EJERCICIOS TEMA 2

Algoritmia y Complejidad

Realizado por:

Gabriel López Cuenca
Miguel Ángel Losada Fernández
Sergio Sanz Sacristán
Álvaro Zamorano Ortega

Ejercicio 2

Se tiene que almacenar un conjunto de n ficheros en una cinta magnética (soporte de almacenamiento de recorrido secuencial), teniendo cada fichero una longitud conocida l1, l2, …, ln. Para simplificar el problema, puede suponerse que la velocidad de lectura es constante, así como la densidad de información en la cinta.

Se conoce de antemano la tasa de utilización de cada fichero almacenado, es decir, se sabe la cantidad de peticiones pi correspondiente al fichero i que se van a realizar.

Tras la petición de un fichero, al ser encontrado la cinta es automáticamente rebobinada hasta el comienzo de esta.

El objetivo es decidir el orden en que los n ficheros deben ser almacenados para que se minimice el tiempo medio de carga, creando un algoritmo voraz correcto.

- Código:

```
public static void main(String[] args) {
    int[] numeroPeticion = {0, 1, 2, 3, 4};
    int[] peticiones = {3, 2, 1, 2, 1};
    int[] longitudes = {2, 4, 3, 2, 5};
    int[] orden = ordenOptimo(longitudes, peticiones);

    System.out.println("Numero de peticion:");
    imprimirArray(numeroPeticion);

    System.out.println("Peticiones:");
    imprimirArray(peticiones);

    System.out.println("Longitudes:");
    imprimirArray(longitudes);

    System.out.println();
    System.out.print("Orden de ejecucion optimo para las peticiones: ");
    imprimirArray(orden);

    int tiempo = tiempoCarga(peticiones, longitudes, orden);
    System.out.println("Tiempo de carga siguiendo este orden: " + tiempo);
}
```

```
//Método para conseguir el orden de ejecución de las peticiones con menor
//tiempo de carga total
public static int[] ordenOptimo(int[] longitudes, int[] peticiones) {
    int n = longitudes.length;
    int posicion;
    float temporal;

    int[] orden = new int[n]; //Array para indicar el orden de las peticiones a seguir

    float[] divisiones = new float[n]; //Array de float para almacenar las divisiones
    //de peticion/longitud
    for (int i = 0; i < n; i++) {
        //Calcular divisiones y guardarlas
        divisiones[i] = (float) peticiones[i] / longitudes[i];
    }

    //Copiamos el array para no cambiar el original al ordenarlo
    //y poder comparar las posiciones finales con iniciales
    float[] divisiones_ordenado = Arrays.copyOf(divisiones, n);

    //Usamos método de ordenación por selección directa
    for (int j = 0; j < n; j++) {
        //Buscar posición del mayor objeto en la zona del array aún por
        //ordenar
        posicion = posicion_mayor(divisiones, j, n - 1);

        //Proceso para datos en las posiciones
        temporal = divisiones_ordenado[posicion];
        divisiones_ordenado[posicion] = divisiones_ordenado[j];
        divisiones_ordenado[j] = temporal;
    }
```

```
        //Comparar posiciones finales con iniciales para conocer el orden que
        //deben seguir las peticiones
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < n; k++) {
                if (divisiones_ordenado[i] == divisiones[k]) {
                    orden[i] = k;
                }
            }
        }

        //Devolver el orden de las peticiones
        return orden;
    }

    //Método complementario para obtener la posición del mayor dato dentro de
    //la zona no ordenada del array, y así poder ordenarlo
    public static int posicion_mayor(float[] vector, int pas, int fin) {
        int pos = pas;
        float mayor = vector[pas];
        for (int i = pas + 1; i < fin + 1; i++) {
            if (vector[i] > mayor) {
                mayor = vector[i];
                pos = i;
            }
        }
        return pos;
    }
```

