Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2022-2023

Convocatoria ordinaria – 18 de mayo de 2023 (Parte 1)

Apellidos, nombre _____ NIF: _____

Pregunta 1 (2 p.)

Responde a las siguientes preguntas (Nota: no utilices calculadora y responde con un valor numérico):

a) (1 p.) Tenemos un método con una complejidad $O(4^n)$. Si para n=5 tarda 2 segundos, ¿cuánto tarda para n=9?

t_1 = 2 seg - n_1 = 5

t_2 = ? seg - n_2 = 9

$$t_2 = \frac{f(n_2)}{f(n_1)} * t_1$$

$$t_2 = \frac{4^{n_2}}{4^{n_1}} * t_1 = \frac{4^9}{4^5} * 2 = 4^4 * 2 = 256 * 2 = \mathbf{512\ segundos}$$

b) (1 p.) Tenemos un método con una complejidad $O(4^n)$. Si para n=5 tarda 2 segundos, que tamaño del problema podemos resolver si tenemos 128 segundos en el mismo ordenador.

t_1 = 2 sec - n_1 = 5

t_2 = 128 sec - n_2 = ?

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

$$4^{n2} = 4^{n1} * 64 \Rightarrow \log_4 4^{n2} = \log_4 4^{n1} * 64 \Rightarrow n_2 = n_1 + \log_4 4^3 \Rightarrow n_2 = 5 + 3 = \mathbf{8}$$

Pregunta 2 (2 p.)

Teniendo en cuenta los siguientes métodos, indica sus complejidades y explica cómo las has calculado (aprovecha el espacio para contestarlo en esta misma hoja):

a) (0,5 p.)

```
public static void method2(int n) {
    int p = 0;
    for (int i = 2*n; i >= 0; i -= 3)
        for(int j = 1; j <= n*n*n*n; j *= 5) {
            System.out.println("Hello");
            p++;
            for (int k = 0; k < i; k++) {
                System.out.println(p);
            }
        }
}
```

O(n) * O(log n) * O(n) = **$O(n^2 \log n)$**

b) (0,5 p.)

```
public static void method1(int n) {
    if (n <= 0) System.out.println("hello");
    else {
        int p = 10;
        method1(n-2);
        method1(n-2);
        method1(n-2);
        for (int i = 0; i<=n-1; i++)
            for (int j=4; j<=n; j++) {
                p++;
                System.out.println(p);
            }
    }
}
```

Divide y vencerás por substracción a=3; b=2; k=2 ➔ $O(a^{n \text{ div } b})$ ➔ $O(3^{n/2})$

c) (0,5 p.)

```
public static void method1(int n){
    int c = 0;
    for (int i=2; i<=n/4; i++)
        for (int j=10; j<=n*n*n; j*=2)
            for (int p=4*n; p>=n/3; p-=1)
                c++;
}
```

Primer bucle: 2 - n/4 ➔ O(n).
Segundo bucle: $\log_2 n^3$ ➔ O(logn)
Tercer bucle: 4n – n/3) ➔ O(n)
➔ Total **$O(n^2 \log n)$**

d) (0,5 p.)

```
public static void method2(int n){
    if (n > 5) {
        for (int i=n; i>0; i--)
            System.out.println("hello");
        method2(n/3);
        method2(n/3);
        method2(n/3);
        method2(n/3);
        method2(n/3);
        method2(n/3);
    }
}
```

Divide y vencerás por división. a=6; b=3; k=1 ➔ **$O(n^{\log_3 6})$**

Pregunta 3 (3 p.)

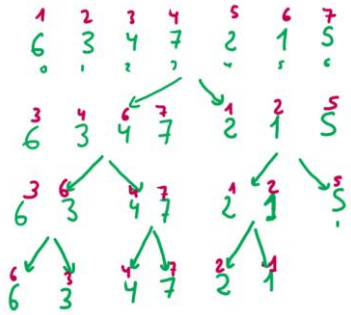
a) (1 p.) Realiza la traza del algoritmo de ordenación Quicksort (usando la mediana de 3 para seleccionar el pivote) para ordenar los siguientes números en orden ascendente 6, 3, 4, 7, 2, 1, 5. Los pasos seguidos deben verse claramente para que el ejercicio sea considerado válido: rodea con un círculo el pivote elegido en cada caso y subraya cada partición obtenida.

