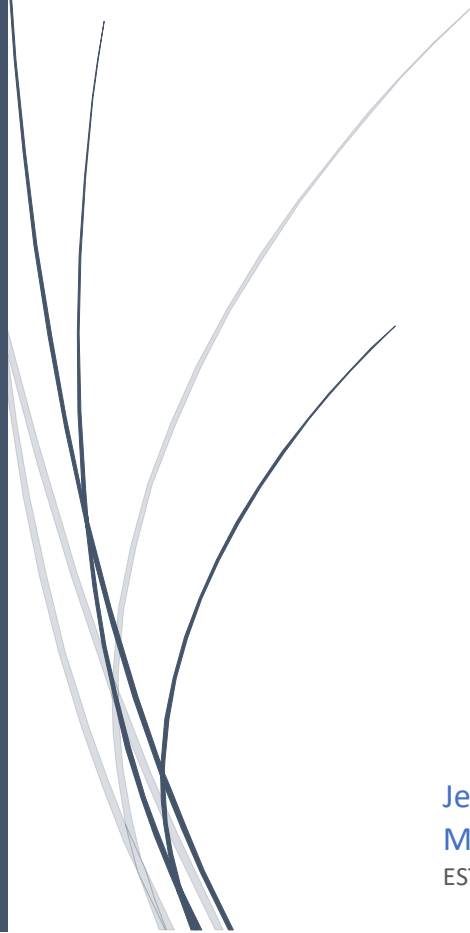


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

1-1-2022

Divide y vencerás

Práctica 01

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

Jefferson Alberto Miranda Lucas
Manuel Martínez Galera
ESTRUCTURA DE DATOS Y ALGORITMOS II

1. Introducción

Esta práctica consistirá en averiguar cuales son los diez mejores jugadores de la NBA desde al año 1950 al 2017. Para ello, disponemos de un archivo (nbastats.csv) que recopila información sobre el nombre de los jugadores, su equipo, posición, puntos...

Para saber que jugador es el mejor tendremos que calcular una variable llamada score, la cual, es el resultado de un cálculo entre el porcentaje de tiros acertados y los puntos anotados en una temporada y será la que utilicemos para comparar que jugador es mejor que otro. Dado que un mismo jugador se puede encontrar en otro año con una puntuación, equipo y posición distinta, se ha optado por hacer una media entre los scores de todos los años, haciendo así aun más fiable la solución de nuestro problema.

El método usado para resolver esta cuestión es “Divide y vencerás” un algoritmo recursivo que divide un problema en varios subproblemas y veremos más a fondo en los siguientes puntos del informe de esta práctica.

2. Divide y vencerás

2.1. Definición

El algoritmo divide y vencerás es un algoritmo que plantea resolver un problema a partir de la resolución de varios subproblemas más pequeños debiendo ser estos del mismo tipo que el problema principal.

El fundamento básico de este algoritmo es la recursividad ya que para resolución de los subproblemas mencionados se basará en una resolución directa o bien de forma recursiva, Una vez se hayan resuelto todos los subproblemas debemos combinar todas las soluciones obtenidas para poder obtener la solución final.

2.2. Partes del algoritmo

Partiendo de la definición anterior, vamos a dividir el algoritmo en tres partes para poder analizarlo usando nuestro código Java. Por nuestra parte, disponemos de dos métodos:

- Public static void mergeArray(int start, int mid, int end)

```
public static void mergeArray(int start, int mid, int end) {
    ArrayList<Player> aux = new ArrayList<Player>();

    int left = start;
    int right = mid + 1;

    while ((left <= mid && right <= end) || aux.size() == top) {
        if (players.get(left).compareTo(players.get(right)) == 0 || players.get(left).compareTo(players.get(right)) == -1) {
            aux.add(players.get(right));
            right++;
        } else {
            aux.add(players.get(left));
            left++;
        }
    }

    while (left <= mid) {
        aux.add(players.get(left));
        left++;
    }

    while (right <= end) {
        aux.add(players.get(right));
        right++;
    }

    for (int i = 0; i < aux.size(); start++) {
        players.set(start, aux.get(i++));
    }
}
```

- Public static void dividirArray(int start, int end)

```
public static void dividirArray(int start, int end) {  
  
    if (start < end && (end - start) >= 1) {  
        int mid = (end + start) / 2;  
        dividirArray(start, mid);  
        dividirArray(mid + 1, end);  
        mergeArray(start, mid, end);  
    }  
}
```

Para la explicación del algoritmo debemos tener en cuenta que lo que buscamos es ordenar el ArrayList de jugadores de mayor a menor y después mostrar solo las diez primeras posiciones.

Dividir

Esta parte coincide en la definición con descomponer el problema en subproblemas del mismo tipo. En nuestro código lo comparamos con siguientes líneas de los métodos “mergeArray” y “dividirArray”.

```
int mid = (end + start) / 2;
```

Esta línea pertenece a dividirArray y la da valor a la variable mid indicando que es en esa posición donde dividimos el ArrayList en dos.

```
int left = start;  
int right = mid + 1;
```

Estas dos líneas del método mergeArray establecen el inicio de cada una de las partes del ArrayList, gracias a esto sabemos que el ArrayList de la izquierda va desde start hasta mid y que el ArrayList de la derecha abarca desde la siguiente mid+1 hasta end.