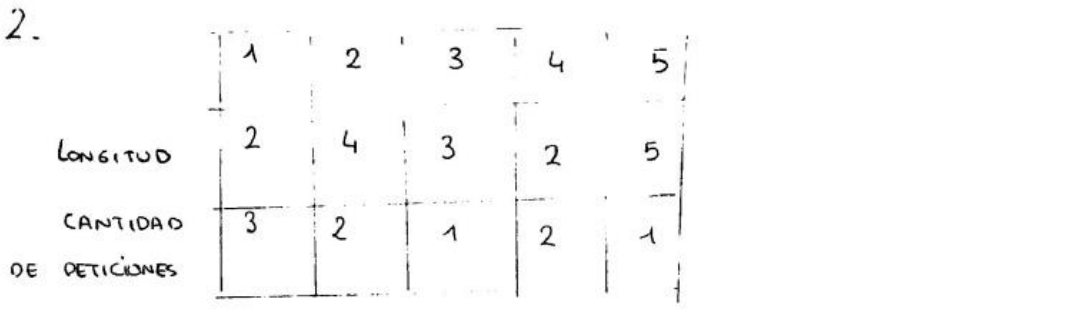
```
//Método para calcular el tiempo de carga de las peticiones en el orden óptimo
public static int tiempoCarga(int[] peticiones, int[] longitudes, int[] orden) {
    int sumatotal = 0;
    int anteriores = 0;

    for (int i = 0; i < peticiones.length; i++) {
        //Sumatorio desde la longitud de la primera petición hasta la petición actual
        anteriores += longitudes[orden[i]];
        //Se multiplica el sumatorio anterior por el valor de la petición actual
        sumatotal += anteriores * peticiones[orden[i]];
    }

    return sumatotal;
}

//Método para imprimir vectores
public static void imprimirArray(int[] vector) {
    for (int i = 0; i < vector.length; i++) {
        System.out.print(vector[i] + " ");
    }
    System.out.println();
}
```

- Resolución del problema:



Coste = $2 \cdot 3 + (2+4) \cdot 2 + (2+4+3) \cdot 1 + (2+4+3+2) \cdot 2 + (2+4+3+2+5) \cdot 1 =$

$= 65$

FÓRMULA DEL COSTE ACUMULADO DE COGER TODOS LOS FICHEROS → $T = \sum_{i=1}^{N} \left(p_{orden(i)} \cdot \sum_{j=1}^{i} l_{orden(j)} \right)$

DISTINTAS PLANIFICACIONES:

* ORDEN DE VECTOR ALEATORIO: T = 65

PRIMERO LONGITUDES MAYORES: orden → 5 2 3 1 4

$T = 5 \cdot 1 + (4+5) \cdot 2 + (4+5+3) \cdot 1 + (4+5+3+2) \cdot 3 + (4+5+3+2+2) \cdot 2 = 109$

* PRIMERO LONGITUDES MENORES: orden → 4 1 3 2 5

$T = 2 \cdot 2 + (2+2) \cdot 3 + (2+2+3) \cdot 1 + (2+2+3+4) \cdot 2 + (2+2+3+4+5) \cdot 1 =$
$= 61$

* PRIMERO MAYOR CANTIDAD PETICIONES: orden → 1 2 4 3 5

$T = 2 \cdot 3 + (2+4) \cdot 2 + (2+4+2) \cdot 2 + (2+4+2+3) \cdot 1 + (2+4+2+3+5) \cdot 1 = 61$

* PRIMERO MENOR CANTIDAD DE PETICIONES: orden → 5 3 4 2 1

$T = 5 + (5+3) + (5+3+2) \cdot 2 + (5+3+2+4) \cdot 2 + (5+3+2+4+2) \cdot 3 = 109$

* CANTIDAD PETICIONES / LONGITUD: orden → 1 4 2 3 5

$T = 2 \cdot 3 + (2+2) \cdot 2 + (2+2+4) \cdot 2 + (2+2+4+3) \cdot 1 + (2+2+4+3+5) \cdot 1 =$
$= 57$

* LONGITUD / PETICIONES: orden → 5 3 2 4 1

$T = 5 \cdot 1 + (5+3) \cdot 1 + (5+3+4) \cdot 2 + (5+3+4+2) \cdot 2 + (5+3+4+2+2) \cdot 3$
$= 113$

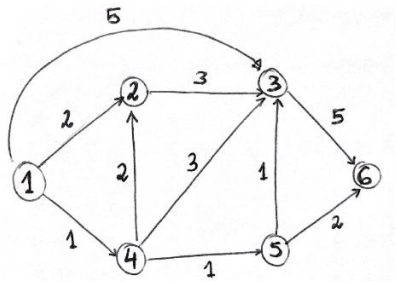
PLANIFICACION ÓPTIMA: SE OBTIENE CUANDO LOS FICHEROS VAN EN ORDEN SEGÚN LA CANTIDAD DE PETICIONES ENTRE SU LONGITUD

Ejercicio 5

Se tiene un grafo dirigido G = < N, A >, siendo N = {1, ⋯, n} el conjunto de nodos y A ⊆ NxN el conjunto de aristas. Cada arista (i, j) ∈ A tiene un coste asociado c_{ij} (c_{ij} > 0 ∀ i, j ∈ N; si (i, j) ∉ A puede considerarse c_{ij} = +∞). Sea M la matriz de costes del grafo G, es decir, M [i, j] = c_{ij}. Teniendo como datos la cantidad de nodos n y la matriz de costes M, se pide encontrar tanto el camino mínimo entre los nodos 1 y n como la longitud de dicho camino usando el algoritmo de Dijkstra, utilizando las siguientes ideas:

