

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

23-3-2022

Practica 01

ESTRUCTURA DE DATOS Y
ALGORITMOS II

Several thin, curved lines in dark blue and light grey originate from the bottom left and curve upwards and to the right.

Hamza El Fallah
Nada Diouane

Contenido

1- Objetivo	2
2- Resolución practica del problema	2
3- Estudio Teórico.....	5
4- Estudio de los tiempos de ejecución:.....	6

1- Objetivo

El Objetivo de esta practica es tener los mejores jugadores aplicando la técnica Divide y Vencerás.

Vamos a utilizar la estructura de array para almacenar los datos de cada jugador.

Utilizamos la clase Player y creamos otras clases para resolver el problema, donde vamos a implementar métodos, especialmente el método recursivo.

2- Resolución practica del problema

Empezamos con la clase player donde añadimos dos métodos

- addPlayer que recibe 3 parametros
 - Team
 - Position
 - Score

Dicho método no hace nada si el score es 0, en caso contrario añadirá el team, positon y score, donde se actualiza sumando al valor anterior recibido por parámetro.

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

- El método getScore hace la devolución del score de cada jugador dividido por el tamaño del array de su equipo.

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

Creamos una clase llamada SolucionNBA donde hemos implementado métodos entre ellos hay 3 importantes:

- cargarArchivo: Este método lo que hace es cargar el archivo con la ruta que se le pasa por parámetro

```
21 public static void cargarArchivo(String filePath) {
22
23     try {
24         nba = new ArrayList<Player>(); // donde se guarda la informacion
25         Scanner sc = new Scanner(new File(filePath)); // leerlo con scanner
26         String linea = ""; // leer la linea
27         String[] arrayString;
28         Player ultimoJugador = null; // jugador ultimo cargado
29         String ultimoNombreJ = ""; // nombre jugador ultimo cargado
30         while (sc.hasNextLine()) {
31             linea = sc.nextLine().trim(); // quitar espacio
32             if (linea.isEmpty() || linea.startsWith("#")) // si la linea no esta vacia y comienza con Blanco la saltamos
33                 continue;
34             arrayString = linea.split(";");
35             // verificar que si tenemos los datos completos o no es decir si arrayString menor o mayor de la longitud esperada
36             if (arrayString.length != 9)
37                 continue; // continuar no cargar porque es distinto de 9
38             double fg = remplazar(arrayString[7]);
39             double pts = remplazar(arrayString[8]);
40             // el nombre jugador esta distinto del nombre de jugador ultimo cargado hay que crear el nuevo jugador
41             if (!arrayString[2].equals(ultimoNombreJ)) {
42                 // crear nuevo jugador con el nombre, teams, position, score donde hay que calcular el score
43                 ultimoJugador = new Player(arrayString[2], arrayString[6], arrayString[4], (int) (fg * pts / 100));
44                 nba.add(ultimoJugador); // agregar a la lista despues de crearlo
45                 ultimoNombreJ = arrayString[2];
46             } else {
47                 ultimoJugador.addPlayer(arrayString[6], arrayString[4], (int) (fg * pts / 100));
48             }
49         }
50         sc.close();
51         // el catch por si salta un error o si no esta el archivo
52     } catch (FileNotFoundException e) {
53         System.out.println("Error: archivo no encontrado");
54     }
55 }
56
57 }
```

- bestPlayer, El siguiente método estático lo que hace la comprobación de si el tamaño del nba es vacío o no, en caso es afirmativo se imprime el mensaje. Si no se crea un objeto arrayList de tipo Player y devolvemos el resultado.

```
71 public static ArrayList<Player> bestPlayer() {
72     if (nba.size() == 0) {
73         // System.out.println("No data");
74         throw new RuntimeException("no hay datos");
75     } else {
76         ArrayList<Player> bPlayers = bestPlayer(0, nba.size() - 1);
77         return bPlayers;
78     }
79 }
80 }
```

