Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2016-2017

Control 1 – 6 de febrero de 2017

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1 p.)

Responde a las siguientes preguntas:

a) (0,5p.) Consideremos el algoritmo Quicksort con pivote central aplicado sobre vectores aleatorios. Si dicho algoritmo toma 27 microsegundos para n=512, calcula el tiempo que tardará para n=1024.

Quicksort con pivote central sobre vectores aleatorios tiene como complejidad media: O(nlog n)

$n_1 = 512 \rightarrow f(n_1)$ -- $t_1 = 27$ microsegundos

$n_2 = 1024 \rightarrow f(n_2)$ -- $t_2 = ?$

$$t_2 = \frac{f(n_2)}{f(n_1)} t_1 = \frac{n_2 \log n_2}{n_1 \log n_1} t_1 = \frac{(Kn_1)\log(Kn_1)}{n_1 \log n_1} t_1 = K \frac{\log K + \log n_1}{\log n_1} t_1$$
$$= 2 \frac{\log 2 + \log 512}{\log 512} 27 = \frac{2(1+9)}{9} \cdot 27 = 20 \cdot 3 = 60\ \mu segundos$$

b) (0,5p.) Considere ahora un algoritmo con complejidad $O(n^2)$. Si para t = 43 microsegundos el método pudiera resolver un problema con un tamaño de n = 512, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 172 microsegundos?

t_1= 43 microsegundos – $f(n_1) \rightarrow n_1 = 512$

t_2= 172 microsegundos $f(n_2)$ – $n_2 = ?$

$$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{172}{43} = 4$$

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

$$f(n_2) = K \cdot {n_1}^2 = 4 \cdot 512^2 \Rightarrow n_2 = \sqrt{4 \cdot 512^2} = \sqrt{4} \cdot 512 = 1024$$

Pregunta 2 (2 p.)

Indica la complejidad temporal de los siguientes fragmentos de código:

a) (0,5 p.) Analizando bucle a bucle:

- 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(logn)

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

```
public void metodo1(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("Secuencia");
                t++;
                for (int k=0; k<n; k *= 2) {
                    System.out.println(t);
                }
            }
        }
    }
```

b) (0,5 p.) D&V por división con a = 2, b = 2 y k = 2. Como $a < b^k$ entonces la complejidad será $O(n^k)$ = **$O(n^2)$**

```
public void metodo2(int n, int p) {
        if (n < 0)
            sum= 1;
        else {
            sum = 0;
            metodo2(n/2, p);
            metodo2(n/2, p);
            for (int i = 0; i<n; i++) {
                for (int j = 0; j<n; j++) {
                    sum++;
                }
            }
            System.out.println("sum:" + sum);
        }
    }
```

c) (1 p.) Escribir un método recursivo en java que simule una función Divide y Vencerás por división con una complejidad $O(n^2 logn)$ y el número de subproblemas = 4.

D&V por división con a = 4 para obtener una complejidad del tipo $O(n^K \cdot \log n)$ se debe dar $a=b^k$, por tanto si a=4, entonces b = 2 y k = 2. Como toda función recursiva debe tener definida una condición de parada.

```
public void simulaDV(int n, int p) {
        if (n < 0) sum= 1;
        else {
            int sum = 0;
            simdv(n/2, p);
            simdv(n/2, p);
            simdv(n/2, p);
            simdv(n/2, p);
            for (int i = 0; i<n; i++) {
                for (int j = 0; j<n; j++) {
                    sum++;
                }
            }
        }
    }
```

Pregunta 3 (3 p.)

Por favor, responde a las siguientes preguntas:

a) (1 p.) Dada la siguiente secuencia de números: **60, 40, 20, 10, 30, 50, 70** ordénalos utilizando Quicksort con la estrategia **del elemento central** para seleccionar el pivote en cada iteración. Indica claramente la traza del algoritmo marcando el pivote escogido en cada paso y las particiones creadas.

SOLUCIÓN:

```
******************** QUICKSORT ********************
Vector a ordenar es:
     Nivel 0 - [60, 40, 20, 10, 30, 50, 70]
Pivote= 10
Izq: Nivel 1 - Vacía ----
Der: Nivel 1 - 10 [40   20    60    30    50    70]
Pivote= 60
Izq: Nivel 2 - 10 [50   20    40    30]   60    70
Pivote= 20
Izq: Nivel 3 - Vacía ----
Der: Nivel 3 - 10 20    [50   40    30]   60    70
Pivote= 40
Der: Nivel 3 - 10 20    [30]  40    [50]  60    70
Izq: Nivel 4 - Vacía ----
Der: Nivel 4 - Vacía2 ----
Der: Nivel 2 - Vacía2 ----
--> Vector ordenado es:
(10, 20, 30, 40, 50, 60, 70)
```

b) (0,5 p.) Explica la complejidad del algoritmo para los casos peor, mejor y medio e indica cuando se dan estos casos.

