Tarea 5: Estrategia voraz y "Divide y Vencerás"

Estructura de Datos y Teoría de Algoritmos

Entrega: 5 de noviembre de 2008

1. De acuerdo con lo demostrado en clase, un código de Huffman es de longitud promedio de palabra mínima para codificar un alfabeto de símbolos dadas las frecuencias (relativas) de estos. Sin embargo en el libro de Gilbert Held[1] (pág. 109) aparece el siguiente ejemplo:

Simb.	**Prob.**	**Shannon-Fano**	**Huffman**
x_1	0.10	`001`	`101`
x_2	0.05	`0000`	`111`
x_3	0.20	`10`	`010`
x_4	0.15	`011`	`011`
x_5	0.15	`010`	`100`
x_6	0.25	`11`	`00`
x_7	0.10	`0001`	`110`

   Con estos códigos se obtienen las siguientes longitudes promedio de palabra:

$$\begin{aligned} \overline{\text{lon}}(Shannon - Fano) &= 2.7 \\ \overline{\text{lon}}(Huffman) &= 2.75 \end{aligned}$$

[1] Held, Gilbert, *Data Compression, Techniques and Applications, Hardware ans Software Considerations*, 2a. ed., John Wiley & Sons, 1987. (con software elaborado por Thomas Marshall).

El autor termina diciendo: *"Although the Shannon-Fano code is more efficient since its average code length is less than that of the Huffman code, the reader should note that it is not necessarily always more efficient."*

Esto aparentemente contradice lo demostrado. ¿Qué ocurrió? Formula tu respuesta en términos de la estrategia *voraz* que definimos para el algoritmo de Huffman.

2. Supongamos que tenemos una fuente $\mathcal{S} = (S, P)$ donde el alfabeto es $S = \{a, b\}$ y $P(a) = 0.8$, $P(b) = 0.2$. Un esquema de codificación de Huffman (H) para esta fuente sería: $f(a) = 0$, $f(b) = 1$, dado que solo hay dos símbolos. La longitud promedio de este código es, evidentemente, $\ell(H) = 1$.

   Que tal si ahora nos fijamos, no en los símbolos que produce la fuente, sino en todas las parejas de símbolos que puede producir: aa, ab, ba y bb. Sea $\mathcal{S}' = (S', P')$ la nueva fuente en la que $S' = \{aa, ab, ba, bb\}$ y cuya distribución de probabilidad es la que resulta de suponer la aparición de cada símbolo como un evento independiente del símbolo previo (i.e. $P'(aa) = P(a) \cdot P(a)$, $P'(ab) = P(a) \cdot P(b)$, etc). A esta nueva fuente se le suele llamar la *primera extensión* de la fuente original.

   Calcula la distribución de cada "símbolo" de la fuente $\mathcal{S}'$. Construye un código de Huffman para ella y calcula su longitud de palabra promedio. Calcula ahora cuál es la longitud promedio $\ell_1(H)$ usada para cada símbolo de la fuente original (Pista. Si cada símbolo de $\mathcal{S}'$ está hecho de dos símbolos de $\mathcal{S}$ entonces ...)

   ¿Qué pasará ahora considerando ternas de símbolos?, ¿y con n-adas de símbolos?, ¿qué pasa con $\ell_n(H)$ conforme n crece?

   Investiga el concepto de entropía de información y su vínculo con la codificación de Huffman. Discute tu resultado en términos de ello.

3. No lo mencionamos en clase, pero es claro que si A desea enviar datos comprimidos a B, usando la codificación de Huffman, debe anexar a los datos el modelo estadístico necesario para reconstruir el árbol de decodificación (la tabla de frecuencias). Si un archivo de datos usa un alfabeto de n símbolos ¿cuál es la complejidad de espacio que se requiere para almacenar el modelo? ¿cómo crece esta complejidad si se codifican extensiones de la fuente?

4. Dadas dos matrices $\mathbf{A}$ y $\mathbf{B}$, de $n \times n$ entradas. La matriz producto:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

está constituida por entradas que pueden calcularse como:

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k}\, b_{k,j}$$

¿Cuál es la complejidad del algoritmo (pedestre) que resultaría de aplicar esta formula directamente, en términos del número de operaciones aritméticas necesarias?

Podemos pensar cada matriz de $n \times n$ como constituida por cuatro sub-matrices de $n/2 \times n/2$, digamos:

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

Definimos:

$$\begin{aligned} M_0 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\ M_1 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2}) \\ M_2 &= (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2}) \\ M_3 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\ M_4 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\ M_5 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\ M_6 &= (A_{2,1} + A_{2,2}) \times B_{1,1} \end{aligned} \tag{1}$$

De esto resulta que:

$$\begin{aligned} C_{1,1} &= M_0 + M_1 - M_3 + M_5 \\ C_{1,2} &= M_3 + M_4 \\ C_{2,1} &= M_5 + M_6 \\ C_{2,2} &= M_0 - M_2 + M_4 - M_6 \end{aligned}$$

podemos aprovechar esto para formular un algoritmo del tipo "divide y vencerás" en el que la partición del problema consiste en fragmentar las matrices recursivamente y en los retornos, la combinación de los

resultados obtenidos consiste en hacer las sumas necesarias para calcular las C's y por tanto el producto (por cierto la suma de matrices es $O(n^2)$).

Encuentra la expresión de la recurrencia que describe la complejidad del algoritmo $T(n)$ (Pista: ¿cuantas multiplicaciones se hacen en las expresiones 1). Suponiendo que $T(1) = 1$ y aplicando el resultado visto en clase acerca de la complejidad de un algoritmo que hace q llamadas recursivas ¿de qué orden es la complejidad del algoritmo? ¿es muy diferente de la del algoritmo pedestre?

Investiga acerca del algoritmo de Strassen para multiplicar matrices y averigua qué otras cotas se han encontrado y en qué casos son aplicables. Discute justificadamente si vale o no la pena diseñar un algoritmo menos complejo (en términos de complejidad algorítmica) que el pedestre, pero mucho más complicado (en términos coloquiales).

5. Dado un vector $\mathbf{V}$ de tamaño n, con entradas reales, el problema del sub-vector de suma máxima consiste en encontrar el tramo contiguo de $\mathbf{V}$ tal que la suma de las entradas sea máxima. El siguiente algoritmo encuentra el valor de dicha suma justamente.

```
MaxSum3(v, n)
 1  return SumParticion(v, 0, n − 1)
 2  end

SumParticion(v, inf, sup)
 1  if inf > sup then
 2      return 0
 3  endif
 4  if inf = sup then
 5      return Maximo{0, v[inf]}
 6  endif
 7  m ← (inf + sup)/2
 8  sumactual ← 0
 9  izqmax ← 0
10  for i ∈ {m, ..., inf} do
11      sumactual ← sumactual + v[i]
12      if sumactual > izqmax then
13          izqmax ← sumactual
14      endif
15  endfor
16  sumactual ← 0
17  dermax ← 0
18  for i ∈ {m + 1, ..., sup} do
19      sumactual ← sumactual + v[i]
20      if sumactual > dermax then
21          dermax ← sumactual
22      endif
23  endfor
24  sumainterseccion ← izqmax + dermax
25  maxpartizq ← SumParticion(v, inf, m)
26  maxpartder ← SumParticion(v, m + 1, sup)
27  return Maximo{sumainterseccion, maxpartizq, maxpartder}
28  end
```

¿Cual es su complejidad en términos dle número de sumas que se hacen?