Práctica 1 EDA II:
Divide y Vencerás

David Makovetskiy Makovetska

Ana María Giménez Giménez

Lucas Soto Sierra

ÍNDICE

Introducción

En esta práctica se va a desarrollar dos algoritmos, uno basado en la técnica de Divide y Vencerás para dar solución a un problema relacionado con un archivo que contiene los datos y estadísticas de todos los jugadores de la NBA. Se trata de encontrar el top 10 de los jugadores con mejor puntuación (score). Este número se basa en una relación entre el porcentaje de acierto y los puntos totales marcados por el jugador en un año.

De dichos algoritmos se realizará un análisis acerca de la solución obtenida y se realizará un análisis de la eficiencia, tanto desde el punto de vista teórico como práctico.

Modelo de clases y estructura de datos

La estructura de datos elegida, con la que se ha trabajado a lo largo del desarrollo del software, ha sido ArrayList. Se ha seleccionado esta estructura, debido a que su uso es muy sencillo y en este caso, no hemos tenido que realizar ordenaciones muy complejas o no se ha tenido que mantener un orden, desde el instante en el que se empiezan a almacenar los datos. Simplemente se ha realizado una sola operación de ordenación, que se realiza tras la ejecución del algoritmo elegido y que se encarga de ordenar la lista de resultados, ordenando por la variable score. Por tanto, por lo simple que es la estructura y la facilidad a la hora de utilizarla se decidió que era la adecuada para almacenar los datos.

Clase Player

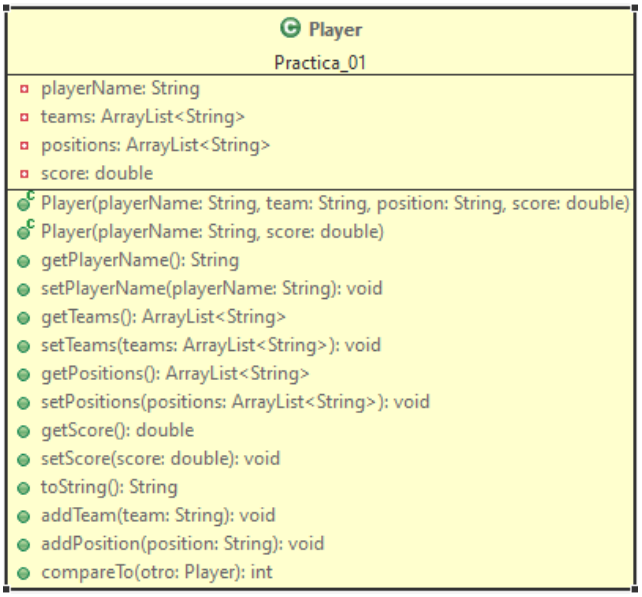


Figura 1. Modelo UML clase Player

Clase Player: Representa los jugadores a los que clasificamos, esta clase posee lo siguiente:

- playerName: String → Nombre del jugador.
- teams: ArrayList<String> → Indica los equipos en los que ha jugado el jugador.
- positions: ArrayList<String> → Indica las posiciones que ha jugado el jugador.
- score: int → Puntuación total que mete un jugador en una temporada con la que se comparan los jugadores.
- Constructor Player (String playerName, String team, String position, int score) → constructor que recibe por parámetros el nombre del jugador, el equipo, la posición en la que juega, y la puntuación.
- Geteres y seteres correspondientes.
- public String toString() → el toString que devuelve el nombre del jugador, el equipo, la posición en la que juega, y la puntuación.
- public int compareTo(Player otro) → compara dos jugadores en base al score.

Para la resolución de este problema hemos modificado la clase Player que se nos otorga en los archivos correspondientes de la práctica.

Hemos añadido el método compareTo y lo hemos modificado para que ordene las puntuaciones de forma descendente.

```
public int compareTo(Player otro) {
    int comp = -Double.compare(this.getScore(), otro.getScore());
    return comp;
}
```

Figura 2. Método compareTo clase Player

También hemos añadido dos métodos (addPosition y addTeam) que vamos a utilizar en el método cargar archivo para cargar las posiciones y los equipos de cada jugador.

Clase Jugadores

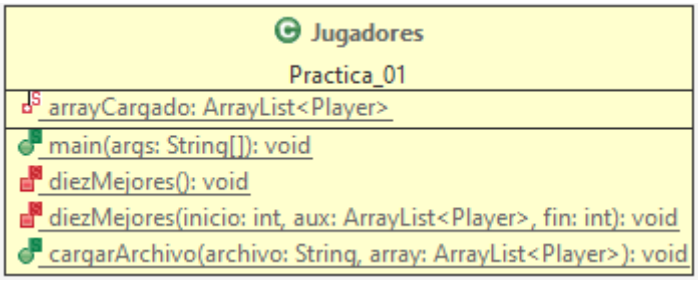


Figura 3. Modelo UML clase Jugadores

Se encarga de cargar el archivo “NbaStats.txt” y guardarlo en un ArrayList<Player> (método cargarArchivo (String archivo, ArrayList<Player> array)). Los dos métodos que hemos creado (Método mergeSort y Método mergeArray) se encargan de coger ese ArrayList y ordenarlo de mayor a menor en base al score y mostrarlo por pantalla. Se ha hecho la explicación en profundidad de cada método más adelante.

Recepción y almacenamiento de datos

Como hemos desarrollado en el apartado anterior, nuestro software se basa en el uso de ArrayList para almacenar los datos, que se pueden generar desde un archivo de texto.

El requisito que nos piden es cargar los jugadores con su puntuación correspondiente. En el archivo que nos pasan pueden aparecer más de una vez un jugador, por tanto, tenemos que hacer la media de sus puntuaciones tantas veces como el jugador aparezca.

Para ello empezamos haciendo uso de una estructura try-catch, la cual utilizamos para comprobar que se ha podido crear el archivo, con ese nombre, y aplicarle el escáner.

```
public static void cargarArchivo(String archivo, ArrayList<Player> array) {
    Scanner sc;
    try {
        sc = new Scanner(new File(archivo));
        sc.nextLine();
        while(sc.hasNext()) {
            String linea = sc.nextLine();
            String items[];
            if(linea.isEmpty()) continue;
            items = linea.split(";");

            double ptos, fg;
            ptos = Double.parseDouble(items[8]);
            fg = Double.parseDouble(items[7].isEmpty()?"0":items[7].replace(",","."));
            fg = fg/100;
            int score = (int)(ptos * fg);

            if(array.isEmpty()) {
                array.add(new Player(items[2], items[6], items[4], score));
                continue;
            }
            if(items[2].equals(array.get(array.size()-1).getPlayerName())) {
                Player aux = array.get(array.size()-1);
                aux.addPosition(items[4]);
                aux.addTeam(items[6]);
                score = (int)((score + aux.getScore()) / 2);
                aux.setScore(score);
            } else {
                array.add(new Player(items[2], items[6], items[4], score));
            }
        }
        sc.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Figura 4. Método cargarArchivo

Como podemos observar en la imagen, el método se divide principalmente en dos if. El primer if se utiliza para obtener la variable score del jugador tomando los datos de los puntos totales y

el porcentaje de acierto, datos que se obtienen con el escáner del archivo. El segundo if se encarga de comprobar si la línea que se ha leído el nombre de ese jugador es igual a uno ya existente en nuestro ArrayList, en caso afirmativo no se vuelve a guardar en el array, sino que se actualizan el resto de variables del jugador (la score, los equipos y las posiciones en las que juega). Por último, para almacenar los datos se ha utilizado un ArrayList de tipo Player.

Explicación práctica y teórica de los métodos

Tenemos dos algoritmos: un algoritmo recursivo y un algoritmo que se llama desde el primer método.

Método mergeSort

Figura 5. Partes del algoritmo recursivo

A este método le pasamos por parámetro un ArrayList, el cual lo obtenemos tras la carga de datos. Por tanto, contiene los jugadores con sus correspondientes puntuaciones.

Al inicio del método declaramos un ArrayList llamado sorted que usaremos más adelante para ir metiendo los jugadores.

El método empieza comprobando que el tamaño de list sea igual a 1. Si es cierta la condición, el ArrayList sorted va a ser igual al que nos pasan por parámetro. Esto es así, porque este método se encarga de ir descomponiendo un array en dos subarrays. Si nos pasan por parámetro un array cuyo tamaño es igual a 1, ya estaría descompuesto y podríamos terminar la ejecución. Esta parte del algoritmo equivale al caso base.

En el caso de que no se cumpla la condición del tamaño igual a 1, procederíamos con la parte que equivale a la descomposición.

Declaramos dos arrays a los cuales les iremos añadiendo los jugadores de list en función del contador del bucle que se esté ejecutando.

Una vez que han sido añadidos los jugadores, vamos a ir haciendo llamadas recursivas al mismo método o al método mergeArray.

Para poder entender mejor el método vamos a guiar la explicación con un ejemplo:

Supongamos que, tras el cargar archivo, nos queda un ArrayList con las siguientes puntuaciones:

9	2	7	1	3	6	8	4	5	15	20	18

Llamamos al método mergeSort pasándole por parámetro este array.

Empezaría la ejecución del método con el caso base comprobando que el tamaño del array sea igual a 1, como en este caso no se cumple esta condición, pasamos a la parte de descomposición.

En esta parte, empezamos calculando la posición en la que se encuentra la mitad del array y declaramos dos nuevos ArrayList llamados left y right. En estos arrays iremos volcando los jugadores.

En el primer bucle del método, vamos a ir volcando en el array left los jugadores que se encuentren en list desde 0 hasta mitad.

En el segundo bucle haremos lo mismo, salvo que empezaremos desde mitad + 1 hasta el final de list.

