Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2017-2018

Convocatoria ordinaria – 17 de mayo de 2018

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1 p.)

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

a) (0,5 p.) Si disponemos de un algoritmo que realiza la búsqueda binaria (o dicotómica), y dicho algoritmo toma 8 segundos para n=4, calcula el tiempo que tardará para n=16.

La complejidad temporal de la búsqueda dicotómica es O(logn)

$n_1 = 4$ – $t_1 = 8$ segundos

$n_2 = 16$ – $t_2 = ?$

$n_1 * k = n_2 \Rightarrow k = n_2/n_1 = 16/4 = 4$

$t_2 = (\log_2 n_2 / \log_2 n_1) * t_1$

$t_2 = \log_2 16 / \log_2 4 * 8$

$t_2 = 4 / 2 * 8 = 16$ segundos

b) (0,5 p.) Considere un algoritmo que itera por toda una matriz tridimensional. Si para t=2 segundos el método pudiera resolver un problema con un tamaño de n = 100, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 54 segundos?

La complejidad del algoritmo es: $O(n^3)$

t1= 2 - n1= 100

t2= 54 - n2 = ?

$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{54}{2} = 27$

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

$(n_2)^3 = 27 * (n_1)^3$

$n_2 = \sqrt[3]{27} * \sqrt[3]{100^3} = 3 * 100 = \mathbf{300\ tamaño}$

Pregunta 2 (1 p.)

Teniendo en cuenta los siguientes métodos, indica sus complejidades y explica cómo las has calculado:

a) (0,5 p.) Complejidad temporal

```
public void metodo1(int n) {
	if (n<1) return;
	metodo1(n-3);
	for (int i = 10; i <= n; i++)
		for (int j = n-1; j >= 0; j--) {
			int k = 2*i + 3*j;
```

```
                }
            }
            metodo1(n-3);
        }
```

Divide y vencerás por sustracción con a = 2; b = 3; k = 2; Como a > 1 la complejidad sigue la forma $O(a^{n/b})$ => **$O(2^{n/3})$**

b) (0,5 p.) Complejidad temporal

```
public void metodo2(int n) {
        if (n<1) return;
        for (int i = 1; i <= n; i++)
                for (int j = 1; j <= i; j++) {
                        for (int k = 1; k <= j; k++) {
                                int l += 2*i + 3*j + k/2;
                }
        }
        for (int k = 0; k < 9; k++) {
                metodo2(n/3);
        }
}
```

Divide & Vencerás por división con a = 9, b = 3 y k = 3. Como a < b^k ← $9<3^3=27$ entonces la complejidad es $O(n^k)$ = **$O(n^3)$**

Pregunta 3 (2 p.)

Necesitamos crear un método que ordene un conjunto de valores numéricos en un array de forma ascendente y sin utilizar memoria externa. Se nos pide que la **media** complejidad sea O(nlogn).

a) (0,25 p.) ¿Qué algoritmo, de los aprendidos en clase, deberíamos utilizar?

El único que vimos con esas características es el algoritmo Rápido o Quicksort.

b) (0,25 p.) ¿Qué debemos hacer para garantizar que para un tamaño de problema lo suficientemente grande, la complejidad sea la pedida y no sea superior?

Debemos tener en cuenta que hay que elegir un buen pivote para que la complejidad del algoritmo, para un tamaño suficientemente grande, no sea cuadrática. Dos buenos ejemplos serían el elemento central o la mediana a tres.

c) (1,5 p.) Ordena, utilizando el algoritmo seleccionado y teniendo en cuenta lo contestado en el apartado `b)` el siguiente conjunto de números: 2 9 3 1 5 4 7 8

Suponiendo que utilizamos mediana a tres para seleccionar el pivote, la solución sería como sigue:

Elegimos el pivote entre el primer elemento, el último y el central (siendo el central el elemento situado en la posición (ini+fin) / 2), eligiendo la mediana (elemento de valor intermedio) de los tres valores. Siempre hay que señalar el pivote escogido, crear las particiones (en clase siempre

"escondíamos" el pivote en la primera posición) y mostrar cómo quedan las particiones, antes de empezar a trabajar con cada una de ellas (aquí se muestra el pivote con fondo: granate, la partición izquierda con fondo verde y la partición derecha con fondo azul).