6	3	4	7	2	1	5	Mediana= 6
6	3	4	2	1	5	7	Escondemos el pivote al principio
5	3	4	2	1	6	7	Particiones y pivote
5	3	4	2	1			Mediana= 4
4	3	5	2	1			Escondemos el pivote al principio
4	3	2	1	5			
1	3	2	4	5			Particiones y pivote
1	3	2					Mediana= 2
2	3	1					Escondemos el pivote al principio
2	1	3					
1	2	3					Particiones y pivote

b) (1 p.) Completa la traza del algoritmo de ordenación Mergesort para ordenar los siguientes números en orden ascendente 6, 3, 4, 7, 2, 1, 5. Los pasos seguidos deben verse claramente para que el ejercicio se considere válido.

Solución:

En verde la división descendente, en rojo la composición ascendente (mezcla).



c) (1 p.) Rellena la siguiente tabla con las complejidades de los diferentes algoritmos de ordenación:

	Caso peor	**Caso medio**	**Caso mejor**
Burbuja	$O(n^2)$	$O(n^2)$	$O(n^2)$
Inserción	$O(n^2)$	$O(n^2)$	$O(n)$
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$

Quicksort (con un buen pivote)	$O(n^2)$	O(nlogn)	O(nlogn)
Mergesort (Mezcla)	O(nlogn)	O(nlogn)	O(nlogn)

Pregunta 4 (3 p.)

Partiendo del algoritmo de ordenación de Mezcla (Mergesort), resuelve las siguientes preguntas:

a) (0,5 p.) Cuál es la altura del árbol de llamadas si tenemos n elementos a ordenar.

Como es un divide y vencerás con división y cada vez dividimos a la mitad el tamaño del problema, la altura del árbol de llamadas será $\log_2 n$.

b) (1,5 p.) Considerando que tenemos una clase en Java denominada Mezcla con la siguiente forma:

```
public class Mezcla {
    private int []elementos;

    public Mezcla(int[]elementos) {
            this.elementos = elementos;
            mezcla(0, elementos.length-1);
    }

    private void mezcla(int izq, int der) {
            //TODO
    }

    /* Proceso que combina dos secuencias ordenadas
     * Usa dos vectores auxiliaries de entreda y lo deja en array elementos[] */
    private void combinar(int x1, int x2, int y1, int y2) {
            //…
    }

}
```

Sabiendo que tenemos el método **combinar()** implementado, escribe el código del método recursivo **mezcla()**, necesario para realizar la mezcla.

Solución:

Como en cualquier algoritmo divide y vencerás disponemos de condición de parada, llamadas recursivas en las que pasamos la mitad de los elementos (a través del cálculo del índice central) y la fusión de soluciones mediante el método combinar.

```
// division a=2; b=2; k=1 => O(nlogn)
private void mezcla(int izq, int der) {
        if (der > izq) {
                int centro = (izq + der)/2;
                mezcla(izq, centro);
                mezcla(centro+1, der);
                combinar(izq, centro, centro +1, der);
        }
}
```

c) (1 p.) Justifica por qué es posible paralelizar este algoritmo y si supondrá ganancia de tiempo si tenemos un procesador de varios núcleos.

Es posible paralelizar el algoritmo, porque es un DV donde divide los elementos del array en dos partes, los separa en dos conjuntos disjuntos (sin elementos en común), esto permite que al convertir cada llamada en un hilo estos trabajen de forma independiente y sin efectos laterales en el otro.

Supondrá ganancia ya que se pasará de la ejecución secuencial de las dos llamadas recursivas a la ejecución en paralelo de estas llamadas ejecutadas cada una en un hilo y permitiendo la ejecución en un núcleo diferente.

Es fundamental en esta pregunta hacer referencia al concepto de hilo que permite utilizando distintos núcleos o procesadores (si están disponibles) la ejecución en paralelo. Otra cuestión es que el framework de Java evite el trabajo directo con ellos, pero este concepto está subyacente y podemos cometer errores si no nos damos cuenta de ello.