Vencer

Esta parte hace referencia a la resolución de problemas recursivamente.

```
dividirArray(start, mid);  
dividirArray(mid + 1, end);  
mergeArray(start, mid, end);
```

En el código podemos ver reflejada la llamada recursiva en las líneas del método dividirArray. Donde el propio método se llama a sí mismo hasta dividir el ArrayList a la menor unidad posible.

```

while ((left <= mid && right <= end) || aux.size() == top) {
    if (players.get(left).compareTo(players.get(right)) == 0 || players.get(left).compareTo(players.get(right)) == -1) {
        aux.add(players.get(right));
        right++;
    } else {
        aux.add(players.get(left));
        left++;
    }
}

while (left <= mid) {
    aux.add(players.get(left));
    left++;
}

while (right <= end) {
    aux.add(players.get(right));
    right++;
}

```

En este fragmento de código de mergeArray lo que buscamos es comparar los objetos de los ArrayList. Cuando el elemento de la izquierda es menor que el de la derecha insertamos en una estructura auxiliar el elemento de la derecha en caso contrario insertamos el de la izquierda ya que buscamos ordenar de mayor a menor. Es claro, que debemos incrementar las valores left y right cada vez que insertamos un elemento en la posición correspondiente

Combinar

Finalmente, Combinar es la parte que une todas las soluciones dando lugar a la solución del problema original.

```

for (int i = 0; i < aux.size(); start++) {
    players.set(start, aux.get(i++));
}

```

Para ello, hacemos uso de un bucle for como se muestra en la imagen. Es imprescindible el uso de la estructura auxiliar ya que es la que en cada llamada al método almacena cada solución obtenida, lo único que debemos hacer es insertar los datos de la estructura auxiliar en nuestra estructura principal cada vez que el método se ejecute.

3. Análisis de eficiencia

En este apartado, veremos la comparativa en nanosegundos entre la implementación de nuestro código inicialmente y nuestro código usando una mejora propuesta por nosotros.

Nuestro código estaba basado en la ordenación del ArrayList completo, pero analizando la situación hemos optado por implementar una condición que pare la ejecución cuando el ArrayList ya ha sido ordenado en la diez primeras posiciones.

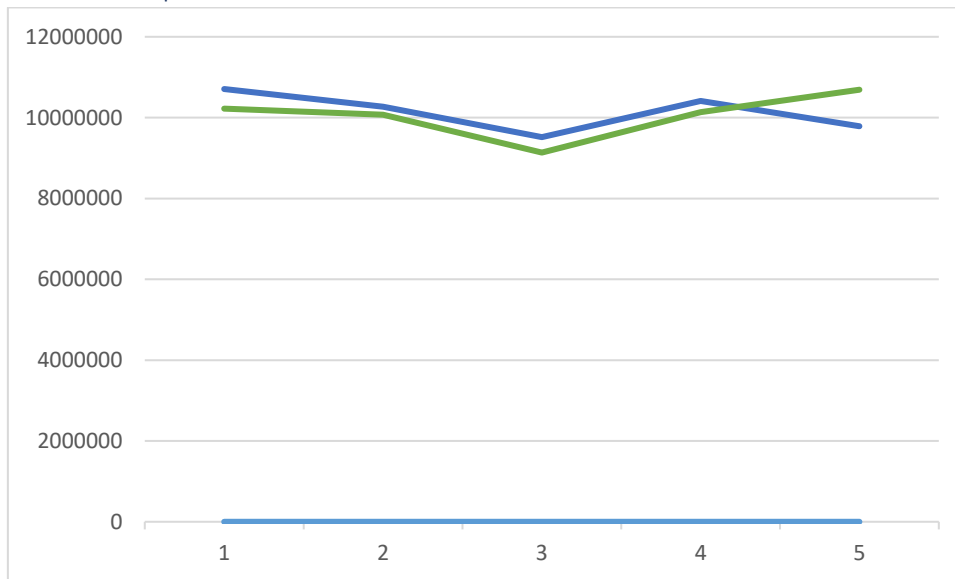
Tiempos de ejecución sin mejora

Numero de ejecución	Tiempo de ejecución (nanosegundos)
1	10224300
2	10073800
3	9135400
4	10136600
5	10690300

Tiempos de ejecución con mejora

Numero de ejecución	Tiempo de ejecución (nanosegundos)
1	10707600
2	10270500
3	9517700
4	10410400
5	9784000

Gráfica comparativa



En la gráfica que observamos la línea verde representa el tiempo de ejecución del algoritmo con mejora mientras que la azul representa el tiempo del algoritmo sin mejora.

Conclusión

Observando las distintas tablas y la gráfica, así como el código implementado hemos llegado a la conclusión de que la mejora propuesta para nuestro algoritmo es prácticamente insignificante puesto que como se puede ver en la gráfica los tiempos de ejecución de la mejora son levemente mayores y puesto que el tiempo es medido en nanosegundos la diferencia es ínfima. A pesar de esto, debemos destacar que hay puntos como por ejemplo en la última ejecución donde el tiempo de ejecución del algoritmo con mejora resulta en un mayor tiempo de ejecución.