Práctica 01

Estrategia Divide y Vencerás

Estructura de Datos y Algoritmos II

Rafael Expósito Mullor
Luis Díaz González
Universidad de Almería

Objetivos

- Conocer el método de resolución algorítmica Divide y Vencerás
- Realizar un análisis teórico de la complejidad de la solución aportada.
- Desarrollar una solución basada en la técnica de Divide y Vencerás
- Comprobar de manera empírica los resultados obtenidos en el problema.

1. Introducción

Queremos zanjar una pregunta que se hacen los analistas de EEUU desde tiempos inmemoriales, quienes son los mejores jugadores de la NBA de todos los tiempos. Para ello contamos con un conjunto de datos de las estadísticas de todos los jugadores desde el año 1950 al 2017. De cada jugador contamos con las siguientes estadísticas de cada uno en sus años en activo:

- `#`: Número identificativo de un jugador durante un año concreto (No nos interesa)
- `SeasonStart`: Año de la temporada.
- `PlayerName`: Nombre del jugador.
- `PlayerSalary`: Salario del jugador un año específico.
- `Pos`: Posición del jugador.
- `Age`: Edad del jugador.
- `TM`: Equipo.
- `FG%`: Porcentaje de aciertos en tiros de campo.
- `PTS`: Puntos anotados en la temporada.

En la siguiente imagen se muestra, a modo de ejemplo, un extracto del archivo de datos con las estadísticas mencionadas en el que podemos ver su estructura:

```
1 #;SeasonStart;PlayerName;PlayerSalary;Pos;Age;Tm;FG%;PTS
2 8035;1986;A.C. Green;;PF;22;LAL;53,9;521
3 8420;1987;A.C. Green;;PF;23;LAL;53,8;852
4 8807;1988;A.C. Green;;PF;24;LAL;50,3;937
5 9242;1989;A.C. Green;;PF;25;LAL;52,9;1088
6 9688;1990;A.C. Green;$1,750,000.00 ;PF;26;LAL;47,8;1061
7 10166;1991;A.C. Green;$1,750,000.00 ;PF;27;LAL;47,6;750
8 10617;1992;A.C. Green;$1,750,000.00 ;PF;28;LAL;47,6;1116
9 11060;1993;A.C. Green;$1,885,000.00 ;PF;29;LAL;53,7;1051
```

Figura 1. Extracto del archivo NbaStats.

Se nos pide implementar una solución recursiva basada en el esquema general de *Divide y Vencerás* que devuelva los 10 mejores jugadores de todos los tiempos si tan solo tenemos en cuenta su score, atributo coincidente con los puntos marcados durante un año multiplicado por FG%/100, teniendo así en cuenta el porcentaje de aciertos y no solo los puntos marcados.

2. Resolución del problema

Antes de comenzar a plantear la solución, es importante definir el concepto básico de un método de resolución DyV.
Dicho método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. Este proceso continuará hasta que estos lleguen a ser lo suficientemente sencillos como para que se resuelvan de manera directa.
Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original, siendo esta la base de los algoritmos eficientes para casi cualquier tipo de problema.

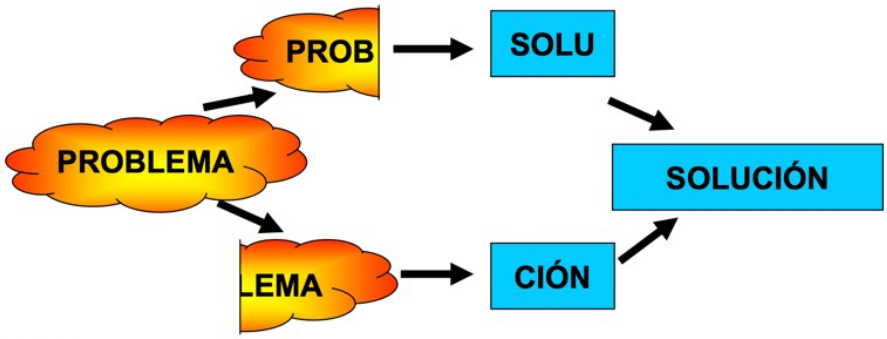


Figura 1. Esquema conceptual del método "Divide y Vencerás"

Centrándonos en el ejercicio que nos ocupa, partimos de la clase Player que se nos proporciona para la carga de los datos:

```
public class Player{

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

public Player(String playerName, String team, String position, int score){
   this.playerName = playerName;
   this.teams = new ArrayList<String>();
   this.teams.add(team);
   this.positions = new ArrayList<String>();
  this.positions.add(position);
  this.score = score;
 }

   public void add(String team, String position, int score) {
         if(score <= 0) return;
```

```
        this.teams.add(team);
        this.positions.add(position);
        this.score += score;
    }

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

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

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

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

    public String toString() {
        return this.playerName + ": " +getScore() + " puntos";
    }
```

En esta clase se ha tenido en cuenta como hacer la media del score, es decir, si un jugador ya está añadido en la estructura de datos, añadiremos el equipo en la variable *teams* y haremos la media del score que ya haya y el score correspondiente en el nuevo año.

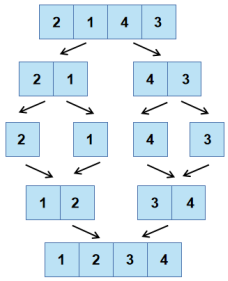
En definitiva, cada vez que se agregue a un jugador iremos sumando su *score* y como al agregarlo también se añade su equipo a la estructura *teams,* cuando queramos hacer la media obtendremos el tamaño de teams (*teams.size()*) y lo utilizaremos para dividir el *score.*

3. Algoritmo principal

Para la resolución del problema se ha utilizado como base la ordenación por mezcla (Mergesort) por ser relativamente sencillo de entender y un buen ejemplo de divide y vencerás.

El MergeSort posee una complejidad computacional logarítmica O(nlogn), como veremos más adelante, que permite ordenar un listado de elementos de manera rápida.

La idea básica es la siguiente:



- **Dividir el problema** (array) en dos trozos de igual tamaño o más parecido posible.
- **Resolver recursivamente** los subproblemas de ordenar los dos trozos.
- Si llegamos a tener **un solo elemento**, este ya estará ordenado.
- Se lleva a cabo un **proceso de combinación** de dos secuencias ordenadas.

En el siguiente extracto de código podemos verlo en detalle.

```
private static ArrayList<Player> mejoresJug(int inicio, int fin) {
    ArrayList<Player> resultado = new ArrayList<Player>(top); //
    if(inicio == fin) {
        resultado.add(nba.get(inicio));
    }else{
        int mitad = (inicio + fin)/2;
        ArrayList<Player> parte1 = mejoresJug(inicio, mitad);
        ArrayList<Player> parte2 = mejoresJug(mitad+1, fin);

        int i = 0;
        int j = 0;
        //Mientras haya elementos en los dos lados, derecho e izquierdo
        while(resultado.size() < top && i <= parte1.size()-1 && j<= parte2.size()-1) {
            if(parte1.get(i).getScore() > parte2.get(j).getScore()) {
                resultado.add(parte1.get(i));
                i++;
            }else {
                resultado.add(parte2.get(j));
                j++;
            }
        }
        //Si hay elementos en un lado y en otro no.
        while(resultado.size() < top && i<= parte1.size()-1) {
            resultado.add(parte1.get(i));
            i++;
        }
        while(resultado.size() < top && j <= parte2.size()-1) {
            resultado.add(parte2.get(j));
            j++;
        }
    }
    return resultado;
}
```

Consideraciones

El método privado será llamado por otro público con mismo nombre, en el que también se maneja el caso en el que no haya datos, lanzando una excepción. De esta forma conseguimos que desde el *main* solo sea necesario hacer una llamada al método público sin necesidad de introducir parámetros que, a veces, pueden llegar a introducirse de forma incorrecta.

En el código anterior se ha tenido en cuenta que la declaración del ArrayList *resultado* sea lo mas eficiente posible, inicializándolo a tamaño *top*, de esta forma conseguimos que nuestro array siempre tenga el tamaño justo y necesario, evitando que tenga que doblar su capacidad y traspasar los datos en los casos en los que se supere la capacidad inicial por defecto.

Con los tres bucles while tenemos en cuenta todos los casos posibles. En primer lugar, hemos dividido el problema en dos partes, el primer bucle se encargará de comprobar que hay elementos tanto en la parte 1 como en la parte 2 y de comparar dichos elementos por su puntuación, añadiendo uno u otro a nuestro array *resultado.*
Los otros dos bucles contemplan los casos en los que haya elementos en una parte y en la otra no. Aquí simplemente iremos añadiendo los elementos restantes de esa parte concreta al array.

En los tres while se ha añadido una condición extra para pararlos cuando ya hallamos alcanzado el número de elementos en nuestro array resultado (resultado.size() < top), es decir, si estamos pidiendo un top 10 de jugadores se irán cogiendo elementos ordenados de una parte o de otra solo hasta que alcancemos dicho top.

