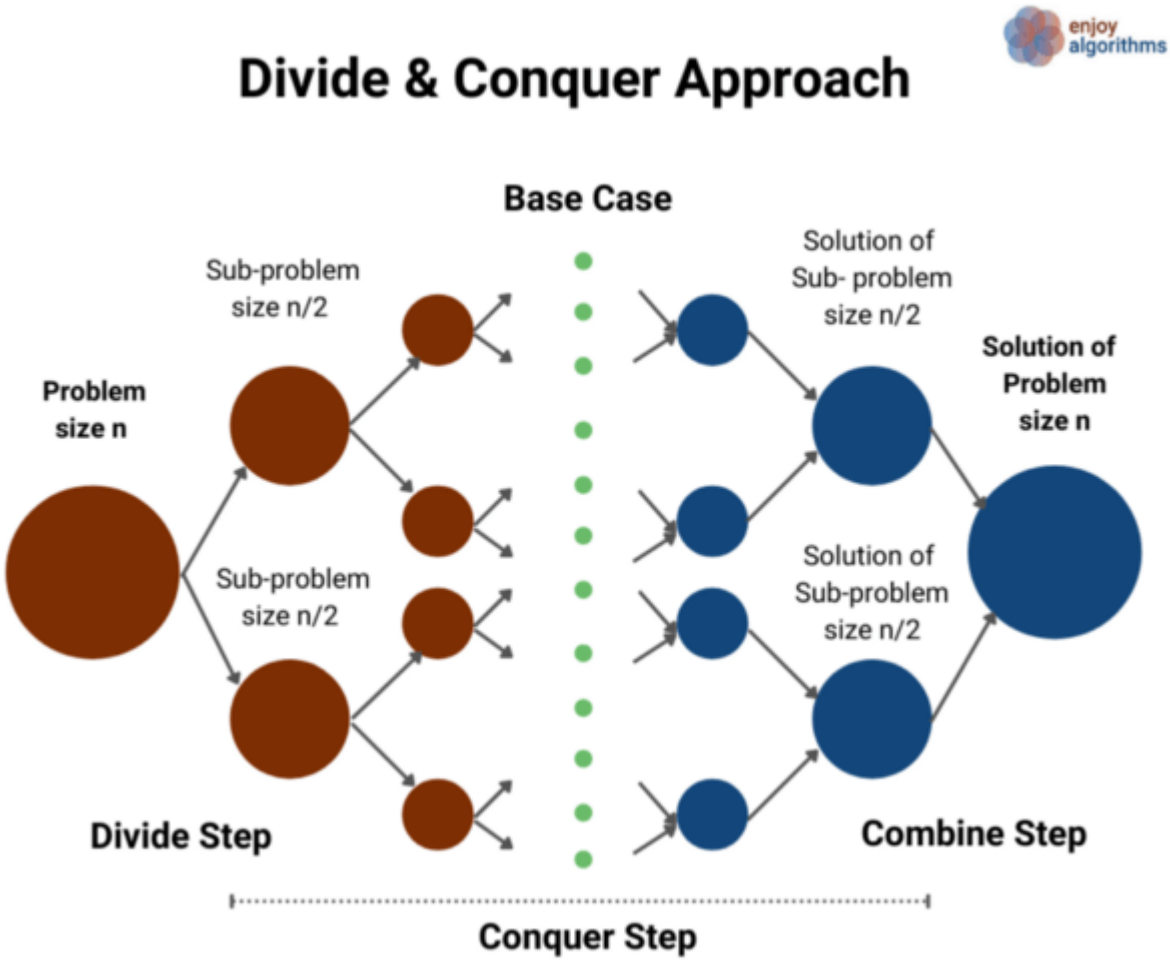
Práctica 1 EDA II, Algoritmo Divide y Vencerás.

Francisco Ramírez Vaquero



Metodología

Se han obtenido datos históricos acerca del número de puntos obtenidos por cada jugador y porcentaje de aciertos en cada temporada (entre otros datos), y para analizarlos se ha desarrollado un programa de línea de comandos que permite ordenar los jugadores según distintos criterios (en caso de que no se especifique uno, se ha establecido un criterio por defecto).

