Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2014-2015

Control 1 – 9 de marzo de 2015

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1,5 p.)

Responde a las siguientes preguntas:

a) Si la complejidad de un algoritmo es O(nlogn), y dicho algoritmo toma 5 segundos para n=4, calcula el tiempo que tardará para n=16.

n1 = 4 – t1= 5 segundps

n2 = 16 – t2 = ?

n1 * k = n2 => k = n2/n1 = 16/4 = 4

t2 = (n2 * logn2) / (n1 * logn1) * t1 = k * (logn2 / logn1) * t1 = 4 * (log16/log4) * 5 = 4 * 4 / 2 * 5 = **40 segundos = t2**

b) Considere ahora un algoritmo con complejidad *O*(n^2). Si para t = 5 segundos el método pudiera resolver un problema con un tamaño de n = 1000, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 45 segundos?

t1= 5 – n1= 1000

t2= 25 – n2= ¿?

$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{45}{5} = 9$

$f(n_2) = \frac{t_2}{t_1} \cdot f(n_1) = K \cdot f(n_1) \Rightarrow n_2 = f^{-1}(K \cdot f(n_1))$

f(n) = n^c → $n_2 = \sqrt[c]{K} \cdot n_1 = \sqrt[2]{9} \cdot 1000 = 3000$

Pregunta 2 (1,5 p.)

Indica la complejidad temporal de los siguientes fragmentos de código:

a) Divide & Vencerás por substracción con a = 3, b = 1 y k = 2. Como a > 1 entonces la complejidad es O($a^{n/b}$) = **O(3^n)**

```
public static void test(int n) {
    for (int i=0; i<n; i++) {
        for (int j=i; j<n; j++) {
            System.out.println("Values: i:"+i+ " j:"+j);
        }
    }
    test(n-1);
    test(n-1);
    test(n-1);
}
```

b) Obviamente el cálculo de la complejidad es independiente del lenguaje de programación y del ordenador en el que se calcule. En este caso tenemos 3 bucles anidados:
   - El primero tiene una complejidad $O(\log_3 n) = O(\log n)$
   - El segundo tiene una complejidad $O(\log_4 n) = O(\log n)$
   - El tercero tiene una complejidad $O(n^2)$

   En resumen la complejidad total es **$O(n^2 \log^2 n)$**

```php
//This is a code snippet created with the PHP language
function test($n) {
    for ($i = 1; $i < $n; $i *= 3) {
        for ($j = n; $j >= 0; $j /= 4) {
            for ($k = 10; $k < $n*$n; $k++) {
                echo "i:$i j:$j k:$k"; //O(1)
            }
        }
    }
}
```

c) Divide & Vencerás por división con a = 2, b = 2 y k = 1. Como $a = b^k$ entonces la complejidad es $O(n^K \cdot \log n)$ = **$O(n \cdot \log n)$**

```
public static boolean metodo5 (int n)     {
     if (n<=5)
         cont++;
     else
     {
         for (int i=0; i<n; i+= 2)
             cont++ ;
         metodo5(n/2);
         metodo5(n/2);
     }
     return true;
 }
```

Pregunta 3 (1 p.)

Teniendo en cuenta el algoritmo de ordenación Quicksort, responde a las siguientes preguntas:

a) Indica la complejidad del algoritmo en los casos mejor, medio y peor cuando se elige un buen pivote como elemento a particionar y el número de elementos es grande
   En el caso mejor y medio es O(nlogn) y en el caso peor es $O(n^2)$ o también O(nlogn)
b) Indica la complejidad del algoritmo en los casos mejor, medio y peor cuando se elige un mal pivote como elemento a particionar y el número de elementos es grande
   La complejidad será $O(n^2)$ en todos los casos, también es válido consierar O(nlogn) en el caso mejor.
c) Si la lista de elementos que queremos ordenar ya viene ordenada y elegimos como pivote el último elemento de la lista en cada iteración, ¿cuál es la complejidad? ¿por

qué? ¿sería mejor o peor que la complejidad del método de Inserción directa en este caso?

Tanto si elegimos el último como el primero, estando la lista ordenada la complejidad será cuadrática porque en cada iteración, cuando se hacen las llamadas recursivas no se está creando un árbol, sino que se están creando sublistas cuyo tamaño es sólo n-1.

El método de inserción directa cuando la lista está ordenada tiene una complejidad O(n), por lo que es mejor que Quicksort en ese caso

d) Si la lista de elementos que queremos ordenar ya viene ordenada y elegimos como pivote el elemento central de la lista en cada iteración, ¿cuál es la complejidad? ¿por qué? ¿cambiaría la complejidad si en lugar del elemento central elegimos la mediana como pivote?

En este caso la complejidad será O(nlogn) porque se crea un árbol perfecto en cada iteración (el problema de descompone en dos subproblemas de idéntico tamaño, lo que supone una complejidad logarítmica).

La mediana en este caso sería exactamente lo mismo a coger el elemento central, por lo que la complejidad sería la misma.

Pregunta 4 (2 p.)