- **Crear una estructura de datos que almacene las distancias temporales conocidas (inicializadas al coste de la arista del vértice 1 a cada vértice j, o +∞ si no existe dicha arista) para los vértices no recorridos (inicialmente, todos salvo el 1).**
- **Seleccionar como candidato el que tenga menor distancia temporal conocida, eliminarle del conjunto de vértices no recorridos, y actualizar el resto de las distancias temporales si pueden ser mejoradas utilizando el vértice actual.**
- **Se necesitará almacenar de alguna manera la forma de recorrer el grafo desde el vértice 1 al vértice n (no necesariamente igual al conjunto de decisiones tomadas).**

- Grafo tomado como ejemplo para la resolución del ejercicio:



PASO	*SELECCIÓN*	*C*	*D[2]*	*D[3]*	*D[4]*	*D[5]*	*D[6]*
Inicial	-	{2,3,4,5,6}	2	5	**1**	∞	∞
1	4	{2,3,5,6}	**2**	4		2	∞
2	2	{3,5,6}		3		**2**	∞
3	5	{3,6}		**3**			4
4	3	{6}					**4**
5	6						

- Matriz usada para ejecutar el código:
  - Para representar ∞ escogemos un valor grande en el coste, en este caso 10000.

	1	*2*	*3*	*4*	*5*	*6*
1	0	2	5	1	10000	10000
2	10000	0	3	10000	10000	10000
3	10000	10000	0	10000	10000	5
4	10000	2	3	0	1	10000
5	10000	10000	1	10000	0	2
6	10000	10000	10000	10000	10000	0

- Código:

```
public class Tema2_EJ5 {

    //Atributos de la clase
    private static int N;
    private static final int MAX_ENTERO = 10;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int[][] matrix = generarMatriz();
        /*N = 6;
        int[][] matrix = new int[][]{{0, 2, 5, 1, 100, 100},
        {100, 0, 3, 100, 100, 100},
        {100, 100, 0, 100, 100, 5},
        {100, 2, 3, 0, 1, 100},
        {100, 100, 1, 100, 0, 2},
        {100, 100, 100, 100, 100, 0}
        };*/
        imprimirM(matrix);
        dijkstra(matrix);
    }

    public static int[][] generarMatriz() {
        //Generamos la matriz dependiendo de si el usuario elige aleatoria
        // o introducirla manualmente
        System.out.println("Generar matriz");
        System.out.print("\tNumero de nodos: ");
        Scanner sl = new Scanner(System.in);
        N = sl.nextInt();
        int[][] matriz = new int[N][N];

        System.out.print("\tIntroducir 0(manual) o 1(aleatoria): ");
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();

        if (n == 0) {
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    System.out.print("Introduzca posicion " + "[" + i + "]"
                            + "[" + j + "]: ");
                    Scanner s2 = new Scanner(System.in);
                    int m = s2.nextInt();
                    matriz[i][j] = m;
                }
            }
        } else {
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    int a = (int) Math.floor(Math.random() * 10);
                    matriz[i][j] = a;
                }
                int d = (int) Math.floor(Math.random() * (N * 2));
                for (int k = 0; k < d; k++) {
                    int b = (int) Math.floor(Math.random() * (N - 1));
                    int c = (int) Math.floor(Math.random() * (N - 1));
                    matriz[b][c] = 100;
                }
            }
        }
        System.out.println();
        return matriz;
    }
```

```java
public static void imprimirM(int[][] vector) {
    //Método para imprimir la matriz
    System.out.println("Imprimiendo matriz");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print(vector[i][j] + " ");
            System.out.print("\t");
        }
        System.out.println();
    }
    System.out.println();
}

public static void imprimirA(int[] vector, String s) {
    //Método para imprimir los diferentes array que se generan
    System.out.println("Hasta el nodo: ");
    for (int j = 0; j < N; j++) {
        System.out.print(j + 1);
        System.out.print("\t");
    }
    System.out.println();
    System.out.println(s + ": ");

    if (s.equals("Nodo anterior")) {
        for (int i = 0; i < vector.length; i++) {
            int n = vector[i];
            n++;
            System.out.print(n + " ");
            System.out.print("\t");
        }
        System.out.println("\n");
    } else {
        for (int i = 0; i < vector.length; i++) {
            System.out.print(vector[i] + " ");
            System.out.print("\t");
        }
        System.out.println("\n");
    }
}

public static void dijkstra(int[][] matriz) {
    int[] distancias = new int[N];
    int[] numAristas = new int[N];
    int[] anteriores = new int[N];
    int min = 0, imin = 0;
    int[] seleccionados = new int[N];

    for (int i = 0; i < N; i++) {
        distancias[i] = matriz[0][i];
        seleccionados[i] = 0;
        if (distancias[i] < MAX_ENTERO) {
            numAristas[i] = 1;
        } else {
            //Si distancia=infinito, numero aristas=0
            numAristas[i] = 0;
        }
    }
```

```
        for (int i = 0; i < N; i++) {
            min = MAX_ENTERO;
            //Calculo de la minima distancia temporal
            for (int j = 1; j < N; j++) {
                if (seleccionados[j] == 0) {
                    if (min > distancias[j]) {
                        min = distancias[j];
                        imin = j;
                    }
                }
            }
            //El nodo con la distancia minima temporal ya encontro su distancia
            //minima
            seleccionados[imin] = 1;
            //Actualiza las distancias del siguiente vector
            for (int j = 0; j < N; j++) {
                //Si distancia minima del nodo j no encontrada
                if (seleccionados[j] == 0) {
                    //si la distancia actual del nodo j mayor que la suma entre
                    //la distancia del nodo j y la minima distancia temporal
                    if (distancias[j] > distancias[imin] + matriz[imin][j]) {
                        //Se actualiza distancia
                        distancias[j] = distancias[imin] + matriz[imin][j];
                        //Indice anterior por el que pasa el camino mas corto
                        //temporal
                        anteriores[j] = imin;
                        //Se suma en 1 el numero de saltos para llegar al nodo
                        numAristas[j] = numAristas[imin] + 1;
                    }
                }
            }
        }

        imprimirA(distancias, "Distancias");
        imprimirA(numAristas, "Numero de aristas a recorrer");
        imprimirA(anteriores, "Nodo anterior");
        caminoFinal(numAristas, anteriores);
    }

    public static void caminoFinal(int[] aristas, int[] anteriores) {
        //Numero de aristas que recorremos para llegar al nodo final
        int n_a = aristas[aristas.length - 1];
        int[] camino = new int[n_a];
        int j = 0;
        for (int i = (anteriores.length - n_a); i < anteriores.length; i++) {
            camino[j] = anteriores[i];
            j++;
        }

        System.out.println("Para llegar al nodo " + N + " hay que pasar por "
                + "los nodos:");

        for (int i = 0; i < camino.length; i++) {
            int n = camino[i];
            n++;
            System.out.print(n + " ");
            System.out.print("\t");
        }
        System.out.println();

    }
}
```

Ejercicio 6

Shrek, Asno y Dragona llegan a los pies del altísimo castillo de Lord Farquaad para liberar a Fiona de su encierro. Como sospechaban que el puente levadizo estaría vigilado por numerosos soldados se han traído muchas escaleras, de distintas alturas, con la esperanza de que alguna de ellas les permita superar la muralla; pero ninguna escalera les sirve porque la muralla es muy alta. Shrek se da cuenta de que, si pudiese combinar todas las escaleras en una sola, conseguiría llegar exactamente a la parte de arriba y poder entrar al castillo.

Afortunadamente las escaleras son de hierro, así que con la ayuda de Dragona van a "soldarlas". Dragona puede soldar dos escaleras cualesquiera con su aliento de fuego, pero tarda en calentar los extremos tantos minutos como metros suman las escaleras a soldar. Por ejemplo, en soldar dos escaleras de 6 y 8 metros tardaría 6 + 8 = 14 minutos. Si a esta escalera se le soldase después una de 7 metros, el nuevo tiempo seria 14 + 7 = 21 minutos, por lo que habrían tardado en hacer la escalera completa un total de 14 + 21 = 35 minutos. Diseñar un algoritmo eficiente que encuentre el mejor coste y manera de soldar las escaleras para que Shrek tarde lo menos posible es escalar la muralla, indicando las estructuras de datos elegidas y su forma de uso. Se puede suponer que se dispone exactamente de las escaleras necesarias para subir a la muralla (ni sobran ni faltan), es decir, que el dato del problema es la colección de medidas de las "miniescaleras" (en la estructura de datos que se elija), y que solo se busca la forma óptima de fundir las escaleras.

- Para resolver el ejercicio lo primero que realizamos es ordenar el vector de escaleras individuales (entrada) de menor a mayor. A partir de esto, buscamos las dos escaleras de menor longitud tanto de la entrada como de la salida (suma de dos escaleras).

