Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2015-2016

Control 1 – 29 de febrero de 2016

Apellidos, nombre ______________________________ NIF: ____________

Pregunta 1 (1 p.)

Responde a las siguientes preguntas:

a) (0,5p.) Si la complejidad de un algoritmo es $O(3^n)$, y dicho algoritmo toma 10 segundos para n=10, calcula el tiempo que tardará para n=14.

n1 = 10 – t1 = 10 segundos
n2 = 14 – t2 = ?

*t2 = (f(n2)/f(n1))*t1=* $(3^{n2} / 3^{n1})$ * t1 = 3^{n2-n1} * t1 = 3^{14-10} * 10 = 81 * 10 = **810 segundos**

b) (0,5p.) Considere ahora un algoritmo con complejidad $O(\log_2 n)$. Si para t = 5 segundos el método pudiera resolver un problema con un tamaño de n = 2, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 50 segundos?

t1= 5 – n1 = 2
t2= 50 – n2 = ?

$$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{50}{5} = 10$$

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

Mucho cuidado con: $f^{-1}\left(K \cdot f(n_1)\right) \neq f^{-1}(K) \cdot n_1$

$\log n2 = \mathrm{k} \cdot \log n1 => \log n2 = 10 \cdot \log_2 2 => 2^{\log n2} = 2^{10 \cdot 1} =>$

$\mathbf{n2 = 1024}$

Pregunta 2 (2 p.)

Indica la complejidad temporal de los siguientes fragmentos de código:

a)

- El primer bucle tiene una complejidad de O(n)
- El segundo bucle tiene una complejidad de $O(\log_4 n^3) = O(\log n)$
- El tercer buble tiene una complejidad de O(n)

Como están anidados se multiplican y por tanto, la complejidad final es **$O(n^2 \log n)$**

El tercer bucle se podría ver como $O(n^3)$ por tanto la complejidad total sería $O(n^4 \log n)$

```
public void method1(int n) {
        int t = 200;
        for (int i = 2*n; i>=0; i -= 3) {
            for (int j = i; j <= n*n*n; j*=4) {
                System.out.println("Hello");
                t++;
                for (int k=0; k<j; k += 2) {
                    System.out.println(t);
                }
            }
        }
    }
```

b) D&V por división con a = 1, b = 2 y k = 2. Como a < b^k entonces la complejidad será $O(n^k)$ = **$O(n^2)$**

```
public void method2(int n, int p) {
    if (n < 0)
        System.out.println("Bye");
    else {
        int sum = 0;
        method2(n/2, p);
        for (int i = 0; i<n; i++) {
            for (int j = 0; j<n; j++) {
                sum++;
            }
        }
        System.out.println("sum:" + sum);
    }
}
```

c) D&V por división. La llamada recursiva se encuentra dentro del bucle. Por tanto en el primer nivel se realizarán n llamadas recursivas, en el segundo n/2 y así sucesivamente. Esto no encaja exactamente con nuestras tablas ya que necesitaríamos una *a* constante. Sin embargo vamos a hacer una aproximación con a = n, b = 2 y k = 0. Como a > b^k entonces la complejidad será $O(n^{\log_b a})$ = **$O(n^{\log_2 n})$** en el caso general

```
public void method3(int n) {
    for (int i=0; i<=n; i++) {
        method3(n/2);
    }
}
```

d) Este es el algoritmo de Fibonacci. D&V por substracción. No podemos utilizar las tablas directamente ya que *b* no es constante, en una llamada b=1 y en otra b=2. Planteamos dos funciones hipotéticas con b constante que nos marcarán los límites superior e inferior de la complejidad:
a = 2, b = 1 y k = 0 => Como a > 1 entonces la complejidad será $O(a^{n \text{ div } b})$ = **$O(2^n)$**
a = 2, b = 2 y k = 0 => Como a > 1 entonces la complejidad será $O(a^{n \text{ div } b})$ = **$O(2^{n/2})$**
Así: **$O(2^{n/2})$ <= O(method4) <= $O(2^n)$**

```
public int method4(int n) {
    if (n <= 0)
        return 0;
```

```
        else if (n==1)
              return 1;
        else return method4(n-1) + method4(n-2);

}
```

Pregunta 3 (3 p.)

Considerando la siguiente secuencia de números: 3, 5, 1, 6, 9, 2, 7, 8, 4, ordénalos utilizando los métodos indicados a continuación e indica claramente los movimientos de números que realizas paso a paso:

a) (1p.) Inserción directa:

Solución:

En esta técnica vamos tratando uno a uno los elementos del vector por orden de aparición y colocándolos en la parte izquierda de forma ordenada. Se marcan en verde los elementos ya tratados.

3	5	1	6	9	2	7	8	4
3	5	1	6	9	2	7	8	4
1	3	5	6	9	2	7	8	4
1	3	5	6	9	2	7	8	4
1	3	5	6	9	2	7	8	4
1	2	3	5	6	9	7	8	4
1	2	3	5	6	7	9	8	4
1	2	3	5	6	7	8	9	4
1	2	3	4	5	6	7	8	9

b) Rápido utilizando como pivote la mediana a tres o pivote central:

Solución: Pivote central y mover al principio.

	3	5	1	6	9	2	7	8	4	
Izq: Nivel 1	4	5	1	6	3	2	7	8	]9	iz= 0, de= 7
Izq: Nivel 2	[2	5	1	4	3	]6	7	8	9	iz= 0, de= 4
Izq: Nivel 3										
Der: Nivel 3	1	[5	2	4	3	]6	7	8	9	iz= 1, de= 4
Izq: Nivel 4										
Der: Nivel 4	1	2	[5	4	3	]6	7	8	9	iz= 2, de= 4
Izq: Nivel 5										
Der: Nivel 5	1	2	3	4	[5	]6	7	8	9	iz= 4, de= 4
Der: Nivel 2	1	2	3	4	5	6	[7	8	]9	iz= 6, de= 7
Izq: Nivel 3										
Der: Nivel 3										iz= 7, de= 7
Der: Nivel 1	1	2	3	4	5	6	7	8	9	iz= 9, de= 8

	Pivote
	Partición izquierda
	Partición derecha

c) ¿Cuáles son las complejidades temporales de los métodos anteriores en los casos mejor y peor? ¿cuándo se dan?
Inserción: caso mejor: O(n) → vector ordenado, caso peor: $O(n^2)$→ vector aleatorio
Rápido: caso mejor: O(nlogn) → pivote es la mediana de la partición tratada, caso peor: $O(n^2)$ → pivote es el mínimo o máximo de la partición tratada.

Pregunta 4 (2 p.)

En la paralelización de un algoritmo DV conseguimos que la ejecución secuencial de las llamadas recursivas se realice ahora en distintos núcleos / procesadores.

a) (0,5 p.) Indica si se ganaría tiempo si paralelizamos la búsqueda binaria (justifica la respuesta).

No se ganaría tiempo, debido a que sólo se realiza una llamada recursiva en cada nivel que no se podría dividir en dos procesadores.

b) (1,5 p.) Convertir la implementación del algoritmo ordenación quicksort DV recursivo a una implementación que utilice hilos para ejecutar cada una de las llamadas de forma paralela.

```
public class Rapido
{
    static int []v;     // vector sobre el que trabajamos

    public static void main (String arg [] )
    {
        int n= 10000;

        v = new int [n];

        Vector.aleatorio (v);
        Vector.mostrar (v); // antes de ordenar

        rapido(v,0,n-1);

        Vector.mostrar (v); // ordenado

    } // fin de main

    public static void quicksort (int[] v, int iz, int de)
    {
        int m;
        if (de>iz)
        {
            m=particion(v,iz,de);
            quicksort(v,iz,m-1);
            quicksort(v,m+1,de);
        }
    }


}
```

Solution:

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

```
public class RapidoParalelo extends RecursiveAction
{
    // guarda el vector y los índices del rango con el que trabajamos
    static int[] v;      // vector sobre el que trabajamos
    private int izq, der;

    public RapidoParalelo(int[] array, int izq, int der)
    {
        this.v= array;
        this.izq= izq;
        this.der= der;
    }

    public static void main (String arg [] )
    {
        int n= 10000;
        v = new int [n];

        Vector.aleatorio (v);
        Vector.mostrar (v); // antes de ordenar

        int threadCount= 2;
        ForkJoinPool pool= new ForkJoinPool(threadCount);
        RapidoParalelo rapido= new RapidoParalelo(v,0,n-1);
        pool.invoke(rapido);

        Vector.mostrar (v); // ordenado
    } // fin de main

    @Override
    protected void compute() {
        int m;
        if (der>izq)
        {
            m=particion(v,izq,der);

            // Creamos tantas instancias como llamada recursivas
            // y pasamos los mismo par□tros
            RapidoParalelo rapido1= new RapidoParalelo(v,izq,m-1);
            RapidoParalelo rapido2= new RapidoParalelo(v,m+1,der);

            // Lanza cada instancia en un hilo distinto
            invokeAll(rapido1,rapido2);

            // No hay que hacer nada más en la fusión
        }

    }
}
```

Pregunta 5 (2 p.)

Queremos diseñar un algoritmo de búsqueda “ternaria”. Partiendo de un vector ordenado, primero compara con el elemento en posición n/3 del vector, si éste es menor que el elemento x a buscar entonces compara con el elemento en posición 2n/3, y si no coincide con x busca recursivamente en el correspondiente subvector de tamaño 1/3 del original. Este algoritmo devolverá la posición del elemento x buscado y -1 si no existe.

a) (0,5 p.) ¿Cuál es la altura máxima del árbol de llamadas?

$\log_3 n$, en cada nivel dividimos el tamaño del problema entre 3.

b) (0,5 p.) ¿Conseguimos así un algoritmo más eficiente que el de búsqueda binaria?

Para medir la eficiencia utilizamos la complejidad. En este caso a= 1, b=3, K=0 → $O(n^K \log n)$ → O(log n). Tiene la misma complejidad que la búsqueda binaria, es igual de eficiente; pero no más.

c) Escribir el código Java del método que implemente este algoritmo.

Solution:

```
public class BusqTernaria
{
    static int []v;
    static int x;

    public static int busqternaria (int[]v,int x)
    {
        return busqternaria(0,v.length-1,x);
    }

    private static int busqternaria(int iz,int de, int x)
    {
        if (iz>de) return -1;  // no existe x
        else
        {
            int mter=(de-iz+1)/3;
            if (v[iz+mter]==x) return iz+mter;
            else if (v[iz+mter]>x)
                // buscar 1er tercio
                return busqternaria(iz,iz+mter-1,x);
            else if (v[de-mter]==x) return de-mter;
            else if (v[de-mter]>x)
                // buscar 2o tercio
                return busqternaria(iz+mter+1,de-mter-1,x);
            else
                // buscar 3er tercio
                return busqternaria(de-mter+1,de,x);
        }
    }

    // PROBAR FUNCIONAMIENTO
    public static void main (String arg [] )
    {
        int n=10;
        v=new int[n];
        for (int i=0;i<n;i++) v[i]=i;
        x=3;
```

```
        System.out.println (busqternaria(v,x));
        x=18;
        System.out.println (busqternaria(v,x));
    }
}
```