- bestPlayer es el método recursivo, recibe dos parámetros, este método está dividido por 4 partes.
 - 1- Caso base, se agrega a la lista con un coste de 1
 - 2- Descomposición: se hace la división del problema en 2
 - 3- Recursiva, se hace la comprobación de si ha llegado al caso base o no, cuando es así se agrega el jugador en la posición correspondiente de cada sub-array
 - a. `ArrayList<Player> sProblema1 = bestPlayer(inicio, mitad);`
se ejecuta cuando se divide el problema en dos. El método hace la llamada a sí mismo, para simplificar o reducir el problema en cada llamada, hasta llegar a un caso base que hemos establecido. Es decir, se actualiza la variable mitad hasta llegar al caso base y guarda el resultado en sProblema1.
 - b. `ArrayList<Player> sProblema2 = bestPlayer(mitad + 1, fin);`
Hace el mismo que el primero, lo único es empezar desde la posición que esta después de la mitad hasta el fin
 - 4- La última parte, hacemos uso de tres bucles while para la composición del resultado.
 - a. El primero consiste en establecer tres condiciones. Establecemos una variable "top" que es el máximo de jugadores que necesitamos mostrar. Mientras que los índices no alcancen el tamaño máximo de sus estructuras correspondientes se hace la comparación entre los jugadores contenidos en cada índice, en caso de que el contenido del índice "i" correspondiente a sProblema1 sea mayor que el del índice "j" correspondiente a sProblema2 se añade a "result" el resultado del índice "i". En el caso contrario se añade el resultado del índice "j". Todo esto mientras que no se cumpla una de estas tres condiciones.
 - b. En nuestro segundo bucle while cuando sale del primero y quedan elementos a comparar en sProblema1 se van añadiendo a "result" ya que se habrían añadido todos los de sProblema2 siempre y cuando el "top" no llegue al máximo.
 - c. En nuestro tercer bucle while cuando sale del primero y quedan elementos a comparar en sProblema2 se van añadiendo a "result" ya que se habrían añadido todos los de sProblema1 siempre y cuando el "top" no llegue al máximo.

En los tres casos cuando el "top" alcanza el máximo se deja de ejecutar el método y se devuelve el resultado.

```

82 public static ArrayList<Player> bestPlayer(int inicio, int fin) {
83     ArrayList<Player> bPlayers = new ArrayList<Player>();
84     //Caso base con coste 1 O(1)
85     if (inicio == fin) {
86         //Agregar a la lista con coste 1 O(1)
87         bPlayers.add(nba.get(inicio));
88     } else {
89         // con coste 1 O(1)
90         int mitad = (inicio + fin) / 2;
91         //2 llamadas recursivas
92         //Guardamos en la lista la llamada recursiva bestPlayer entre inicio y mitad
93         //
94         ArrayList<Player> sProblema1 = bestPlayer(inicio, mitad);
95         ArrayList<Player> sProblema2 = bestPlayer(mitad + 1, fin);
96         int i = 0;
97         int j = 0;
98         //top en el peor de los casos se ejecuta el bucle n veces si cojemos uno de la derecha y otro de la izquierda
99         while (bPlayers.size() < top && i <= sProblema1.size() - 1 && j <= sProblema2.size() - 1) {
100             if (sProblema1.get(i).getScore() > sProblema2.get(j).getScore()) {
101                 //agregar al final de la lista 1 con coste1
102                 bPlayers.add(sProblema1.get(i));
103                 i++;
104             } else {
105                 //agregar al final de la lista 2 con coste1
106                 bPlayers.add(sProblema2.get(j));
107                 j++;
108             }
109         }
110         //agregar con coste1
111         while (bPlayers.size() < top && i <= sProblema1.size() - 1) {
112             bPlayers.add(sProblema1.get(i));
113             i++;
114         }
115         //agregar con coste1
116         while (bPlayers.size() < top && j <= sProblema2.size() - 1) {
117             bPlayers.add(sProblema2.get(j));
118             j++;
119         }
120     }
121     return bPlayers;
122 }
123

```

3- Estudio Teórico

En método recurso que se observa en la captura anterior tiene la parte no recursiva que tiene un orden n para la composición del resultado.

- **while (bPlayers.size() < top && i <= sProblema1.size() - 1 && j <= sProblema2.size() - 1)**

Se ha usado la siguiente formula para llevar a cabo la parte recursiva.

O(t(n))

$$t(n) = c_1 n^{k_1}$$

Para $0 \leq n < b$

$$t(n) = at(n/b) + c_2 n^{k_2}$$

En $n \geq b$

- El orden de complejidad, en este caso, es:

$$t(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Donde $k = \max(k_1, k_2)$

$t(m) \rightarrow$ caso base $O(1) \rightarrow m^{K_1+1} = m^0 \Rightarrow K_1 = 0$
 $t(m) \rightarrow a \cdot t\left(\frac{m}{b}\right) + g(m) \rightarrow m^{K_2+2} = m^1 \Rightarrow K_2 = \frac{1}{2}$

- $g(m)$: Es el coste no recursivo
- a : Las veces que se hace la llamada recursiva
- b : numero de subproblemas

$a = 2$ y $b = 2$
 $K = (\max(K_1, K_2)) = \frac{1}{2}$
 Siendo nuestro caso $a = b^K$
 $O(m \log m)$
 Si ponemos $m = 10$, el orden no recursivo seria $O(10)$

$t(m) \rightarrow$ caso base $O(1) \rightarrow m^{K_1} = m^0 \Rightarrow K_1 = 0$
 $t(m) \rightarrow a \cdot t\left(\frac{m}{b}\right) + O(10) \rightarrow m^{K_2} = m^0 \Rightarrow K_2 = 0$
 $a > b^0 \Rightarrow O(m^{\log_b a}) > O(m)$

4- Estudio de los tiempos de ejecución:

