\right)!\right)^k} = \log n! - \log\left(\left(\frac{n}{k}\right)!\right)^k = n \log n - k\frac{n}{k} \log \frac{n}{k} + \mathcal{O}(n) = \Omega(n \log k)$ comparaciones.

c) Para esto basta con ordenar, con un algoritmo de ordenación óptimo, cada uno de los k bloques a un costo de $\mathcal{O}\left(\frac{n}{k} \log \frac{n}{k}\right)$ cada uno, teniendo de esta forma un costo total de $\mathcal{O}\left(n \log \frac{n}{k}\right)$.

d) Las hojas del árbol de decisión respectivo contienen cada uno de los $\left(\left(\frac{n}{k}\right)!\right)^k$ posibles inputs y por lo tanto una cota inferior para el problema será $\log\left(\left(\frac{n}{k}\right)!\right)^k = k\left(\frac{n}{k} \log \frac{n}{k} + \mathcal{O}\left(\frac{n}{k}\right)\right) = \Omega\left(\frac{n}{k} \log \frac{n}{k}\right)$ comparaciones.

P2. Tuercas y Tornillos

[1] Un algoritmo de búsqueda de media en tiempo lineal se basa en la elección de pivote de mediana de medianas, una explicación del algoritmo se puede encontrar en https://users.dcc.uchile.cl/~mcaceres/MedianFinding.pdf

a) Podemos interpretar el input de este problema como dos arreglos (uno de tuercas y uno de tornillos), por lo que el número de posibles input es $(n!)^2$. Luego, si consideramos que cada uno de los input se presenta con igual probabilidad y que las comparaciones realizadas por el algoritmo codifican aquel input podemos usar el Teorema de Shannon para mostrar que el algoritmo no puede hacer menos de $\log n!^2 = \Omega(n \log n)$ comparaciones en promedio.

b) Lo que hacemos es tomar un tornillo y particionar el arreglo de tuercas según ese tornillo. Con esto, tendremos las tuercas menores a ese tornillo a la izquierda, las mayores a la derecha y la tuerca que calza con el tornillo, con esta última tuerca particionamos los tornillos con lo que finalmente tendremos 2 subproblemas (el de la izquierda y el de la derecha) que resolvemos recursivamente.

   Para concluir, notamos que la división de problemas en subproblemas es idéntica a la de QuickSort con la diferencia que en nuestra división hacemos 2 "particionar" en vez de 1, lo que aumenta el numero de comparaciones al doble, pero no cambia la complejidad promedio de $\mathcal{O}(n \log n)$ de QuickSort.

P3. Distinción de Elementos

Supongamos por contradicción que existe un algoritmo A que resuelve "Distinción de Elementos" en $o(n \log n)$ comparaciones.

Construiremos otro algoritmo que ordena en $o(n \log n)$ comparaciones en base a las comparaciones que realiza A, lo que será una contradicción dada la cota inferior conocida de "ordenar".

Antes veamos que si etiquetamos los elementos del input de menor a mayor, $a_1, a_2, \ldots, a_n$, A debe haber comparado a_{i-1} con $a_i \forall i$, si no un adversario puede construir un arreglo con todos distintos salvo $a_{i-1} = a_i$, y otro donde estos dos se hacen diferir infinitesimalmente, conservando todas las demás relaciones de orden. Así la relacion entre ellos dos no se puede inferir sin compararlos directamente.

Consideremos ahora el grafo dirigido de comparaciones (aquel donde los nodos son los elementos del input y hay un arco de u a v si es que ambos se compararon y u resultó ser menor a v). Este grafo no tiene ciclos dirigidos (pues si tuviese uno el input sería inconsistente), por lo que podemos encontrar un orden topológico de los nodos del grafo que de hecho corresponde a $a_1, a_2, \ldots, a_n$, por la propiedad del párrafo anterior.

Con esto nuestro algoritmo, ejecuta A ($o(n \log n)$), con las comparaciones realizadas construye el grafo de comparaciones ($o(n \log n)$) y averigua el orden topológico del grafo ($o(n \log n)$), con lo que ordena el input en $o(n \log n)$ comparaciones, lo que es una contradicción.