PRÁCTICA 1 - EDA

Índice:

Nota: El hecho de que haya ‘1º caso’ y ‘2º caso’ es debido a que hemos encontrado dos maneras de resolver el mismo problema, y aunque son distintas maneras de diseñar el problema

1. Explicación de algoritmo DyV (1º caso)

Para ello, lo explicaremos a partir de nuestro código desarrollado (incluyendo capturas de nuestro código):

```
public static void main(String [] args) {
    cargarArchivo(ruta);

    //Hacemos el calculo de tiempos de 10 elementos para sacar un tiempo medio
    //El tiempo no no contemplamos en la carga de datos, solo en los metodos

    long tiempoInicio;
    long tiempoFinal;
    long tiempoResultante;
    long[] tiempos = new long[10];

    for (int i = 0; i < 10; i++) {
        tiempoInicio = System.currentTimeMillis();
        diezMejoresJugadores();
        tiempoFinal = System.currentTimeMillis();
        tiempoResultante = tiempoFinal - tiempoInicio;
        tiempos[i] = tiempoResultante;
    }

    double valorTiempoMedio = calcularMedia(tiempos);
    System.out.printf("\nTiempo Medio = %.4f milisegundos\n", valorTiempoMedio);

}
```

Este es el 'main' de mi programa, en el que llamo al método *cargarArchivo(File string)*, en el cual cargamos el archivo .txt en la ruta seleccionada. Si nos dirigimos ahora al método de *cargarArchivo(File string):*

```
public static void cargarArchivo(String ruta) {
    estadisticas = new ArrayList<Player>();
    File f = new File(ruta);
    String[] items = null;
    try {
        Scanner scan = new Scanner(f);
        String linea = "";
        Player ultimoJugadorCargado = null;
        String ultimoNombreCargado = "";
        while(scan.hasNextLine()) {
            linea = scan.nextLine().trim();
            if (linea.isEmpty() || linea.startsWith("#")) continue;
            items = linea.split(";");
            if (items.length != 9) continue;
            double fg = comprobarValor(items[7]);
            double pts = comprobarValor(items[8]);
            if(!items[2].equals(ultimoNombreCargado)) {
                ultimoJugadorCargado = new Player(items[2], items[6], items[4], (int) (fg*pts/100));
                estadisticas.add(ultimoJugadorCargado);
                ultimoNombreCargado = items[2];
            } else {
                ultimoJugadorCargado.add(items[6], items[4], (int) (fg*pts/100));
            }
        }
        scan.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

En dicho metodo, el objetivo principal es ir leyendo linea a linea,y para ello:
(1): Para los parámetros (columnas) que correspondan a 'fg' y 'PTS', habrá que llamar al metodo comprobarValor, el cual nos parsea el valor a un tipo primitivo 'double' y nos cambia la ',' por un '.' para que se puedan realizar las cuentas correctamente (y en el caso de que no se le pase valor, devolver 0; señalado en rojo):

```
private static double comprobarValor (String value) {
    if(value.isEmpty()) return 0;
    try {
        double d = Double.parseDouble(value.replace(",","."));
        return d;
    } catch ( NumberFormatException e ) {
        return 0;
    }
}
```

Imagen del metodo 'comprobarValor (string value)'

(2): Cómo las líneas del documento 'NbaStats.txt' están ordenadas de tal manera que, dado el caso de que un jugador haya jugado más de una temporada (independientemente de si ha cambiado de equipo, posición…) aparecerá en la siguiente fila, por lo que queremos guardar el nombre de la última fila que leemos para no repetir en el ArrayList un elemento nuevo (si el jugador ya estaba no se introduce; el nombre del jugador anterior lo guardamos en el string señalado en verde).

(3): Si el jugador ya estaba en la estructura, lo que haremos será simplemente introducir sus nuevos valores para el equipo, la posición y el score (señalado en naranja; usando el

método add que hemos implementado en la clase Player), y sino estaba el nombre del jugador, lo añadimos por primera vez (señalado en morado).

Una vez que hemos leído el contenido del archivo, siguiendo el código, se llama al método 'diezMejores()':

```
public static ArrayList<Player> diezMejoresJugadores() {
    if(estadisticas.size() == 0) {
        throw new RuntimeException("No hay datos.");
    } else {
    ArrayList<Player> arrayADevolver = diezMejoresJugadores(0,estadisticas.size()-1,estadisticas);
    System.out.println("\nLos diez mejores jugadores de toda la historia son:\n");
    //arrayADevolver.forEach((n) -> System.out.println(n));
    return arrayADevolver;
    }
}
```

Básicamente, se comprueba que los datos que se han leído correctamente (o en su defecto, no se ha leído ningún dato y no hay datos (size() == 0). En el caso de que si haya elementos, haremos una llamada al método 'diezMejores (int inicio, int fin, ArrayList<Player> auxiliar)':