<u>Vector inicial:</u>

13	**18**	**4**	**21**	**7**	**24**	**2**	**17**

<u>Proceso:</u>

E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S								
1º menor: 2 2º menor: 4 Suma: 6								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6							
1º menor: 6 2º menor: 7 Suma: 13								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13						
1º menor: 13 2º menor: 13 Suma: 26								

E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13	26					
1º menor: 17 2º menor: 18 Suma: 35								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13	26	35				
1º menor: 21 2º menor: 24 Suma: 45								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13	26	35	45			
1º menor: 26 2º menor: 35 Suma: 61								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13	26	35	45	61		
1º menor: 45 2º menor: 61 Suma: 106								
E	**2**	**4**	**7**	**13**	**17**	**18**	**21**	**24**
S	6	13	26	35	45	61	106	
FIN **COSTE = 6 + 13 + 26 + 35 + 45 + 61 + 106 = 292**								

- Código:

```
public class Tema2_EJ6 {

    //Atributos de la clase
    private static int N;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //Array para almacenar los valores de entrada introducidos o generados
        ArrayList<Integer> entrada = generarArray();

        System.out.println("Imprimiendo valores");
        System.out.println(entrada.toString() + "\n");

        //Mostrar coste total
        System.out.println("COSTE TOTAL: " + algoritmoShrek(entrada));
    }
```

```java
//Método para rellenar el array de entrada con los valores ya sea
//manual o aleatoriamente
public static ArrayList<Integer> generarArray() {
    ArrayList<Integer> entrada = new ArrayList<>();

    System.out.println("Introducir valores");
    System.out.print("\tNumero de valores: ");
    Scanner s = new Scanner(System.in);
    N = s.nextInt();

    System.out.print("\tIntroducir 0(manual) o 1(aleatoria): ");
    Scanner s1 = new Scanner(System.in);
    int n = s1.nextInt();

    if (n == 0) {
        for (int i = 0; i < N; i++) {
            int k = i+1;
            System.out.print("Introduzca posicion " + "[" + k + "]: ");
            Scanner s2 = new Scanner(System.in);
            int m = s2.nextInt();
            entrada.add(m);
        }
    } else {
        for (int i = 0; i < N; i++) {
            //Introducimos valores aleatorios entre el 1 y el 30
            int m = (int) Math.floor(Math.random() * 29);
            entrada.add(m + 1);
        }
    }

    System.out.println();
    return entrada;
}
```

```java
public static int posMenor(int[] array) {
    //Seleccionamos como valor minimo el primero del array
    int min = array[0];
    //Posicion del valor mínimo
    int pos = 0;
    //Recorremos el array comparando todos los elementos con el menor
    //temporal, seleccionando el minimo valor del array y su posición
    for (int i = 0; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
            pos = i;
        }
    }
    return pos;
}
```

```java
    public static int algoritmoShrek(ArrayList<Integer> entrada) {
        int coste = 0;
        ArrayList<Integer> salida = new ArrayList<>();

        //Ordenar array de entrada
        Collections.sort(entrada);

        do {
            //Array para almacenar sumas, se inicializa con 1000 para no tenerlo
            //en cuenta en la menor suma
            int[] sumas = {1000, 1000, 1000};

            //Diferentes tipos de sumas
            if (entrada.size() >= 2) {
                sumas[0] = entrada.get(0) + entrada.get(1);
            }
            if (entrada.size() >= 1 && salida.size() >= 1) {
                sumas[1] = entrada.get(0) + salida.get(0);
            }
            if (salida.size() >= 2) {
                sumas[2] = salida.get(0) + salida.get(1);
            }

            //Buscar la posición de la menor suma
            int pos = posMenor(sumas);
            //Añadimos la suma menor al coste total
            coste += sumas[pos];

            switch (pos) {
                case 0:
                    //La suma menor es con 2 valores de entrada
                    entrada.remove(0);
                    entrada.remove(0);
                    break;
                case 1:
                    //La suma menor es con 1 valor de entrada y otro de salida
                    entrada.remove(0);
                    salida.remove(0);
                    break;
                case 2:
                    //La suma menor es con 2 valores de entrada
                    salida.remove(0);
                    salida.remove(0);
                    break;
            }
            //Añadimos la suma al array de salida para tenerla en cuenta a la
            //hora de calcular sumas menores
            salida.add(sumas[pos]);

            //Lo hacemos mientras haya datos en entrada o el array de salida
            //tenga al menos un elemento
        } while (!entrada.isEmpty() || salida.size() > 1);

        //Devolver el coste total
        return coste;
    }
}
```