2	9	3	1	5	4	7	8	Pivote mediana 3, entre 2,1,8 elige 2
2	1	3	9	5	4	7	8	Intercambia un menor
1	2	3	9	5	4	7	8	Particiones creadas. Partición izquierda un solo elemento, no hay nada que hacer
		3	9	5	4	7	8	Trabajamos partición derecha, pivote elige 5
		5	9	3	4	7	8	Escondemos el pivote en la primera posición
		5	3	4	9	7	8	Colocados memores, intercambiando con otros
		4	3	5	9	7	8	Colocamos el pivote. Dos particiones
		4	3					Trabajamos partición izquierda
		3	4					Pivote mediana 3, elige 3
					9	7	8	Trabajamos partición derecha, pivote 8
					8	7	9	Escondemos pivote primera posición partición
					7	8	9	Colocamos el pivote. Dos particiones
1	2	3	4	5	7	8	9	Volvemos de las llamadas recursivas, vector ord.

Pregunta 4 (2 p.)

Dado un sistema monetario compuesto por monedas de distinto valor, el problema del cambio consiste en descomponer en monedas cualquier cantidad dada *j*, utilizando el menor número posible de monedas de dicho sistema monetario.

Siendo *n* el número de tipos de monedas distintos, *j* la cantidad que queremos descomponer y *T*(1..n) un vector con el valor de cada tipo de moneda del sistema. Suponemos que tenemos una cantidad suficiente de monedas de cada tipo. Se debe calcular la cantidad mínima final de monedas para devolver dicha cantidad *j* planteada.

Disponemos de la función *C(i,j)* que nos proporciona la solución al problema para obtener la cantidad *j* restringiéndose a los tipos $T_1, T_2, \ldots T_i$.

$$C(i,j) = \begin{cases} \infty \text{ si } (\mathrm{i}=0) \text{ ó } (\mathrm{j}<0) \\ 0 \ \textit{si} \ \ \mathrm{j}=0 \text{ y } \mathrm{i}>0 \\ Min(C(i\text{-}1,j), C(i,j\text{-}T_i)+1) \ \text{ en otro caso} \end{cases}$$

Utilizaremos la técnica de Programación Dinámica para obtener la solución óptima al problema.

a) (0,5 p.) Representar la tabla (sin rellenar) necesaria para almacenar los valores intermedios. Sabiendo que tenemos que devolver 7 céntimos con los cinco primeros tipos de monedas del sistema euro. Marcar en la tabla los valores podemos rellenar de forma directa.

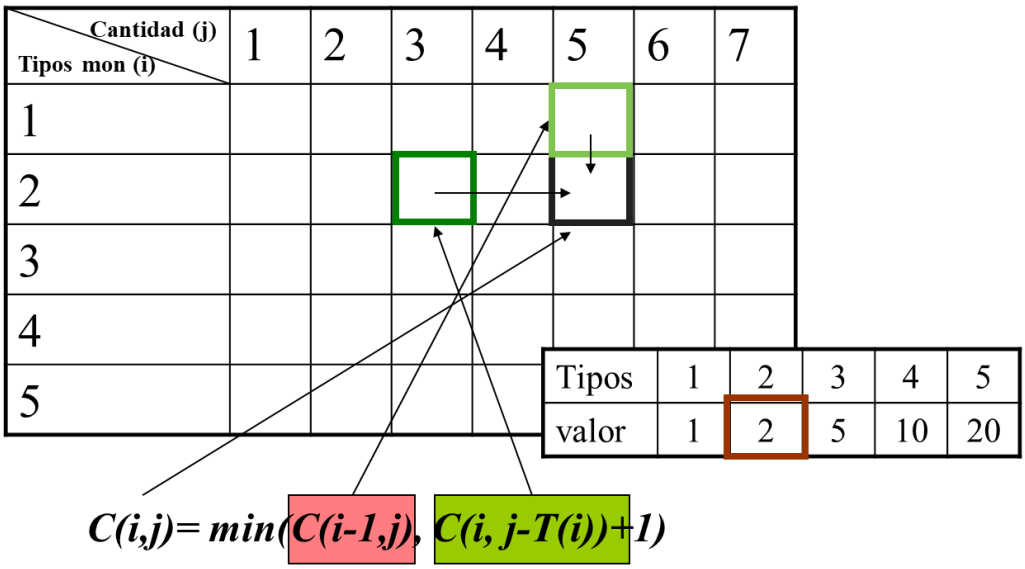
Para devolver 7 céntimos con los 5 primeros tipos de monedas. Cantidad → j, Tipos monedas → n