Finalmente, left y right quedarían así:

9	2	7	1	3	6	8	4	5	15	20	18
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}

9	2	7	1	3	6

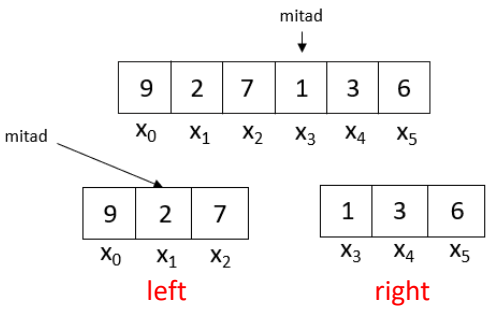
left

8	4	5	15	20	18

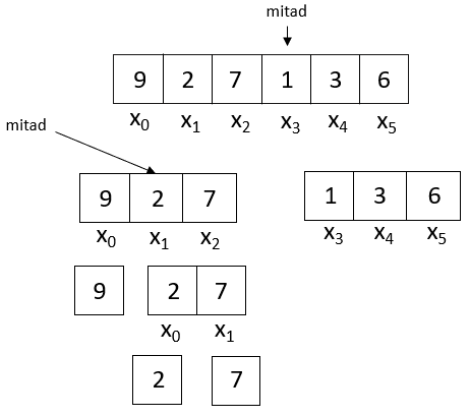
right

El método seguiría su ejecución haciendo la primera llamada recursiva. El resultado de llamar a este método lo vamos a guardar en left.

La ejecución sería igual que la anteriormente explicada: comprobaría que el tamaño del arrayList que le hemos pasado por parámetro, en este caso left, sea 1. Como no lo es, empieza a descomponer el array en 2 subarrays. Quedando de nuevo un ArrayList left y un ArrayList right.



Esto lo iría haciendo de forma recursiva hasta quedar así:



Al llegar a esta situación, el arrayList que le pasaríamos por parámetro al método recursivo es de tamaño 1, por lo tanto, no pasaría del caso base y terminaría la ejecución.

Finalmente, haría una llamada al método mergeArray.

Método mergeArray

A este método le pasamos por parámetros 2 arrays. Su función es ir ordenando las puntuaciones de cada ArrayList e ir añadiéndolo en un nuevo array llamado merged.

Si seguimos con el ejemplo anterior, le pasaríamos por parámetro los 2 últimos arrayList con las puntuaciones 2 y 7, respectivamente.

Al principio del método, declaramos los contadores que se van a ir moviendo en función del paso en el que nos encontremos.

Accedemos al primer bucle while, el cual va a garantizar que solo se pueda ejecutar el código si contadorIzq y contadorDcha son menores que los tamaños de los arrayList que pasamos por parámetro. En caso de que sean igual o mayores, saltaríamos al siguiente while.

En nuestra situación, siempre que hagamos una llamada al método mergeArray vamos a cumplir esa condición debido a que los contadores están inicializados a 0.

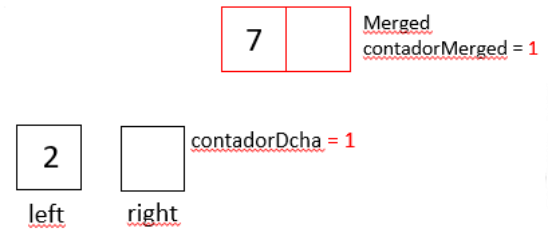
Una vez que hemos cumplido la condición vamos a ir comparando los elementos que tengan los arrays left y right en la posición de los contadores izquierda y derecha, respectivamente.

Si la puntuación que hay en left en la posición del contadorIzq es menor que la puntuación que hay en right en la posición del contadorDcha, añadimos a merged la puntuación del array left e incrementamos su contador. En caso contrario, añadimos la puntuación de del array right en merged e incrementamos su contador.

Independientemente, de qué puntuación añadamos, si la de left o la de right, siempre vamos a incrementar el contador de merged. Esto es así porque en ambas situaciones añadimos contenido al array.

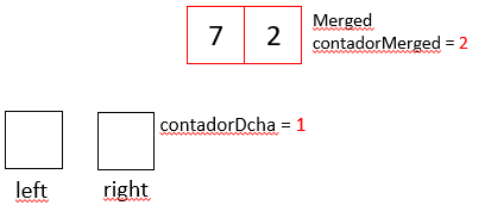
Cuando comparamos las puntuaciones que están contenidas en los ArrayList, lo hacemos de forma inversa. Es decir, en vez de ir comparando los elementos para que queden ordenados de forma ascendente, vamos a ordenarlos para que queden ordenados descendentemente. Para ello hemos hecho modificaciones en el compareTo de la Clase Player.

En nuestro ejemplo, teníamos dos arrays que contenían una puntación de 2 puntos y otra de 7, respectivamente. El resultado de hacer el compareTo entre los dos elementos sería positivo y pasaríamos a la sección del *else*, donde agregaríamos el 7 a merged.



Posteriormente, volvemos a comprobar que contadorIzq y contadorDcha sean menores que los tamaños de left y right. Ahora el valor del contadorDcha se ha incrementado en 1, y como el tamaño del arrayList right también es 1, no se cumpliría la condición y tendríamos que pasar al siguiente while.

En los dos while siguientes comprobamos que los contadores izquierda y derecha sean menores que los tamaños de left y right. Como previamente hemos actualizado el valor del contador de la derecha, no podríamos ejecutar el segundo while. A su vez, el contador de la izquierda permanece con el mismo valor. Por tanto, añadimos a merged la puntuación 2 e incrementamos contadorIzq y contadorMerged.



Finalmente, el método devolvería el arrayList merged con los valores ordenados de forma descendente. Esto lo va a hacer de forma recursiva con todos los ArrayList que le pasemos por parámetro.

Análisis de eficiencia del algoritmo

Para calcular el tiempo de ejecución del algoritmo vamos a hallar la ecuación de recurrencia. Para ello tenemos que identificar los tiempos de ejecución de cada parte del algoritmo.