ANÁLISIS DE EFICIENCIA

```
private static ArrayList<Player> mejoresJugV2(int inicio, int fin) {
    ArrayList<Player> resultado = new ArrayList<Player>(top);
    if(inicio == fin) { Caso Base → Operación de agregar:
        resultado.add(nba.get(inicio));
    }else{
        int mitad = (inicio + fin)/2; O(1)
        ArrayList<Player> parte1 = mejoresJugV2(inicio, mitad);
        ArrayList<Player> parte2 = mejoresJugV2(mitad+1, fin);

        int i = 0;
        int j = 0;
        //Mientras haya elementos en los dos lados, derecho e izquierdo
        while(resultado.size() < top && i <= parte1.size()-1 && j<= parte2.size()-1) {
            if(parte1.get(i).getScore() > parte2.get(j).getScore()) {
                resultado.add(parte1.get(i)); Operación de agregar:
                i++;
            }else {
                resultado.add(parte2.get(j)); Operación de Agregar:
                j++;
            }
        }
        //Si hay elementos en un lado y en otro no.
        while(resultado.size() < top && i<= parte1.size()-1) {
            resultado.add(parte1.get(i)); Operación de agregar:
            i++;
        }
        while(resultado.size() < top && j <= parte2.size()-1) {
            resultado.add(parte2.get(j)); Operación de agregar:
            j++;
        }
    }
    return resultado;
}
```

Coste recursivo

Coste No recursivo

Bucle while: Ejecución n veces O(n)

Para este análisis estamos enfocándonos en el caso de trabajar con todos y cada uno de los jugadores (top N jugadores).
Una vez hecho un recorrido por el algoritmo para identificar los casos base y recursivo, así como los diversos costes, podemos formular lo siguiente:

Caso Base: **T(n) = O(1)** → $n^{k1} = n^0$ → $K_1 = 0$
T(n) = at(n/b) + g(n) = **2t(n/2) + O(n)** → $n^{k2} = n^1$ → $K_2 = 1$

Por tanto, tenemos que:

a = 2
b = 2
k = max (k_1, k_2) = 1

y como **a = b^k** ($2 = 2^1$) podemos decir que el coste de nuestro algoritmo es el siguiente:

T(n) ∈ O(nlog(n))

Ya sabemos el orden de nuestro algoritmo en el caso de querer obtener los mejores N jugadores. Concretando un poco más, podríamos obtener el orden de complejidad para el caso concreto del ejercicio propuesto en la práctica, la obtención de los 10 mejores jugadores.

Si nos fijamos en el código anterior, vemos que la complejidad se reduce considerablemente. El coste no recursivo que obteníamos para el bloque de código formado por los tres bucles while se reduce a orden constante al hacer un número de llamadas fijo.
Por lo tanto, obtendríamos la siguiente formulación:

Caso Base: **T(n) = O(1)** → $n^{k1} = n^0$ → $K_1 = 0$
T(n) = at(n/b) + g(n) = **2t(n/2) + O(1)** → $n^{k2} = n^0$ → $K_2 = 0$

Y sabiendo que:

a = 2
b = 2
k = max (k_1, k_2) = 0

$a > b^k = 2 > 2^0$

Podemos indicar lo siguiente: **$T(n) \in O(n)$**, siendo este un orden de complejidad notoriamente mejor que en el caso anterior.

Para visualizar estos datos de forma más visual y ponerlos en práctica, hemos recopilado una serie de tiempos de ejecución para distintos tamaños de "n" y así observar como se comporta nuestro algoritmo para distintas cargas de trabajo, en concreto para n = 10, 50, 150, es decir, aquellos casos en los que quisiéramos obtener los 10, 50 o 150 mejores jugadores y cuando contamos con un número total de jugadores diferente en cada ejecución.

Hemos implementado la clase *Rendimiento* para hacer estos cálculos, la cual tiene la siguiente estructura:

```
public class Rendimiento {

	public static void main(String[] args) {
		long inicio;
		long fin;

		System.out.println("N\tTiempo");
		for(int i = 1; i<= 10000000; i*=2) {
			creaDatos(i, i);
			inicio = System.nanoTime();
			TopNba.mejoresJugV2();
			fin = System.nanoTime();
			System.out.println(i + "\t" + (fin-inicio));

		}
```

```
	}

private static void creaDatos(int N, int top) {
	TopNba.nba = new ArrayList<Player>();
	for(int i = 0; i < N; i++) {
		Player jugador = new Player("Jugador"+i, "", "",(int)
		(Math.random()*N));
		TopNba.nba.add(jugador);
	}
		TopNba.top = top;

	}
}
```