Tipos mon (i) \ Cantidad (j)	1	2	3	4	5	6	7
1 (1 cent)							
2 (2 cent)							
3 (5 cent)							
4 (10 cent)							
5 (20 cent)							

Se podría incluir en la tabla la fila 0 y la columna 0.

Hemos eliminado de la tabla los valores que se podían rellenar de forma directa, por tanto, todas las celdas deben de ser calculadas. Si se crean la fila y columna 0, habría que rellenarlas con ∞ y 0 respectivamente.

b) (0,5 p.) Indicar el patrón de dependencia de una celda cualquiera, es decir, marcar las celdas que son necesarias para calcular una celda dada. Además, explicar de forma específica cómo rellenaríamos la celda (1,1).



Para rellenar la celda (1,1) tendríamos un patrón de dependencia que se saldría de la tabla, dependería de la celda (0,1) y de la celda (1,0), debemos tener programado esta posibilidad donde los valores de las dos celdas virtuales serían 0 y por tanto el resultado final es 1.

c) (1 p.) Si hablamos de eficiencia en tiempo de ejecución qué técnica deberíamos escoger: programación dinámica o un algoritmo voraz. Razona la respuesta planteando las complejidades medias.

Preguntan por eficiencia en tiempo de ejecución, no confundir con la posibilidad de que el algoritmo devuelva la solución óptima en todos los casos.

Programación dinámica: Dos bucles anidados que van recorriendo cada una de las celdas del array. O(i·j)→ $O(n^2)$

Algoritmo voraz, partiendo de los tipos de monedas ordenados O(n). Si no están ordenados O(nlogn). Vease el código a continuación:

```
public void calcularCambio(int[] monedas, int cantidad)
{
    int tipoMon= 0;
    while (cantidad>0 && tipoMon<monedas.length)
    {
        // Estado es valido
        if (cantidad-monedas[tipoMon]>=0)
        {
            // nueva moneda en el conjunto solución
            solucion[tipoMon]++;
            cantidad-= monedas[tipoMon];
        }
        else
        // pasa a comprobar la moneda de valor inferior
            tipoMon++;
    }
}
```

(El heurístico nunca tiene influencia sobre la eficiencia del algoritmo)

Es mejor voraz en los dos casos. Además, para este caso el heurístico sería óptimo.

Pregunta 5 (2 p.)

Se pide calcular el camino simple de menor coste entre un nodo origen y un nodo destino (origen <> destino) en un grafo dirigido de pesos positivos.