Casos mejor y medio: O(nlogn), caso peor: $O(n^2)$

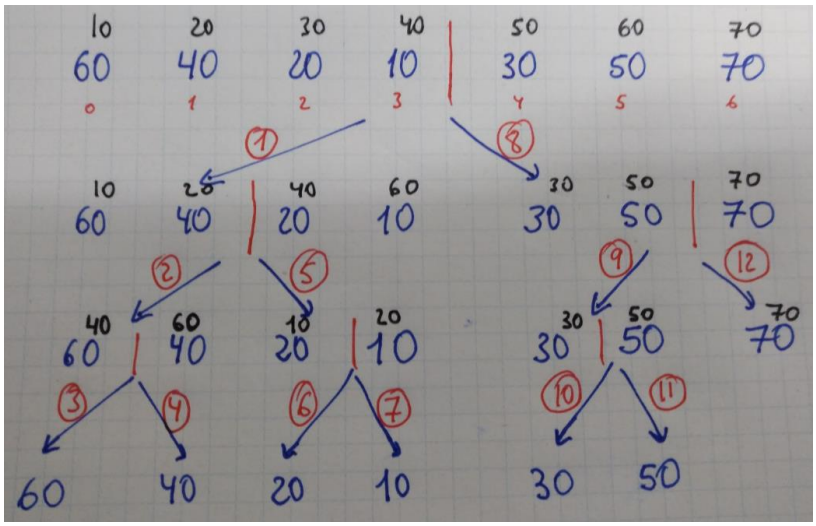
El caso mejor se da cuando el pivote que elegimos en cada nivel coincide con la mediana de los elementos y las particiones que creamos son del mismo tamaño.

El caso peor se da cuando el pivote elegido es el elemento mayor o menor de cada partición que tenemos que ordenar, es decir una de las particiones queda vacía.

El caso medio se daría cuando el pivote elegido provoca particiones desiguales, pero en ningún caso vacías.

c) (0,75 p.) Dada la siguiente secuencia de números: **60, 40, 20, 10, 30, 50, 70** ordénalos utilizando Mergesort. Indica claramente la traza del algoritmo, marcando el orden en el que se realizan las diferentes llamadas recursivas y la combinación de los diferentes subvectores para alcanzar el vector ordenado final.

SOLUCIÓN:



d) (0,25 p.) Explica cuál es la complejidad de dicho algoritmo.

La complejidad de Mergesort es O(nlogn) en todos los casos.

e) (0,5 p.) ¿Cuáles son las diferencias y similitudes entre los algoritmos de ordenación Quicksort y Mergesort?

SOLUCIÓN:

Principales similitudes:

- Ambos son algoritmos Divide y Vencerás
- Ambos tienen de media una complejidad O(nlogn)

Principales diferencias:

- Quicksort es un algoritmo interno (es decir, no necesita memoria extra para funcionar). Mergesort es un algoritmo externo (es decir, necesita memoria extra para trabajar).
- Dependiendo de la calidad del pivote, Quicksort puede llegar a tener una complejidad de $O(n^2)$.

Pregunta 4 (2 p.)

Convertir la implementación del algoritmo que suma los elementos de un vector DV recursivo a una implementación que utilice hilos para ejecutar cada una de las llamadas de forma paralela.

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

      public static int suma3 (int[]v)
```

```
	{
		return recDiv (0,v.length-1);
	}

	private static int recDiv(int iz,int de)
	{
		if (iz==de)
			return v[iz];
		else
		{
			int m=(iz+de)/2;
			return recDiv(iz,m)+recDiv(m+1,de);
		}
	}

	public static void main (String arg [] )
	{
		int n = 100;
		v=new int[n];
		for ( int i=0; i<n;i++) v[i]=i;
		System.out.println ("SOLUCION3 ="+suma3(v));
	}
}
```

SOLUCIÓN:

Se da la solución con RecursiveAction, pero también se puede realizar con RecursiveTask.

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

public class SumaVectorParalelo extends RecursiveAction
{
	private int []v;
	private int iz, de;
	int suma;

	public SumaVectorParalelo(int[] v, int iz, int de) {
		this.v= v;
		this.iz= iz;
		this.de= de;
	}

	@Override
	protected void compute() {
		if (iz==de)
			suma= v[iz];
		else
		{
			int m=(iz+de)/2;
			SumaVectorParalelo sumaVecIz=
				new SumaVectorParalelo(v,iz,m);
			SumaVectorParalelo sumaVecDe=
				new SumaVectorParalelo(v,m+1,de);
			invokeAll(sumaVecIz,sumaVecDe);
			suma= sumaVecIz.suma + sumaVecDe.suma;
		}
```

```
        }

        public static void main (String arg [] )
        {
                int n = 100;
                int[] v=new int[n];
                for ( int i=0; i<n;i++) v[i]=i;
                SumaVectorParalelo sm=
                        new SumaVectorParalelo(v,0,v.length-1);
                ForkJoinPool fj= new ForkJoinPool();
                fj.invoke(sm);
                System.out.println ("Solución3 paralela= "+sm.suma);
        }

}
```

Pregunta 5 (2 p.)

