Universidad de Talca
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Diseño y análisis de algoritmos

Tarea 1

Erik Regla
eregla09@alumnos.utalca.cl

20 de Junio del 2015

1. Problema 1

1.1. a

$$\begin{aligned} lg^k(n) &\quad ? \quad n^\epsilon \\ lg^k(n) &\quad ? \quad n^{c-1} \\ nlg^k(n) &\quad ? \quad n^c \end{aligned}$$

Por tanto, para $k = 1$, $A = O(B)$, y en otro caso, $A = \Omega(B)$.

1.2. b

$A = \Omega(B)$

1.3. c

$A = \omega(B)$ debido a que B es oscilante.

1.4. d

$A = o(B)$

1.5. e

$$\begin{aligned} n^{lg(c)} &\quad ? \quad c^{lg(n)} \\ log_2(c)log_c(n) &\quad ? \quad log_2(n) \\ log_c(n) &\quad ? \quad log_2(n-c) \end{aligned}$$

Por tanto, para $c > 2$, $A = \omega(B)$, y en otro caso, $A = O(B)$.

1.6. f

$A = \omega(B)$

2. Problema 2

2.1. a

$$a + bi \times c + di = ac + adi + bci + bi + di$$

$$ac + adi + bci + bi + di = \overbrace{-bc - bd}^{\alpha} + \overbrace{2(ca + cb)}^{\beta+\beta} - \overbrace{ac + ad}^{\gamma}$$

$$-bc - bd + 2(ca + cb) - ac + ad = -bc - bd + ca + cb + ca + cb - ac + ad$$

$$-bc - bd + ca + cb + ca + cb - ac + ad = (ac - bd) + (bc + ad)$$

$$(ac - bd) + (bc + ad) = \overbrace{-b(c + d)}^{\alpha} + \overbrace{c(a + b)}^{\beta} + \overbrace{c(a + b)}^{\beta} + \overbrace{a(-c + d)}^{\gamma}$$

$$(ac - bd) + (bc + ad) = \underbrace{(\alpha + \beta)}_{\text{parte real}} + \underbrace{(\beta + \gamma)i}_{\text{parte imaginaria}}$$

Siendo solo necesario computar las multiplicaciones respectivas para α, β y γ las cuales son tres en total.

2.2. b

Basado en *RadixSort*, es posible ordenar los enteros en tiempo lineal dependiendo solo de la cantidad de bits que este tenga (asumiendo una implementación para representación binaria). Sin embargo, se puede tomar ventaja del algoritmo de selección, dado que si se conoce de antemano el valor de n entonces, para cada pivotaje se puede calcular la cantidad de números a ambos lados.

	G	1			G	2			G	3		G	4	G5
0	0	0	0	0	0	0	0							
0	0	0	1	0	0	0	1							
0	0	1	0	0	0	1	0							
0	0	1	1	0	0	1	1							
0	1	0	0	0	1	0	0							
0	1	0	1	0	1	0	1							
0	1	1	0	0	1	1	0							
0	1	1	1	0	1	1	1							
1	0	0	0											
1	0	0	1	1	0	0	1	0	0	1	0	1	1	
1	0	1	0	1	0	1	0	0	1	0	1	0		
1	0	1	1	1	0	1	1	0	1	1	1	1		
1	1	0	0	1	1	0	0	1	0	0				
1	1	0	1	1	1	0	1	1	0	1				
1	1	1	0	1	1	1	0	1	1	0				

Tabla 1: Ejemplos de búsqueda para un conjunto de números

Tómese como ejemplo el conjunto $G1$ provisto en la Tabla 2.2. Del conjunto $G2$, claramente se puede identificar que falta el número 8. Se sabe que el arreglo que contiene un 0 en el bit más significativo debe ser de 8 elementos de longitud y a su vez, el que contiene un 1 en su bit más significativo también debe de ser 8. Dado que la última condición no se cumple, se pasa al conjunto $G3$. Del cual nuevamente, que de cumplir con la condición de partición basada en *RadixSort* debería contener igual cantidad de elementos tanto en el lado de los 0's como de los 1's para el bit más significativo. Ya para $G4$ al volver a reducir nuevamente hasta llegar a $G5$ el complemento del bit existente determina la posición del elemento en cuestión. Luego, solo es necesario devolver la posición del índice en donde dicho número se encuentra.

El método anterior es válido tanto para el caso donde el arreglo se encuentra ordenado o desordenado, solo diferenciando en que si está ordenado, basta con realizar una búsqueda binaria sobre el bit.

RadixSort opera en $O(kn)$, donde k representa la cantidad de bits usada en la representación del número. Dado que es posible realizar un ordenamiento en tiempo lineal y una búsqueda en tiempo logarítmico, el orden del método presentado sigue siendo lineal.