```
public static ArrayList<Player> diezMejoresJugadores(int inicio, int fin, ArrayList<Player> auxiliar)
    if (inicio >= fin) {
        //System.out.println(inicio + "-" + fin);
        ArrayList<Player> auxiliar2 = new ArrayList<Player>();
        auxiliar2.add(auxiliar.get(inicio));
        //System.out.println("Auxiliar 2" + auxiliar2);
        return auxiliar2;
    }
    int mitad = (inicio + fin) / 2;
    ArrayList<Player> izq = diezMejoresJugadores(inicio,mitad,auxiliar);
    //System.out.println(izq.get(izq.size()-1));
    ArrayList<Player> der = diezMejoresJugadores(mitad+1,fin,auxiliar);
    ArrayList<Player> arr = combinarArray(izq, der);
    return arr;
}
```

Aqui ya sería el esquema general del algoritmo de DyV, en el que tenemos un caso base (señalado en naranja), diversas llamadas recursivas (señaladas en rojo; y hacemos dos en este caso porque dividiremos el array en dos partes, la izquierda y la derecha, sucesivamente), y un metodo que nos combina las soluciones, llamado 'combinarArray()' y señalado en azul). Por ultimo, tenemos el metodo 'combinarArray (ArrayList<Player> izquierda, ArrayList<Player> derecha):

```java
public static ArrayList<Player> combinarArray(ArrayList<Player> izquierda, ArrayList<Player> derecha)
    int indiceIzq = 0;
    int indiceDer = 0;

    ArrayList<Player> arrayOrdenado = new ArrayList<Player>();

    while (indiceIzq < izquierda.size() && indiceDer < derecha.size()) {

        if (izquierda.get(indiceIzq).getScore() < derecha.get(indiceDer).getScore()) {
            arrayOrdenado.add(izquierda.get(indiceIzq));
            indiceIzq++;
        }
        else {
            arrayOrdenado.add(derecha.get(indiceDer));
            indiceDer++;
        }
    }

    if (indiceIzq >= izquierda.size()) {
        while (indiceDer <= derecha.size()-1) {
            arrayOrdenado.add(derecha.get(indiceDer));
            indiceDer++;
        }
    } else if (indiceDer >= derecha.size()) {
        while (indiceIzq <= izquierda.size()-1) {
            arrayOrdenado.add(izquierda.get(indiceIzq));
            indiceIzq++;
        }
    }
```

Ahora si, en este último método antes de que los datos se nos muestren por consola, creamos un array local que será el que devolvamos ya ordenado. En el if, comprobamos que los índices (indiceIzq e indiceDer, ambos inicializados a 0), no sean mayores que el tamaño de cada parte del array (izquierda y derecha respectivamente). Se podría decir que dentro del while tratamos un caso “normal”, en el que tenemos tanto elementos como a la izquierda y derecha del array. Comprobamos que el elemento de la izquierda sea menor al de la derecha (usando para ello el método getScore(), de una clase residual llamada Player, la cual se añade ahora a continuación:

```java
package practica_1;

import java.util.ArrayList;

public class Player{

    private String playerName;
    private ArrayList<String> teams;
    private ArrayList<String> positions;
    private int score;

    public Player(String playerName, String teams, String positions, int s

        this.playerName = playerName;
        this.teams = new ArrayList<String>();
        this.teams.add(teams);
        this.positions = new ArrayList<String>();
        this.positions.add(positions);
        this.score = score;
    }

    public void add(String teams, String positions, int score) {
        if(score <= 0) return;
        this.teams.add(teams);
        this.positions.add(positions);
        this.score += score;
    }

    public String getPlayerName() {
        return playerName;
    }

    public void setPlayerName(String playerName) {
        this.playerName = playerName;
    }

    public ArrayList<String> getPositions() {
        Set<String> hashSet = new HashSet<String>(positions);
        positions.clear();
        positions.addAll(hashSet);
        return positions;
    }

    public void setPositions(ArrayList<String> positions) {
        this.positions = positions;
    }

    public ArrayList<String> getTeams() {
        Set<String> hashSet = new HashSet<String>(teams);
        teams.clear();
        teams.addAll(hashSet);
        return teams;
    }

    public void setTeams(ArrayList<String> teams) {
        this.teams = teams;
    }