En muchas entrevistas de trabajo, las grandes empresas piden a los candidatos resolver un problema algorítmico para pasar una prueba. Un ejemplo es el siguiente:

Se parte de una colección de números ordenados, donde cada número aparece exactamente dos veces (excepto un número que aparecerá solamente en una ocasión). Diseña un algoritmo **altamente eficiente** para encontrar el valor que aparece solo.

a) (0,5 p.) Explica el algoritmo (en lenguaje natural o pseudocódigo es suficiente) y por qué éste mejora a otras soluciones.

SOLUCIÓN:

La solución sencilla es O(n), ya que bastaría con iterar a través de todo el vector (en parejas) hasta encontrar un número que no sea igual al siguiente de la lista. Esta es la solución más "obvia" pero no la mejor.

Basándonos en el algoritmo de la búsqueda binaria, podríamos hacer esta operación con una complejidad logarítmica O(logn). La diferencia entre la búsqueda binaria y este algoritmo radica en que ahora no sabemos exactamente el número que estamos buscando. Partiendo de esto tenemos que dividir el problema en cada iteración para alcanzar complejidades logarítmicas. Tenemos que fijarnos en varias cosas:

- Los números repetidos van siempre en parejas "POSICION_PAR_EN_VECTOR, POSICION_IMPAR_EN_VECTOR", empezando en las posiciones (0, 1), (2, 3) ...
- Cuando dividimos en dos el vector podemos caer en una posición par o en una posición impar. Para seguir el orden natural de la ordenación tendremos dos opciones:
    - Si la mitad del vector coincide con una posición par, debemos además seleccionar el elemento justo a su derecha
    - Si la mitad del vector coincide con una posición impar, debemos además seleccionar el elemento justo a su izquierda.
- Si el par de elementos seleccionados son iguales, eso quiere decir que desde el inicio del vector hasta la posición actual todo ha ido correctamente, por lo que el elemento sin

pareja tiene que necesariamente estar a la derecha de esa posición (dividimos en dos y **seleccionamos el vector de la derecha**).

- Si el par de elementos seleccionados no son iguales, eso quiere decir que el problema (el elemento sin pareja) tiene que necesariamente estar a la izquierda de esa posición, pues ya se ha descompuesto el orden natural de los elementos (dividimos en dos y **seleccionamos el vector de la izquierda**, teniendo en cuenta que el primer elemento del par de elementos que hemos seleccionado podría ser la solución final del problema.
- Dejamos de dividir el problema cuando encontramos un elemento aislado.

Código que implementa esta descripción (no era necesario):

```
private int buscarRec(int iz, int de) {
        if (de==iz)
                return v[iz];
        else {
                int m= (iz+de)/2;
                int pareja= m-1;  // elemento de la izquierda, suponiendo
                                  // que la mitad está en pos impar
                if (m%2==0)  // si la mitad coincide con posición par
                        pareja= m+1; // elemento de la derecha

                if (v[m]==v[pareja])  // desde inicio hasta esta posición
                                      // todos emparejados
                        return buscarRec(m+1,de); // buscar por la derecha
                else
                        return buscarRec(iz,m);
        }

}
```

b) (0,5 p.) Razona la complejidad del algoritmo.

SOLUCIÓN:

La solución sencilla es O(n), ya que consiste en iterar a través de todo el vector (en parejas) hasta encontrar un número que no sea igual al siguiente de la lista. Esta es la solución más "obvia" pero no la mejor.

La solución compleja se programa como DV con división, teniendo una llamada recursiva por cada nivel (a=1), cada vez que llamamos dividimos el vector a la mitad (b= 2) y las operaciones que realizamos para decidir por qué parte del vector vamos son de una complejidad constante (K=0). Por tanto, podríamos hacer esta operación con una complejidad logarítmica O(logn).

c) (1 p.) Aplica el algoritmo a los siguientes casos:

5

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

10 10 17 17 18 18 19 19 21 21 23

1 3 3 5 5 7 7 8 8 9 9 10 10

Si las soluciones no son claras o no están bien explicadas, se darán por incorrectas, por lo que se recomienda explicar muy claramente tanto el algoritmo como cada paso realizado.

SOLUCIÓN: