Ejercicio:

10MejoresJugadores

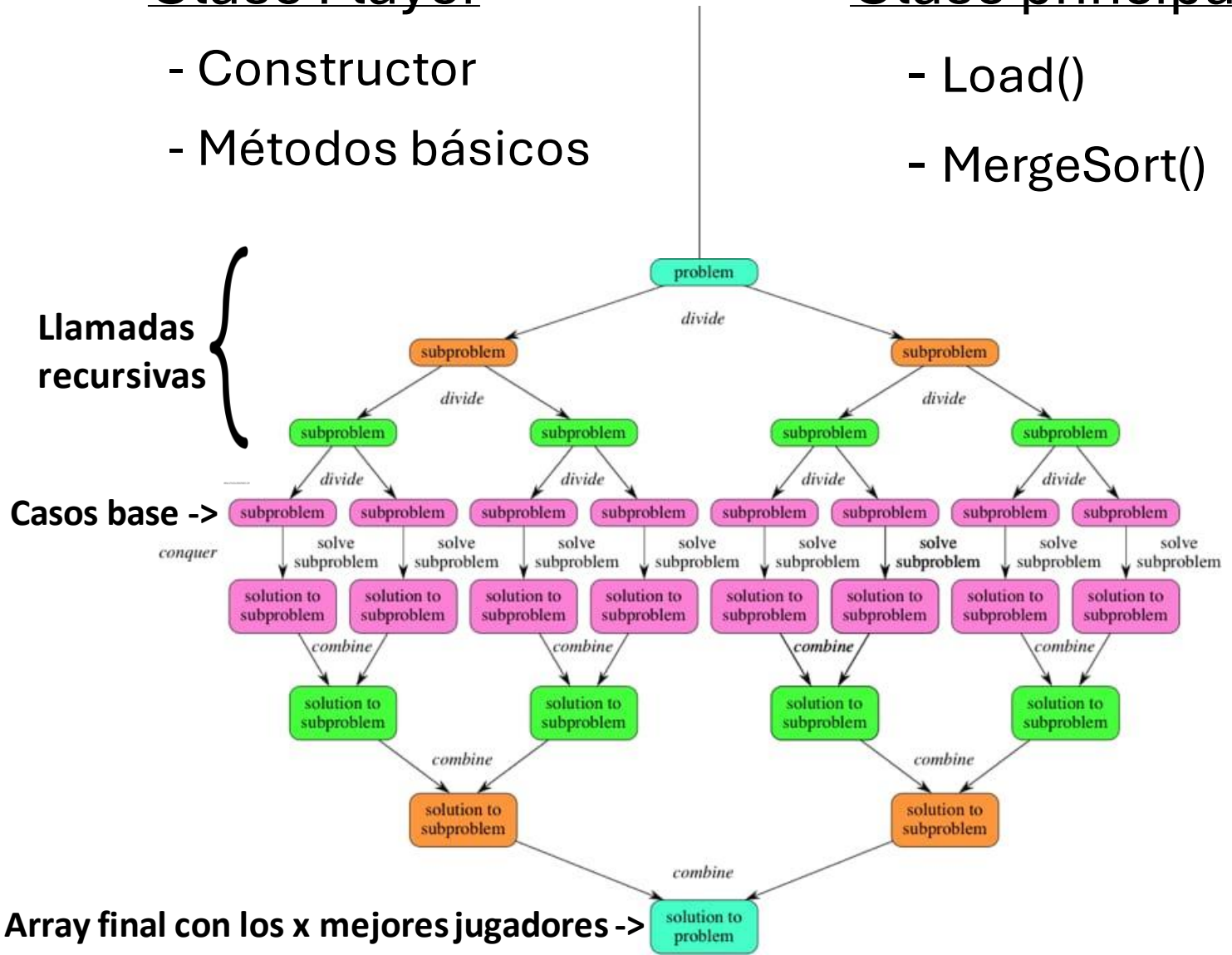
Técnica de resolución:

Divide y vencerás

Algoritmo utilizado:

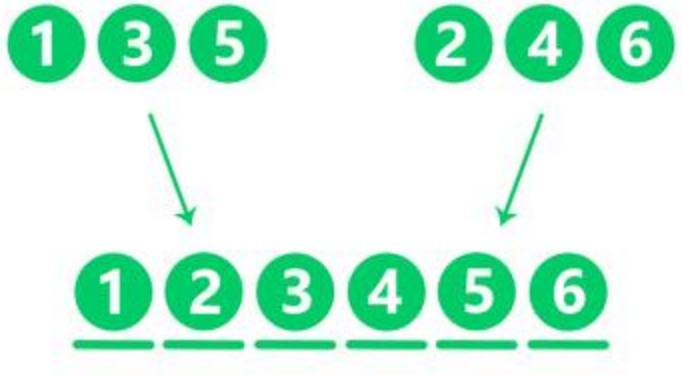
MergeSort

- Clase Player
  - Constructor
  - Métodos básicos
- Clase principal
  - Load()
  - MergeSort()