2.3. Problema 3

2.4. f

$$\mathrm{T(n)} = \mathrm{T}\left(\tfrac{n}{2}\right) + \mathrm{T}\left(\tfrac{n}{4}\right) + \mathrm{T}\left(\tfrac{n}{8}\right) + \mathrm{n}\ ,\ \mathrm{T(1)} = 0,\ \mathrm{T(2)} = 1,\ \mathrm{T(4)} = 2.$$

Con n = 2^k, y r(k) = T(2^k),

$$\begin{aligned} r(k) &= r(k-1) + r(k-2) + r(k-3) + 2^k \\ \text{Avanzando en 3:} \quad r(k+3) &= r(k+2) + r(k+1) + r(k) + 8 \cdot 2^k \end{aligned}$$

Y aplicamos $\mathcal{G}$:

$$\begin{aligned} \frac{r(z) - r_0 - r_1 - r_2}{z} &= \frac{r(z)-r_0-r_1}{z} + \frac{r(z)-r_0}{z} + r(z) + \frac{8}{1-2z} \\ r(z) &= -\frac{8}{(1-2z)(1+z)} - \frac{2}{1+z} \end{aligned}$$

Separando fracciones se obtiene:

$$\begin{aligned} r(z) &= -\frac{\frac{8}{3}}{1-2z} + \frac{\frac{2}{3}}{1+z} \\ \text{Con } n = 6^k \quad k &= \log_6 n \text{ y aplicando } \mathcal{G}^{-1} \\ T(n) &= -\frac{8}{3}n + \frac{2}{3}(-1)^{\log_2 n} \end{aligned}$$

3. Problema 4

$$\begin{aligned} \sum_{i=0}^{n} i^2 &= \underbrace{\mathcal{G}(n^2)} \cdot \mathcal{G}(1) \\ &= \frac{z+z^2}{(1-z)^3}\frac{1}{(1-z)} \\ &= \frac{z+z^2}{(1-z)^4} \\ &= \frac{z(1-z)}{(1-z)^4} \\ &= \frac{z}{(1-z)^4} + \frac{z^2}{(1-z)^4} \\ &= \binom{n+2}{3} + \binom{n+1}{3} \\ &= \frac{(n+2)!}{(n-1)!3!} + \frac{(n+1)!}{(n-2)!3!} \\ &= \frac{n(n+1)(2n+1)}{6} \end{aligned}$$

4. Problema 5

4.1. a

4.1.1. Prueba

Sea c_i el costo de la tarea i, y asumiendo que $c_i < c_{i+1}$ definimos T_1 y T_2 como los tiempos de ejecución para un conjunto de procesos ordenados crecientemente y otro idéntico a este pero con

un elemento cambiado entre las posiciones i y j respectivamente.

$$Tn = T_1 + T_2 + T_3 + ... + T_n, T_i = c_i + T_{i-1} \forall i \in \mathbb{N}$$
$$T_1 = nC_1 + (n-1)c_2 + ... + c_n$$
$$T_2 = nC_1 + (n-1)c_2 + ... + (n-i+j)c_i + ... + (n-i)c_{i-j} + ... + c_n$$
$$T_2 - T_1 = (n-i+j)c_i + (n-i)c_{i-j} - ((n-i+j)c_{i-j} + (n-i)c_i) = jc_i - jc_{i-j} \geq 0$$

4.1.2. Algoritmo

Algoritmo 1 Secuenciamiento mínimo en $O(nlog(n))$

Precondición: T un arreglo de tareas
Postcondición: A es el tiempo promedio

```
1: T ← introsort(T) //en base al tiempo consumido
2: A ← 0
3: A' ← 0
4: para i ← 0 ... |T| hacer
5:     A' ← A' + T_i
6:     A ← A' + A
7: fin para
8: devolver A/|T|
```

4.2. b

PENDIENTE!

5. Problema 6

5.1. Diseño y descripción

Se utilizan los valores inmediatamente anteriores de primer o segundo nivel resultantes de la suma tanto para filas y columnas de forma iterativa.

6. Problema 7

6.1. Diseño y descripción

Sobre el arbol representado por el problema, se utiliza *BFS* como motor para sumar los valores desconocidos recursivamente utilizando una pila.

7. Notas respecto a los problemas 6 y 7

Todos los problemas se pueden compilar utilizando el archivo makefile provisto, el cual depende de tener instalada una versión de compilador de `G++` igual o superior a la 5, junto con `makefile`. Además, ambos fueron enviados a *UVa AutoJudge* para ser probados con la batería de pruebas oficial. Esto puede ser corroborado en mi perfil de *UHunt*: (`http://uhunt.felix-halim.net/id/715258`).