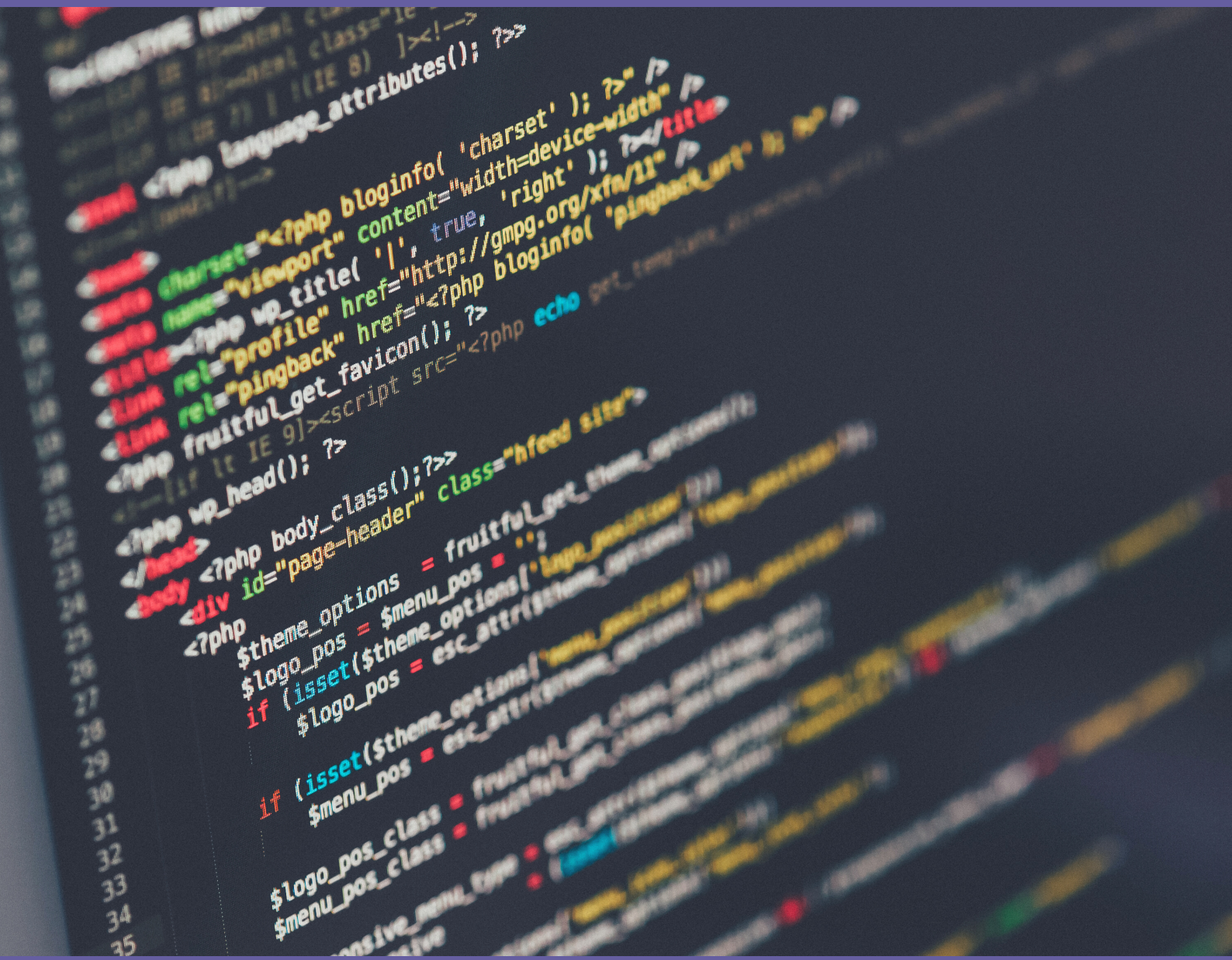
ESTRUCTURA DE DATOS Y ALGORITMOS II

Practica 1:
Divide y venceras



JUAN RAUL MELLADO GARCIA
ALEJANDRO MARTINEZ TERRONES

ALGORITMOS

Heap

Este Algoritmo divide los datos entre 2 recursivamente hasta llegar a tener arrays de tamaño 1

```
public PriorityQueue<Jugador> reduce() { // O(n*log(n))

    PriorityQueue<Jugador> data = new PriorityQueue<>();
    ArrayList<ArrayList<Jugador>> aux = new ArrayList<>();
    ArrayList<ArrayList<Jugador>> dividir = null;
    // boolean seguir=true;

    while (aux.isEmpty() || aux.get(aux.size() - 1).size() != 1) { //n
        if (aux.isEmpty()) {
            dividir = this.divide(this.datos); // n * log (n)
            aux.addAll(dividir);
        } else {
            dividir = new ArrayList<>();
            for (int i = 0; i < aux.size(); i++) {
                if (aux.get(i).size() > 1)
                    dividir.addAll(divide(aux.get(i)));
                else
                    dividir.add(aux.get(i));
            }
            aux = new ArrayList<>();
            aux.addAll(dividir);
        }
    }
```

```java
private ArrayList<ArrayList<Jugador>> divide(ArrayList<Jugador> jugadores) { // O(log(n))

    ArrayList<ArrayList<Jugador>> data = new ArrayList<ArrayList<Jugador>>();
    ArrayList<Jugador> aux = new ArrayList<Jugador>();

    int mitad = jugadores.size() / 2;

    for (int i = 0; i < mitad; i++) {
        aux.add(jugadores.get(i));
    }

    data.add(new ArrayList<Jugador>(aux));

    aux.clear();
    for (; mitad < jugadores.size(); mitad++) {
        aux.add(jugadores.get(mitad));
    }

    data.add(new ArrayList<Jugador>(aux));
    aux.clear();

    return data;
}
```

Al tener los arrays de tamaño 1, introduce los datos en una cola de prioridad para ir ordenándolos, y cuando la cola de prioridad supera los 10 elementos, se elimina el elemento de menor tamaño.

```java
    for (int i = 0; i < aux.size(); i++) {
        data.add(aux.get(i).get(0));
        while (data.size() > 10)
            data.poll();

    }

    return data;

}
```

Su complejidad es de O(n*log(n)) pero el tiempo de inserción en la cola hace que sea poco eficiente.

MergeSort

Este conocido Algoritmo de ordenación se basa en en subdividir el problema en problemas más pequeños hasta llegar al caso base , que en este caso es que solo contemos con un ítem, una vez llegado al caso base , empezamos a mezclar ordenadamente los grupos de ítems hasta tener un array o estructructura de datos ordenados.

```
private void mergesort(int izq, int der) { // O(n*log(n))
    if (izq < der && (der - izq) >= 1) {
        int medio = (izq + der) / 2;
        mergesort(izq, medio); // log(n)
        mergesort(medio + 1, der); // log(n)
        merge(izq, medio, der); // n

    }
}
```

Como podemos observar en nuestro caso dividimos nuestro arraylist mediante una llamada recursiva al método mergesort, y una vez en el caso base empezamos a unir los items con el método merge.

```java
private void merge(int izq, int medio, int der) {
    int i, j, x;
    ArrayList<Jugador> aux = new ArrayList<Jugador>();

    j = medio + 1;
    x = izq;

    while (izq <= medio && j <= der) {
        if (this.datos.get(izq).getScore() > this.datos.get(j).getScore()) {
            aux.add(this.datos.get(izq));
            izq++;
        } else {
            aux.add(this.datos.get(j));
            j++;
        }
    }

    while (izq <= medio) {
        aux.add(this.datos.get(izq));
        izq++;

    }
    while (j <= der) {
        aux.add(this.datos.get(j));
        j++;
    }
    i = 0;
    while (i < aux.size()) {
        this.datos.set(x, aux.get(i++));
        x++;
    }

}
```

A este método le llegan las posiciones de inicio y final de los “paquetes de ítems” y empieza a comparar ordenadamente los ítem , introduciendo siempre los ítems de mayor valor en un array auxiliar hasta que uno de los dos paquetes quede vacío, en ese momento termina de llenar el array auxiliar con los ítems que quedaban en el otro paquete (en caso de que existieran).

Y para finalizar y haciendo uso de los límites que le han llegado por parámetro, sustituye en en la estructura de datos original los valores de esta franja ordenados.

DeDiezEnDiez

Este algoritmo divide el array de datos en subarrays ordenados, usando colas de prioridad, de tamaño máximo 10, cuando tiene todos los subarrays, junta el último con el penúltimo y elimina los valores 10 valores inferiores

```
public PriorityQueue<Jugador> deDiezEnDiez(){ //O(n*log(n))

    PriorityQueue<Jugador> result = new PriorityQueue<>();
    ArrayList<PriorityQueue<Jugador>> aux = new ArrayList<>();

    for(int i = 0 ; i<this.datos.size(); i ++) { // n
        aux.add(new PriorityQueue<Jugador>());
        for(int cont = 0; cont<10; cont++) {
            if((10*i+cont)<this.datos.size())
                aux.get(i).add(this.datos.get(10*i+cont)); // n * log (n)
            //if((i+1)<this.datos.size())
                //i++;
        }

    }

    for(int i = aux.size()-1 ; i>=0 ; i--) {
        //System.out.println(i);

        if(i==aux.size()-1) {
            //System.out.println(aux.get(i));
            result.addAll(aux.get(i));
            //System.out.println(result.size());
            i--;
        }
        result.addAll(aux.get(i));
        while(result.size()>10) {
            result.poll();
            //System.out.println(result.size());
        }

    }

    return result;
}
```