MergeSort

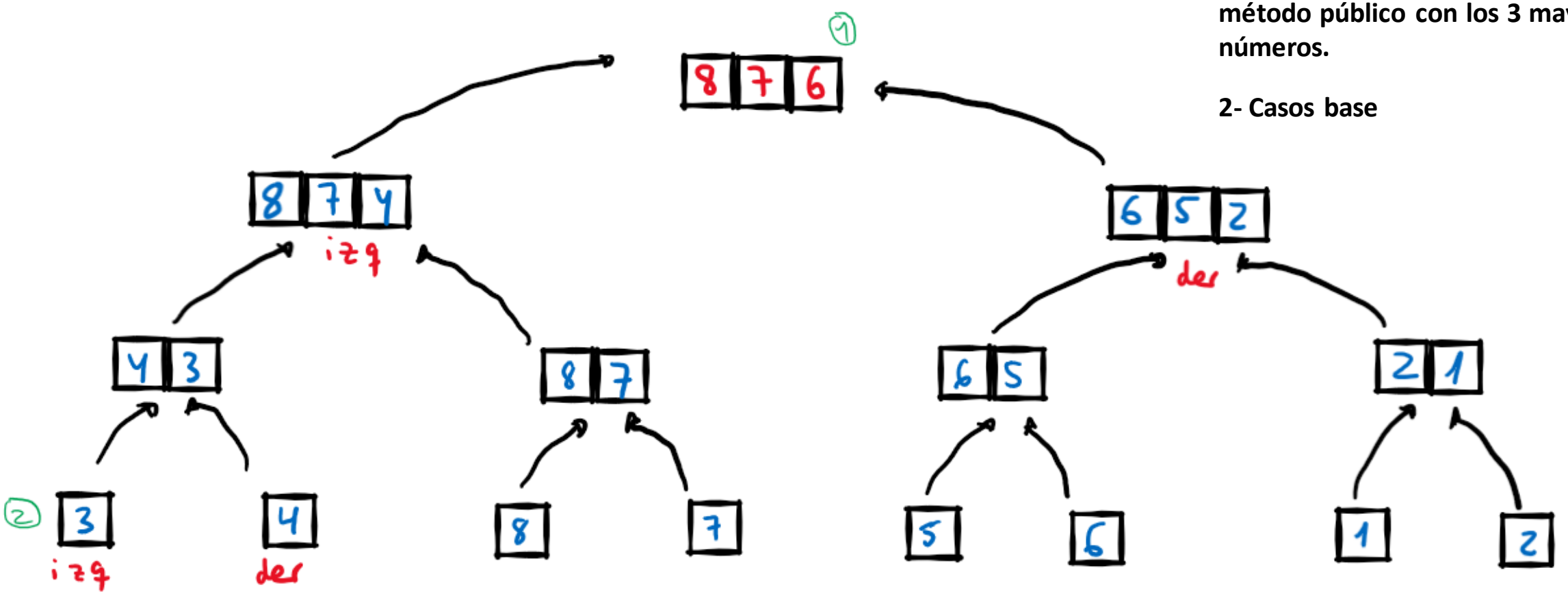
- Para obtener los diez mejores jugadores del archivo NbaStats.txt, hemos hecho uso del algoritmo MergeSort.
- Implementac



Visión gráfica del algoritmo

1- Array final que nos devuelve el método público con los 3 mayores números.

2- Casos base



Mejora

Uso del Tamaño del Array final "result" proporcionado previamente para prescindir así del uso de un tercer índice.

De esta manera optimizamos en uso de recursos, memoria, y por tanto en tiempo de ejecución.

```
private static ArrayList<Player> bestPlayersV2(int inicio, int fin) {
    ArrayList<Player> result = new ArrayList<Player>(numPlayers);
```

```
    int i=0;
    int j=0;
    while(result.size() < numPlayers && i <= p1.size()-1 && j <= p2.size()-1) {
        if(p1.get(i).getScore() > p2.get(j).getScore()) {
            result.add(p1.get(i));
            i++;
}
      else {
      result.add(p2.get(j));
      j++;

      }
}

while(result.size() < numPlayers && i <= p1.size()-1) {
      result.add(p1.get(i));
      i++;
}

while(result.size() < numPlayers && j <= p2.size()-1) {
      result.add(p2.get(j));
      j++;

    }
```

Orden de complejidad

```
    if (inicio == fin) {
    result.add(players.get(inicio));
} else {
    int mitad = (inicio + fin) / 2;
    ArrayList<Player> p1 = bestPlayersV2(inicio, mitad);
    ArrayList<Player> p2 = bestPlayersV2(mitad+1, fin);

    int i=0;
    int j=0;
    while(result.size() < numPlayers && i <= p1.size()-1 && j <= p2.size()-1) {
        if(p1.get(i).getScore() > p2.get(j).getScore()) {
            result.add(p1.get(i));
            i++;
        }
        else {
        result.add(p2.get(j));
        j++;
        }
}

while(result.size() < numPlayers && i <= p1.size()-1) {
        result.add(p1.get(i));
        i++;
}

while(result.size() < numPlayers && j <= p2.size()-1) {
        result.add(p2.get(j));
        j++;
    }
}
```

- Costes O(1).
- Costes O(n).
- Llamadas recursivas.

Cálculos realizados para obtenerlo

Caso base: $t(n) = O(1)^{k_1} \Rightarrow k_1 = 0$

Coste recursivo + coste NO recursivo:

$t(n) = a\,t(n/b) + g(n) = 2\,t(n/2) + O(n)^{k_2} \Rightarrow k_2 = 1$

- $a\,t(n/b)$ → coste recursivo.
- $g(n)$ → coste no recursivo.

$a = 2$ (llamadas recursivas).

$b = 2$ (partes en las que se divide el array).

Dado $a = b^k \Rightarrow 2 = 2^1$ obtenemos

Orden de complejidad de $O(n \log(n))$.

Dado que $a = b^k \Rightarrow 2 = 2$ y $O(n^k \log(n)) = O(n \log n)$