Partiendo de la siguiente secuencia de enteros `(7, 1, 4, 5, 6, 2, 3)` realizar la traza para ordenar la secuencia utilizando el algoritmo Rápido (Quicksort) con el elemento central como pivote. Indicando claramente en cada paso, cuales son el pivote (circulo) y las particiones obtenidas (subrayando cada parte del vector).

```
Nivel 0    - [7   1     4     5     6     2     3     ]
Nivel 1 iz - [3   1     4     2     ]5    7     6
Nivel 2 iz - [1   3     4     2     5     7     6
Nivel 2 de - 1    [3    4     2     ]5    7     6
Nivel 3 iz - 1    [2    3     ]4    5     7     6
Nivel 4 iz - 1    ][2   3     4     5     7     6
Nivel 4 de - 1    2     [3    ]4    5     7     6
Nivel 3 de - 1    2     3     4     ][5   7     6
Nivel 1 de - 1    2     3     4     5     [7    6     ]
Nivel 2 iz - 1    2     3     4     5     [6    ]7
Nivel 2 de - 1    2     3     4     5     6     7     ]
```

Pregunta 5 (2,5 p.)

Dado un vector de n enteros (a_1, a_2, a_3, …, a_n) obtener la suma máxima de un conjunto de elementos consecutivos. Por ejemplo, dado el vector (-2, 11, -4, 13, -5, -2) la solución sería 20, desde a_2, hasta a_4.

Resolver el problema mediante la técnica DV:

a) Escribir el código del método principal recursivo
b) Escribir el código del método que combina las soluciones parciales.
c) Qué complejidad el algoritmo diseñado

Solución:

```
public class SumaMaxima
{

	static int []v;

	/** Algoritmo es recursivo
		DyV POR DIVISIÓN con a=2;b=2;k=1 === O(nlogn)
	 */
	public static int sumamax3 (int[]v)
	{
		return sumarec (0,v.length-1);
	}

	private static int sumarec (int iz,int de)
	{
	private static int sumarec (int iz,int de)
	{
		if (iz==de)
			return v[iz];
		else
		{
			int m=(iz+de)/2;
			int maxiz=sumarec(iz,m);
			int maxde=sumarec(m+1,de);
			int maxCent= maxDesdeCentro(iz,de,m);

			return mayor (maxiz,maxde,maxCent);
		}
	}
...
}
```

b)

```
	private static int maxDesdeCentro(int iz, int de, int m)
	{
		// calcular el máximo de la subsecuencia que pasa
		// de una mitad a la otra mitad
		int s1=0;int maxs1=0;
		for(int i=m;i>=iz;i--)
		{
			s1=s1+v[i];
			if (s1>maxs1) maxs1=s1;
		}
		int s2=0;int maxs2=0;
		for(int i=m+1;i<=de;i++)
		{
			s2=s2+v[i];
			if (s2>maxs2) maxs2=s2;
		}
		return maxs1+maxs2;
	}
```

c)

Complejidad O(nlog n), ya que es DV con división a= 2, b= 2, K= 1.

Pregunta 6 (1,5 p.)

Convertir la implementación del algoritmo ordenación por mezcla o mergesort DV recursiva a una implementación que utilice hilos para ejecutar cada una de las llamadas de forma paralela.

```
public class MezclaParalelo
{
	static int []v;

	public static void mezcla(int[]v)
	{
		mezclarec (v,0,v.length-1);
	}

	private static void mezclarec(int[] v, int iz,int de)
	{
		if (de>iz)
		{
			int m=(iz+de)/2;
			mezclarec(v,iz,m);
			mezclarec(v,m+1,de);
			combina(v,iz,m,m+1,de);
		}
	}

	public static void main (String arg [] )
	{
		int n=10;
		v = generarVector(n);
		mezcla(v);
		imprimirVector(v);
	}
...
}
```

Solución:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class MezclaParalelo extends RecursiveAction
{
	int []v;	/** eliminamos static */
	int iz, de; /** variables de instancia para los índices */

	@Override
	protected void compute() {
		if (de>iz)
		{
			int m=(iz+de)/2;

			/** creamos instancias con las partes */
			MezclaParalelo mezIz= new MezclaParalelo(v,iz,m);
			MezclaParalelo mezDe= new MezclaParalelo(v,m+1,de);
			/** invocación en paralelo */
			invokeAll(mezIz,mezDe);

			combina(v,iz,m,m+1,de);
```

```
            }
        }

        /** constructor al que pasamos vector e índices */
        public MezclaParalelo(int[] vIni, int izIni, int deIni)
        {
            v= vIni;
            iz= izIni;
            de= deIni;
        }

        public static void main (String arg [] )
        {
            int n=10;
            /** declaramos y creamos v */
            int v[] = generarVector(n);
            imprimirVector(v);

            /** Creamos instancia inicial, pool e invocamos */
            MezclaParalelo mez= new MezclaParalelo(v,0,v.length-1);
            ForkJoinPool pool= new ForkJoinPool();
            pool.invoke(mez);

            imprimirVector(v);
        }
...
}
```