PRÁCTICA 1 DYV - EDA

Índice:

Nota: El hecho de que haya una variante 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 (mejora incluida)

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

```
private static ArrayList<Player> estadisticas;
public static int n;

    public static void main(String [] args) {
        cargarArchivo(ruta);
        long startNano = System.nanoTime();
        long startMili = System.currentTimeMillis();
        diezMejores();
        long endNano = System.nanoTime();
        long endMili = System.currentTimeMillis();
        System.out.println("\nTiempo de ejecuciónn: " + (endNano-startNano) + " nanosegundos. "
                + "|| " + (endMili-startMili) + " milisegundos.");
    }
```

Primeramente, comentar que hemos declarado dos atributos estáticos, uno llamado ‘estadísticas’, donde se van a guardar los datos del archivo ‘NbaStats.txt’, y otro llamado ‘n’, sera el nº del top de jugadores que va a sacar y nos facilitará para calcular los tiempos (si la n = 10, nos sacara el top 10 de jugadores, y si n=25, nos sacará los mejores jugadores).

Este es el ‘main’ de nuestro 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):*

```java
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 método, el objetivo principal es ir leyendo línea a línea,y para ello:
(1): Para los parámetros (columnas) que correspondan a 'fg' y 'PTS', habrá que llamar al método 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):

```java
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> diezMejores() {
    if(estadisticas.size() == 0) {
        throw new RuntimeException("No hay datos.");
    } else {
        ArrayList<Player> ganadores = diezMejores(0,estadisticas.size()-1);
        //ganadores.forEach((n) -> System.out.println(n));
        return ganadores;
    }
}
```

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)':

```
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;
}
```

Equivale a la etapa "Si Suficientemente Pequeño".

Si se cumple la cláusula anterior, procedemos a ejecutar un algoritmo específico que nos devuelve el ArrayList vacío de Players.

Equivale al proceso de "divide" dentro del algoritmo "Divide y Vencerás".

Hacemos dos llamadas recursivas al propio método para las dos mitades.

Equivale al proceso de "combinar" dentro del algoritmo.

Se ejecuta alguno de ellos únicamente si todavía quedan espacios en algún array.

Partiendo desde el bloque amarillo, creamos un bucle while que actúe como puerta lógica AND de modo que si alguna de las condiciones deja de cumplirse, el bucle acabaría. Esencialmente en dicho bucle mantenemos como requisitos el hecho de que no estén completos / llenos. Para ello, 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
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 score){

        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() + " | " + "Teams = "  + getTeams() + " | Positions = " + getPositions() + "]";
    }
```

Siguiendo con el anterior while, si el elemento de la izquierda posee mayor puntuación que el elemento de la derecha, añadimos dicho elemento a nuestro ‘resultado’ e incrementamos el indiceIzq (i). Hacemos lo análogo en el caso de que el elemento de la izquierda fuera mayor que el de la derecha. Vamos haciendo estas comprobaciones hasta que lleguemos a algún caso ‘especifico’, que son los dos siguientes while.

Respecto a estos otros dos while, es para cuando los índices (izquierda o derecha), sean mayores que los propios arrays (también izq o der), es decir, como añadir elementos en el caso de que falten elementos libres sin añadir en el array (nosotros lo hemos llamado ‘resultado’), P.e.j: En el caso de que al array resultado le falte un hueco de su array por rellenar y en el array izquierdo haya un elemento restante que no ha sido añadido, se añade automáticamente a esa hipotética posición final. Lo mismo pasaría con el caso de la parte derecha.

La primera sentencia de cada while se corresponde con una mejora ya inducida en el método, que será el if “suelto” que este al final de la siguiente variante.

