Examen de Teoría de la Programación

E. U. ING. TEC. EN INFORMÁTICA DE OVIEDO

Final septiembre – Curso 2003-2004 10 de septiembre de 2004

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

Soluciones

(A continuación se proporciona una idea de las soluciones a las preguntas propuestas. Hay que tener en cuenta que las respuestas no son exhaustivas, se pretende más bien de orientar sobre la respuesta que se pedía. Por otra parte, a cada ejercicio se proporciona una respuesta correcta que normalmente no es la única, por tanto, el que tu respuesta no coincida con esta no quiere decir que el ejercicio este mal)

3.

a) ¿Cuál es la complejidad del algoritmo de Strassen? (justifica la respuesta)

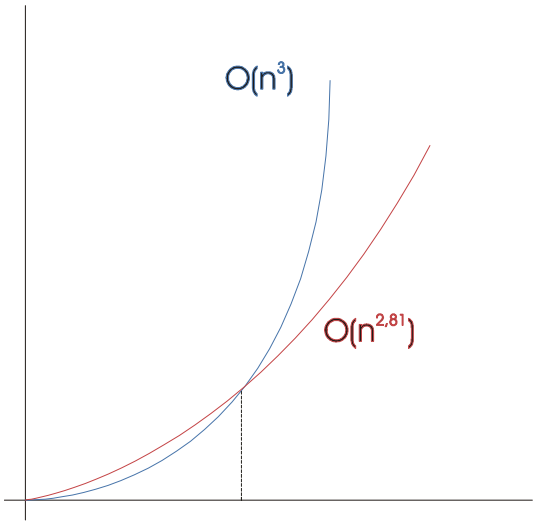
Ecuación recurrente:
$T(n)= 7 \cdot T(n/2) + O(n2)$

Complejidad temporal:
Podemos recurrir a las tablas que nos permiten calcular la complejidad temporal a partir de ciertos parámetros del algoritmo.
Divide y vencerás con división
a=7 (llamadas recursivas), b=2 (constante de división), K=2 (exponente complejidad del algoritmo eliminando las llamadas recursivas) → $O(n^{\log 7})= O(n^{2,81})$

b) ¿Cuál elegirías si tuvieses que resolver una matriz de muchos elementos? ¿Y si tuviese pocos elementos? (razona la respuesta)

Complejidad Strassen: $O(n^{2,81})$
Complejidad algoritmo clásico $O(n^3)$
La expresión de la complejidad nos indica claramente que si tenemos que resolver una matriz con muchos elementos, es decir, n es muy grande, elegiríamos el algoritmo de Strassen.

Sin embargo, si hay pocos elementos, habría que estudiar las constantes multiplicativas. Por lo dicho en clase, el algoritmo clásico se comporta mejor que el de Strassen. Por lo tanto, para matrices pequeñas elegiríamos el algoritmo clásico.



4. (2 puntos) Tenemos un vector *v* de *n* números enteros positivos. Queremos resolver el siguiente problema: formar 3 subconjuntos disjuntos, a partir del vector inicial, cuyos elementos sumen el mismo valor y que este valor sea mayor que cero. Por tanto, cada elemento del vector inicial sólo puede pertenecer a uno de los subconjuntos solución o a ninguno de ellos.
Basándose en la técnica de *backtracking* (vuelta atrás), escribir, en Java, el método principal para el *backtracking*, su llamada y las declaraciones necesarias para encontrar una solución para el problema propuesto.

```
public boolean buscarSubconjuntos(int nele, boolean solEncontrada)
{
	/* Saber todos los estados accesibles desde el actual:
	 * Para cada nodo desarrollamos un hijo por cada subconjunto posible
	 * (incluyendo un subconjunto especial que significaría que el
	 * elemento no está en ningún subconjuno): i<=nSubconj
	 * Paramos cuando encontremos la primera solución.
	 */
	for (int i=0; i<=nSubconj && !solEncontrada; i++)
	{
		// Estado posible
		if (nele<n)  // debemos trabajar con los n elementos de v
		{
			// Anotar nuevo estado
			if (i<nSubconj) // sino elemento no va a ningún subconj.
			{
				solucion[nele]= i;
				suma[i]= suma[i] + v[nele];
			}
			// Comprobar si es solución
			if (!esSolucion())
			{
				// sigue buscando solución
				solEncontrada=
					buscarSubconjuntos(nele+1,solEncontrada);
				if (!solEncontrada)
					// Borrar anotación
					if (i<nSubconj)
					{
						solucion[nele]= -1;
						suma[i]= suma[i] - v[nele];

					}
			}
			else
			{
				solEncontrada= true;
			}
		}
	}
	return solEncontrada;
}
```

(En posteriores versiones daremos solución al resto de los ejercicios)