Por tanto, se ha tenido que resolver el problema de ordenar una serie de datos. Para ello, en la primera versión se ha utilizado el MergeSort, que consiste en dividir el array o estructura en 2 (aproximadamente la mitad) hasta tener estructuras de tamaño 1, y ordenar sucesivamente las estructuras de mayor tamaño (se explicará con mayor detalle más adelante).

Además, se ha hecho un estudio de rendimiento. Se espera obtener un orden de O(n*log(n)), ya que es el orden de MergeSort, pero se comprobará de forma tanto empírica como analítica.

Explicación de los algoritmos

Descomponer

En este caso hemos ido dividiendo el problema en 2 subproblemas, de la mitad de tamaño. Es esta parte del método:

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

ArrayList<Player> left = getBestPlayersMS(begin, mid, num, comp);
ArrayList<Player> right =getBestPlayersMS(mid + 1, end, num,comp);
```

Con la primera línea obtenemos la posición media del array, y las llamadas recursivas lo que hacen es ejecutar el método desde el inicio a la mitad (left) y desde la mitad (más uno) hasta el final (right).

Caso base

Es el momento en el que el problema no se puede seguir dividiendo (o el problema se puede resolver directamente).

```
if (begin == end) {
    solution.add(allPlayers.get(begin));
    return solution;
}
```

Lo que se hace es comprobar cuándo el tamaño del array es 1 (el único momento en el que la posición de inicio es la misma que la de fin), y se añade el elemento que se encuentre en dicha posición.

Combinar

Una vez se ha resuelto el problema base, hay que juntar las soluciones parciales. Para ello, he utilizado el algoritmo mergeSort, añadiendo la comprobación de que el array solution no sea mayor que el número máximo que queremos.

```
// Combinar es ordenar izquierda y derecha
int i = 0;
int j = 0;
while (solution.size() < num && i < left.size() && j <
right.size()) {
    if (comp.compare(left.get(i), right.get(j)) < 0) {
        solution.add(left.get(i));
        i++;
    } else {
        solution.add(right.get(j));
        j++;
    }
}

while (solution.size() < num && i < left.size()) {
    solution.add(left.get(i));
    i++;
}
while (solution.size() < num && j < right.size()) {
    solution.add(right.get(j));
    j++;
}
return solution;
```

De esta forma, cada vez que se ejecuta este método devuelve un ArrayList ordenado cuyo tamaño nunca va a ser mayor del que se pide, ganando tanto en tiempo como en memoria, ya que sólo ordenamos y almacenamos los elementos que realmente son relevantes.

Análisis teórico

El algoritmo MergeSort está ampliamente estudiado, su orden de complejidad es O(n*log(n)).

La demostración es la siguiente:

En el coste no recursivo influyen los 3 while y el caso base. El caso base es constante, y en total, los 3 while tienen tantas iteraciones como jugadores se quieren obtener (num, que llamaremos n a partir de ahora), y por la regla de la suma, el coste total es O(n).

El coste recursivo se define como t(n) = a*t(n/b). En este caso, como hay 2 llamadas recursivas, a = 2 y como dividimos el tamaño a la mitad, b = 2. Por tanto, el coste recursivo se define como t(n) = 2*t(n/2).

El coste no recursivo, g(n) = O(n). Para calcular la complejidad con notación O tengo: a = b = 2, k1 = 0, k2 = 1, k = 1. Como a = b^k = 2^1 = 2, tiene un coste de O(n*log(n)).

Análisis empírico

Para analizar el rendimiento, se ha creado un método que mide el tiempo que tarda en ejecutarse el método. No es exacto porque el método que realmente ordena es privado, y no se puede llamar desde fuera. Además, al ejecutarse dentro de un sistema operativo, hay tareas en segundo plano, y, por supuesto, depende del hardware en el que se ejecute. Por eso los algoritmos se miden en función de cómo aumenta el tiempo de ejecución en función del tamaño del problema, o lo que ocupa en memoria.

Dicho método consiste sencillamente en ejecutar varias veces el algoritmo con un tamaño de entrada fijo para obtener un tiempo medio con distintos tamaños, y se han representado en un gráfico, con el eje X representando el tamaño y el eje Y el tiempo.

El método que se encarga de medir guarda el tiempo medio que se ha tardado en ejecutar para cada tamaño en un fichero .csv. Usando Excel, se ha generado el siguiente gráfico en el que además se compara la curva de crecimiento con el de la función