2. Posible variante

Para este 2º caso, el esquema que se sigue es similar al anterior, pero tenemos un método 'combinarArray()', el cual nos une la solución (tanto de la parte izquierda cómo de la parte derecha) en un array al que llamamos 'arrayOrdenado':

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

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

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

    return arrayOrdenado;
}
```

Aclarar además, que el último if (if arrayOrdenado >= n…), se trata de una mejora significativa al método, ya que reduce en torno a un 45% del tiempo de ejecución. En el caso de la mejora podríamos explicarla cómo que, al haber 10 o más elementos (para el caso del ejercicio; 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.

3. Resultados obtenidos

Si ejecutamos nuestro programa obtenemos:

```
[Name = Wilt Chamberlain* | Score = 1153 | Teams = [SFW, PHI, LAL, TOT, PHW] | Positions = [C]]
[Name = Kareem Abdul-Jabbar* | Score = 1076 | Teams = [MIL, LAL] | Positions = [C]]
[Name = Michael Jordan* | Score = 1075 | Teams = [CHI, WAS] | Positions = [SF, SG]]
[Name = George Gervin* | Score = 1059 | Teams = [SAS, CHI] | Positions = [SF, SG]]
[Name = LeBron James | Score = 1034 | Teams = [MIA, CLE] | Positions = [SF, SG, PF]]
[Name = Karl Malone* | Score = 1005 | Teams = [UTA, LAL] | Positions = [PF]]
[Name = Karl-Anthony Towns | Score = 965 | Teams = [MIN] | Positions = [C]]
[Name = Kevin Durant | Score = 935 | Teams = [OKC, GSW, SEA] | Positions = [SF, SG]]
[Name = Oscar Robertson* | Score = 925 | Teams = [CIN, MIL] | Positions = [PG]]
[Name = Jerry West* | Score = 854 | Teams = [LAL] | Positions = [SG, PG]]
Tiempo de ejecución: 3470100 nanosegundos. || 3 milisegundos.
```

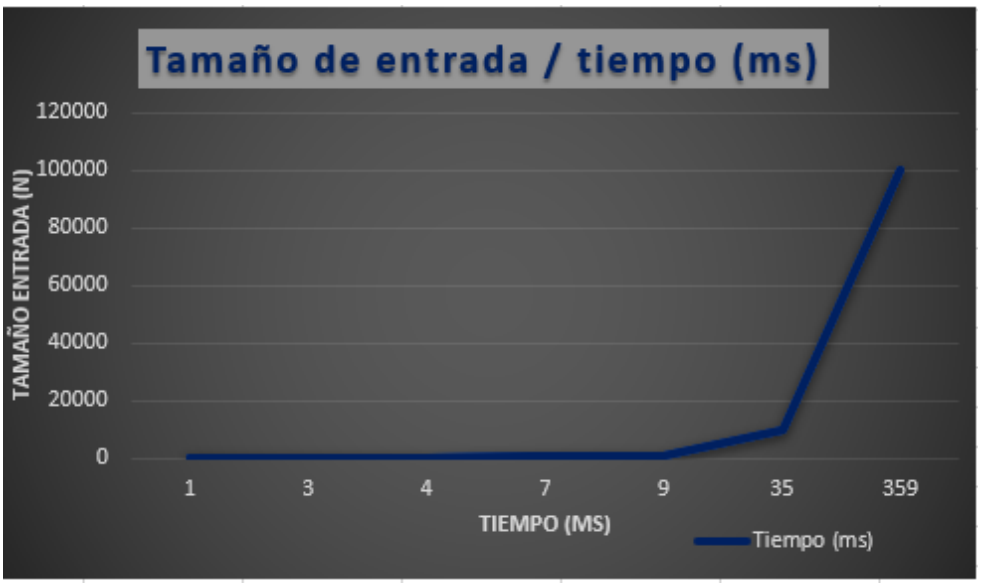
Además, para comprobar que todos estos test funcionan, hemos implementado unos juegos de prueba, en los que hemos implementado varios test:

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 código), en nuestro repositorio de GutHub: https://github.com/jlp717/ual_eda2_2022

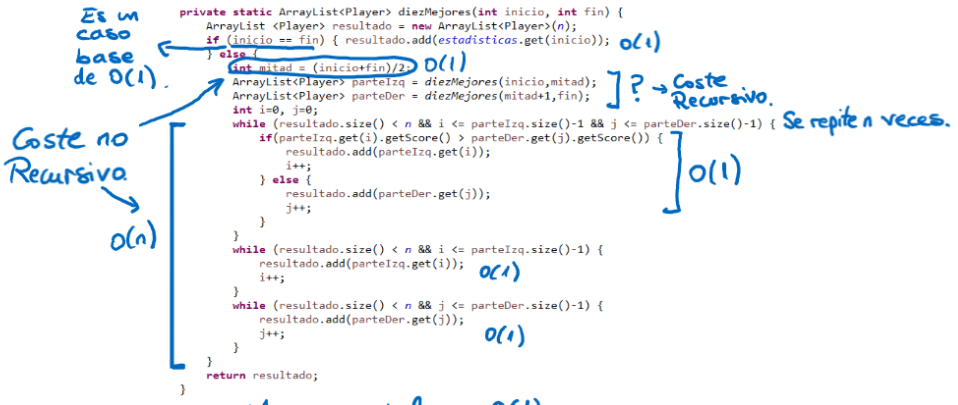
Por otra parte, nos quedan dos últimos puntos/cuestiones a tratar. La primera de ellas, es ir subiendo el top de jugadores que queremos obtener (para ello ir modificando el atributo int n que hemos declarado y enseñado al principio de la presentación), de tal manera en ver cómo afecta al rendimiento, Estos resultados los volcaremos en la tabla mostrada a continuación:

N	Tiempo (ms)
10	1
25	3
50	4
100	4
500	7
1000	9
10000	35
100000	359



Podemos sacar cómo conclusiones que cómo esperábamos, conforme vamos subiendo el tamaño de entrada (N), el tiempo de ejecución va aumentado, hasta el punto de que para sacar el top 100.000 de jugadores, tarda 359 ms.

Por último, vamos a calcular el orden de complejidad del algoritmo:




```
private static ArrayList<Player> diezMejores(int inicio, int fin) {
    ArrayList <Player> resultado = new ArrayList<Player>(n);
    if (inicio == fin) { resultado.add(estadisticas.get(inicio)); O(1)
    } else {
        int mitad = (inicio+fin)/2; O(1)
        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;
}
```

*Agregar equivale a O(1).

$t(n) \rightarrow$ caso base $\rightarrow O(1) \rightarrow n^{k_1} = n^0 \Rightarrow k_1 = 0.$

$t(n) \rightarrow a\,t(n/b) + g(n)$.

Coste Recursivo. Coste no Recursivo

$2t(n/2) + O(n) \rightarrow n^{k_2} = n^1 \Rightarrow k_2 = 1.$

$a = 2$

$b = 2$

$k = \max(k_1, k_2) = 1.$

Siendo $a = b^k$, pues $2 = 2$, optamos por:

*Orden de complejidad $\rightarrow O(n^1 \log n)$

ya que $k = 1$.

El caso de antes únicamente ocurre cuando el top es igual a n. Ahora, ¿Qué pasará si fuese un número?
Primeramente, un número x se considera constante, luego, el coste del while de la composición equivaldría a O(1). Partiendo de eso, el orden del caso base se queda igual.
Sin embargo, en cuanto a lo demás:

$t(n) \rightarrow 2t(n/2) + O(1) \rightarrow n^{k_2} = n^0 \Rightarrow k_2 = 0.$

$a = 2$

$b = 2$

$k = \max(k_1, k_2) = 0.$

Como se cumple que $a > b^k$ pues $2 > 2^0$:

*Orden de complejidad $\rightarrow O(n^{\log_2 2}) = O(n)$.