    public int getScore() {
        return this.score / this.teams.size();
    }

    public void setScore(int score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "[Name = " + getPlayerName() + " | Score = " + getScore();
    }
}

```

Siguiendo con el anterior while, si el elemento de la izquierda posee menos puntuación que el elemento de la derecha, añadimos dicho elemento a nuestro arrayOrdenado e incrementamos el indiceIzq. Hacemos lo análogo en el caso de que el elemento de la izquierda fuera mayor que el de la derecha.

Respecto a los otros dos if (el primer if con un else if), es para cuando los índices (izquierda o derecha), sean mayores que los propios arrays (también izq o der). Y es básicamente para, por ejemplo, como añadir elementos en el caso de que algún array no contengan elementos, etc… Con esto contemplamos todas las casuísticas posibles y se nos quedara el array “definitivo”.

Por último, ahora ya por último, queda con un bucle for recorrer desde el size del array – 10 hasta size, sacando con ello los diez mejores jugadores. Volveríamos a repetir dicho proceso 10 veces para que el tiempo medio que obtuviéramos fuera más correcto.

2. Explicación de algoritmo DyV (2º caso)

Para este 2º caso, el esquema que se sigue es similar al anterior, pero no tenemos un metodo ‘combinarArray()’, sino que este proceso de combinacion se hace en ‘diezMejores (int inicio, int fin)’:

```
private static ArrayList<Player> diezMejores(int inicio, int fin) {
    ArrayList <Player> resultado = new ArrayList<Player>(n);
    if (inicio == fin) { resultado.add(estadisticas.get(inicio));
    } else {
        int mitad = (inicio+fin)/2;
        ArrayList<Player> parteIzq = diezMejores(inicio,mitad);
        ArrayList<Player> parteDer = diezMejores(mitad+1,fin);
        int i=0, j=0;
        while (resultado.size() < n && i <= parteIzq.size()-1 && j <= parteDer.size()-1) {
            if(parteIzq.get(i).getScore() > parteDer.get(j).getScore()) {
                resultado.add(parteIzq.get(i));
                i++;
            } else {
                resultado.add(parteDer.get(j));
                j++;
            }
        }
        while (resultado.size() < n && i <= parteIzq.size()-1) {
            resultado.add(parteIzq.get(i));
            i++;
        }
        while (resultado.size() < n && j <= parteDer.size()-1) {
            resultado.add(parteDer.get(j));
            j++;
        }
    }
    return resultado;
}
```

3. Mejora al algoritmo encontrada (1º caso)

En el primer punto nos faltó el cierre (return) del método, y esto es debido a que no se mostró una mejora, la cual nos reduce un 45% el tiempo de ejecución.Dicha mejora se trata de:

```
if (arrayOrdenado.size() >= 10) {
    ArrayList<Player> auxIzq = new ArrayList<Player>();
    for (Player s: arrayOrdenado.subList(arrayOrdenado.size()-10, arrayOrdenado.size())) {
        auxIzq.add(s);
    }
    arrayOrdenado = auxIzq;
}
return arrayOrdenado;
```

Podríamos explicarlo cómo que, al haber 10 o mas elementos (ordenados de menor a mayor) en 'arrayOrdenado', al recorrer su parte izquierda ya son elementos que "nos sobran" para este problema, ya que buscamos unicamente los 10 mejores jugadores, por lo que no debemos de seguir buscando. Cómo se comentaba anteriormente, es una mejora simple pero sin duda efectiva.

4. Ejecución con los tiempos y resultados obtenidos

Si ejecutamos nuestro programa obtenemos:

```
Los diez mejores jugadores de toda la historia son:

[Name = Jerry West* | Score = 854
[Name = Oscar Robertson* | Score = 925
[Name = Kevin Durant | Score = 935
[Name = Karl-Anthony Towns | Score = 965
[Name = Karl Malone* | Score = 1005
[Name = LeBron James | Score = 1034
[Name = George Gervin* | Score = 1059
[Name = Michael Jordan* | Score = 1075
[Name = Kareem Abdul-Jabbar* | Score = 1076
[Name = Wilt Chamberlain* | Score = 1153

Tiempo Medio = 15,7778 milisegundos
```

*El tiempo medio en todas las ejecuciones que he realizado suele oscilar entre los 5-15 milisegundos, dependiendo de la capacidad de respuesta del ordenador en dicho momento. También cabe recalcar que las ejecuciones han sido realizadas en un ordenador no demasiado potente, por lo que en uno mas potente suelen estar entre 1-2 milisegundos.

Además, para comprobar que todos estos test funcionan, hemos implementado unos juegos de prueba, en los que hemos implementado varios test:

1. Comprobar si el top10 de jugadores del método es equivalente al que queremos que salga por pantalla
2. Comprobar si el top10 de jugadores del método es equivalente al que queremos que salga por pantalla (pero usando reverse por si se requiriese de mayor de menor)

3. Comprobar que si el archivo que lee es nulo, nos muestre una excepción con el mensaje ‘No datos.’.
4. Comprobar únicamente que el mejor jugador que se obtiene es el correcto.
5. Comprobar únicamente que el peor jugador que se obtiene es el correcto

Todos estos test pueden verse (al igual que el resto del codigo), en nuestro repositorio de GutHub: https://github.com/jlp717/ual_eda2_2022