De esta forma obtenemos lo siguiente:

N	TiempoT10	TiempoT50	TiempoT150
1	13000	12541	12875
2	6000	5958	6000
4	9042	9792	9666
8	24292	26750	29042
16	64167	65250	58583
32	138750	225084	254917
64	534459	554209	977584
128	225750	890333	595792
256	433917	718459	645125
512	759625	3377250	2222750
1024	4409958	577959	707500
2048	511583	970625	1376708
4096	928833	2307875	3049000
8192	2185708	3768625	16315416
16384	5706375	17229375	7682459
32768	7676583	12455125	15459666
65536	25606167	26585334	47939292
131072	47356291	129380459	179860042
262144	232335625	77063875	107596750
524288	87099333	124026083	199856500
1048576	179069584	305114750	477331958
2097152	255099917	398692000	655561125
4194304	617497667	757407291	1112784833
8388608	1486359625	1,739E+09	3174567959

Figura 2. Tiempos obtenidos para distintos N.

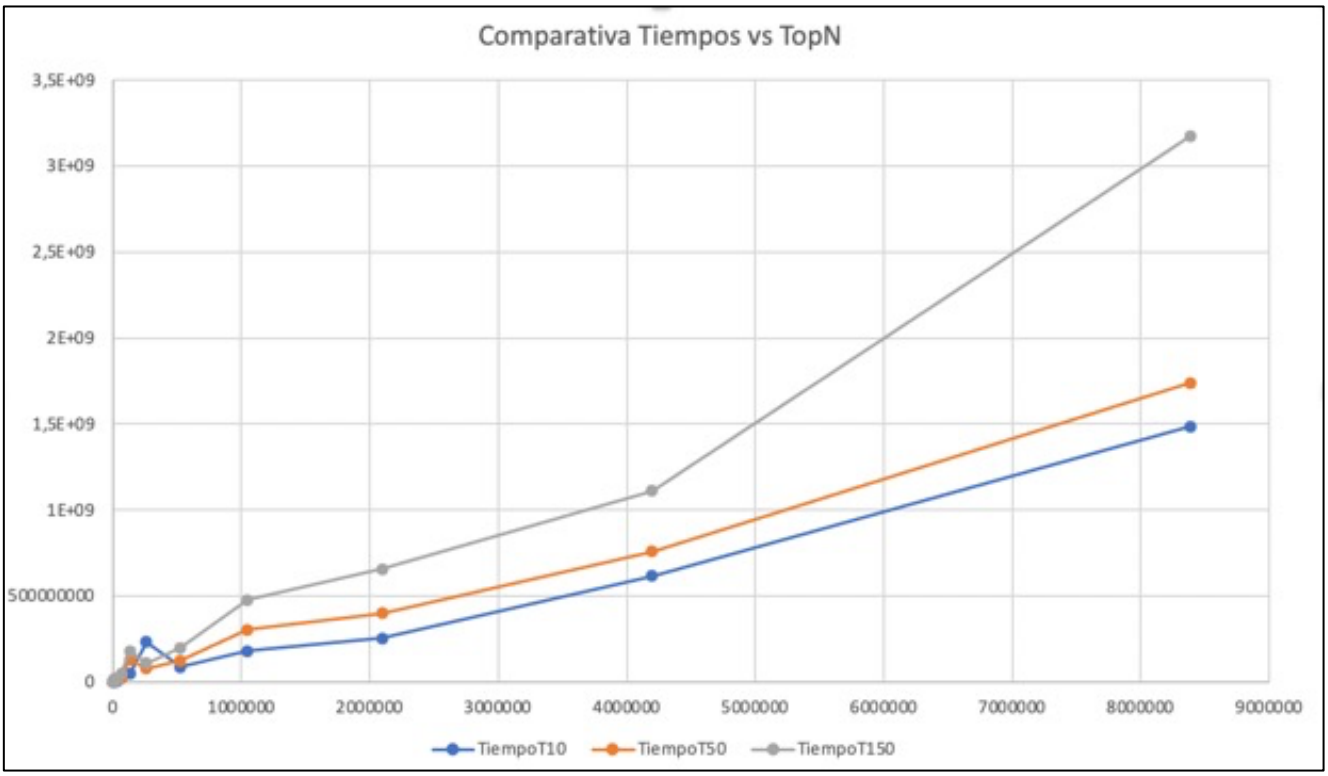


Figura 3. Grafica de tiempos.

Vemos como de forma general para mayores TopN mayores tiempos de ejecución arroja.