Este algoritmo tiene una complejidad de n(log(n)) y consigue ser mas eficiente que mergesort pero menos que mergesort2 (mergesort mejorado)

MergeSort 2

Este algoritmo en una mejora del anterior MergeSort

```java
public ArrayList<Jugador> mergesort2() { //O(n*log(n))
    mergesort(0, this.datos.size()/2); // n*log (n)
    mergesort(this.datos.size()/2 + 1, this.datos.size() - 1); //n*log (n)
    ArrayList<Jugador> salida = new ArrayList<Jugador>();
    for (int i = 0; i < 10; i++) { //10
        salida.add(this.datos.get(i));
    }
    for (int j = this.datos.size()/2 + 1; j < this.datos.size()/2 + 11; j++) { //10
        salida.add(this.datos.get(j));
    }
    DyV ObjetoAux = new DyV(salida);
    ObjetoAux.mergesort(); //20
    return ObjetoAux.datos;

}
```

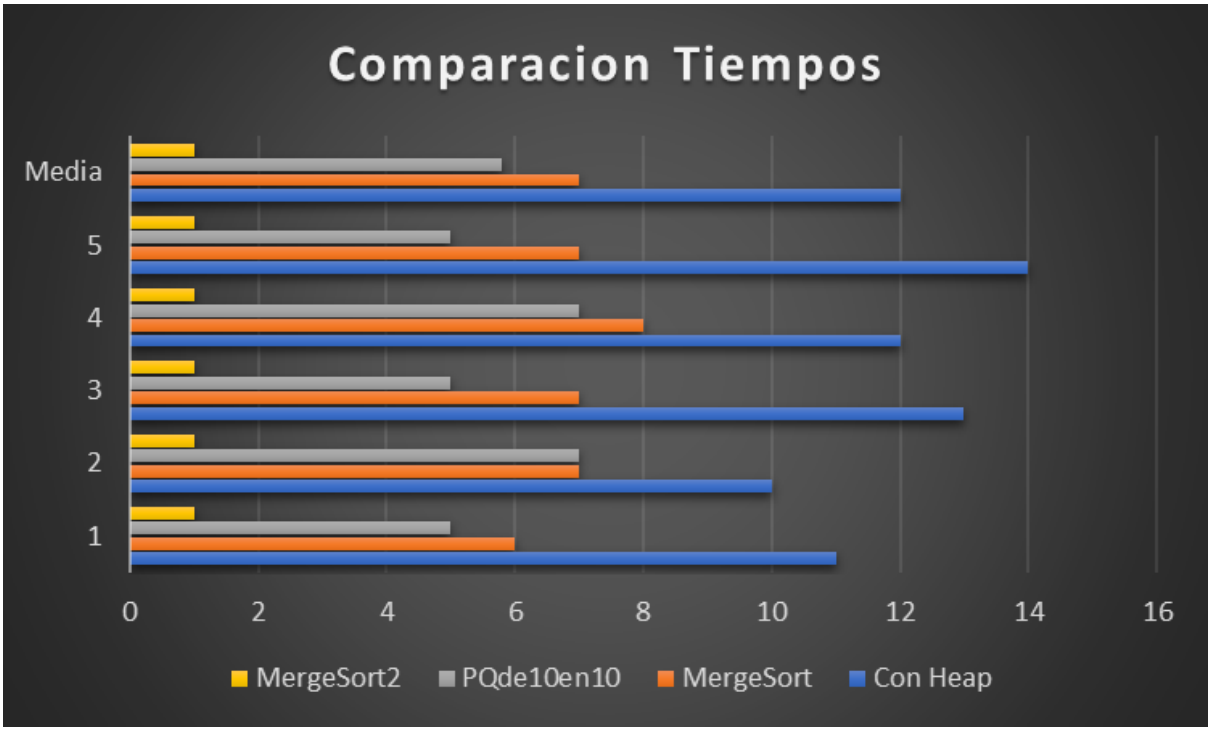
Como podemos ver hace uso del anterior algoritmo de MergeSort , la diferencia es que en este caso hemos dividido el array en 2 , acabando con nuestro array principal ordenado en 2 partes por así decirlo, ordenado desde 0 a n/2 y de n/2 + 1 a n.
Una vez tenemos esto hemos creado otro Arraylist y le hemos introducido los 10 mejores jugadores de cada uno de nuestros bloques imaginarios. Y una vez más usamos

RESULTADOS TEMPORALES

Tabla de Tiempos

Nº medida	Con Heap	MergeSort	PQde10en10	MergeSort 2
1	11 ms	6 ms	5 ms	1 ms
2	10 ms	7 ms	7 ms	1 ms
3	13 ms	7 ms	5 ms	1 ms
4	12 ms	8 ms	7 ms	1 ms
5	14 ms	7 ms	5 ms	1 ms
Media	12 ms	7 ms	5,8 ms	1 ms

Gráfica con todos los tiempos



Gráfica de Medias