a) (1,5 puntos) El problema lo hemos resuelto utilizando la técnica de Backtracking. Completar el código Java para dar solución a este problema (escribirlo en los recuadros preparados a tal efecto).
Los pesos de las aristas vienen especificados en la matriz *w* y la ausencia de arista estará marcada como -1 en esta matriz.

```
public class CaminoMejorPoda
{
    static int n;       // número de nodos
    static int[][]w;    // matriz de pesos de aristas
    static int origen;
    static int destino;
```

```
    static boolean [] marca;// nodos ya visitados
    static int[] camino;  // secuencia de nodos del camino
    static int coste;     // coste acumulado del camino
    static int longitud; // longitud de camino
    static int nsol;      // número de caminos hallados

    static int []caminoMejor; // camino mejor
    static int costeMejor;    // coste del camino mejor
    static int longMejor;     // numero aristas del camino mejor

    static void backtracking (int actual)  {
        if (actual==destino)  {
            nsol++;
            if (coste<costeMejor) {
                for (int l=0;l<=longitud;l++)
                    caminoMejor[l]=camino[l];
                costeMejor=coste;
                longMejor=longitud;
            }
        }
        else
            for (int j=0;j<n;j++)
→               if (!marca[j] && w[actual][j]!=-1)
                    longitud++;
                    coste=coste+w[actual][j];
                    marca[j]=true;
                    camino[longitud]=j;
                    backtracking(j);
                    longitud--;
                    coste=coste-w[actual][j];
                    marca[j]=false;
                }
    }
}
```

b) (0,5 puntos) Qué código se debe añadir y dónde (señalar con una flecha), para realizar una poda que descarte los estados que no llevan a una mejor solución.

```
if (!marca[j] && w[actual][j]!=-1
&& coste<costeMejor)
```

Pregunta 6 (2 p.)

Supongamos que tenemos 3 clases programadas en Java con las siguientes características:

```
public class ColaPrioridad { //Almacena, ordena nodos que estamos utilizando
    …
    public void insertar(Estado nodo) { … } //Inserta nodo en la cola
    public boolean esVacia() { … } //Comprueba si la cola está vacía o no
    public int estimaMejor() { … } //Estima cual es el mejor nodo de la
    cola (sin sacarlo)
    public Estado extraer() { … } //Extrae el mejor nodo de la cola
```

```
}

public abstract class Estado implements Comparable<Estado> { //Para
representar los diferentes estados de un problema
      //TODO
}

public class RamificacionYPoda { //Clase principal para solucionar problemas
      protected ColaPrioridad cola;
      protected Estado mejorSolucion;
      protected int cotaPoda;

      public BranchAndBound() { //Constructor
            cola = new ColaPrioridad();
      }

      public void realizarRamificaYPoda(Estado estadoInicial) {
            //TODO
      }
```

a) (0,75 p.) La clase abstracta Estado representa los estados del problema. Implementar los métodos de esta clase que se llaman desde el método principal realizarRamificaYPoda().

```
public abstract class Estado implements Comparable<Estado> { //Para
representar los diferentes estados de un problema
    public int valorInicialPoda();//Obtiene el valor del heurístico
    public abstract void getValorHeuristico();//Obtiene el valor del
heurístico
    public abstract ArrayList<Estado> expandir();//Crea los nodos "hijos"
    public abstract boolean esSolucion();//Indica si el estado es solución
      […]
}
```

b) (1,25 p.) Implementa el código necesario en TODO para completar el método branchAndBound() de modo que en él se implemente toda la lógica necesaria para llevar a cabo el proceso de Ramifica y Poda, desde el inicio hasta el final en el que bestNode guardará el valor de la mejor solución obtenida. Ten cuenta en cuenta todo lo que se ha explicado en clase para llevar a cabo el proceso de forma adecuada.

Solución:

```
public void realizarRamificaYPoda (Estado estadoInicial) {
    cotaPoda= estadoInicial.valorInicialPoda();

    cola.insertar(estadoInicial);   // mete estado e en la cola
    while (!cola.esVacia() && cola.estimacionMejor()<cotaPoda) {
        Estado actual= cola.extraer();
        // Examinar todos los hijos del estado actual
        for (Estado estadoHijo : actual.expandir())  {
            // Comprueba que no hay que podar
            if (estadoHijo.getHeuristico() < cotaPoda)
```

```
                if (estadoHijo.esSolucion()) {
                        mejorSolucion = estadoHijo;
                        cotaPoda= estadoHijo.getHeuristico();
                    }
                else
                    cola.insertar(estadoHijo);
            } // for
        } // while
}
```