```java
public static ArrayList< Player > mergeSort(ArrayList<Player> list) {
  ArrayList<Player> sorted;
  if (list.size() == 1){
      sorted = list;                                   // O(1)
  }else {
      int mitad = list.size()/2;                       // O(1)

      ArrayList< Player > left = new ArrayList< Player >();
      ArrayList< Player > right = new ArrayList< Player >();

      for ( int x = 0; x < mitad; x++) {               // O(N)
          left.add(list.get(x));                       // O(1)
      }
      for ( int x = mitad; x < list.size(); x++) {     // O(N)
          right.add(list.get(x));                      // O(1)
      }
      left = mergeSort(left);
      right = mergeSort(right);
      sorted = mergeArray(left,right);
  }
  return sorted;
}
```

Figura 6. Análisis tiempo de ejecución método mergeSort

```java
private static ArrayList<Player> mergeArray(ArrayList<Player> left, ArrayList<Player> right) {
    ArrayList<Player> merged = new ArrayList<Player>();

    int contadorMerged = 0;
    int contadorIzq = 0;
    int contadorDcha = 0;

    while (contadorIzq < left.size() && contadorDcha < right.size()) {   // SE EJECUTA N VECES
        if ((left.get(contadorIzq)).compareTo(right.get(contadorDcha)) < 0) {   // O(N)
            merged.add(left.get(contadorIzq));       // O(1)
            contadorIzq++;
        } else {
            merged.add(right.get(contadorDcha));     // O(1)
            contadorDcha++;
        }
        contadorMerged++;
    }

    while (contadorIzq < left.size()){                // O(N)
        merged.add(left.get(contadorIzq));           // O(1)
        contadorIzq++;
        contadorMerged++;
    }

    while (contadorDcha < right.size()) {             // O(N)
        merged.add(right.get(contadorDcha));         // O(1)
        contadorDcha++;
        contadorMerged++;
    }
    return merged;
}
```

ECUACIÓN DE RECURRENCIA:

- Resolución mediante división

T (n) = c1*n^{k1} **Para 0 ≤ n < b**

T(n) = at(n/b) + c2*n^{k2} **En n ≥ b**

T(n) = O(1) → n^0 → k1 = 0

T(n) 2t(n/2) + O(n) → O(n) == n^{k2} → k2 = 1

K = máximo (k1, k2) = máximo (0, 1) = 1

a = 2, b = 2, k = 1

¿a < b^k, a = b^k, a > b^k?

a = b^k → 2 = 2^1

Ecuación de recurrencia: O (n^k logn) → O (nlogn)

Propuesta de mejora del algoritmo

Hemos hallado una mejora para nuestro código en el método mergeArray. El funcionamiento del método es el mismo, salvo que le añadimos una condición al final del método que comprueba si el tamaño del array merged es mayor que 10. En caso de que lo sea llamamos al método solución.

En dicho método añadimos a un array llamado *solución* las 10 primeras posiciones.

Esto nos ahorra tiempo debido a que nos garantiza que cuando el array supere el tamaño de 10, termine el método. Por tanto, nos ahorrará tiempo ya que nos ahorramos hacer comparaciones innecesarias.

Figura 7. Método mejorado

Análisis de resultados y tiempos de ejecución

Para comprobar los tiempos de ejecución del algoritmo hemos creado unos archivos de texto que contienen jugadores con puntuaciones aleatorias.

Tiempos de ejecución método sin mejora

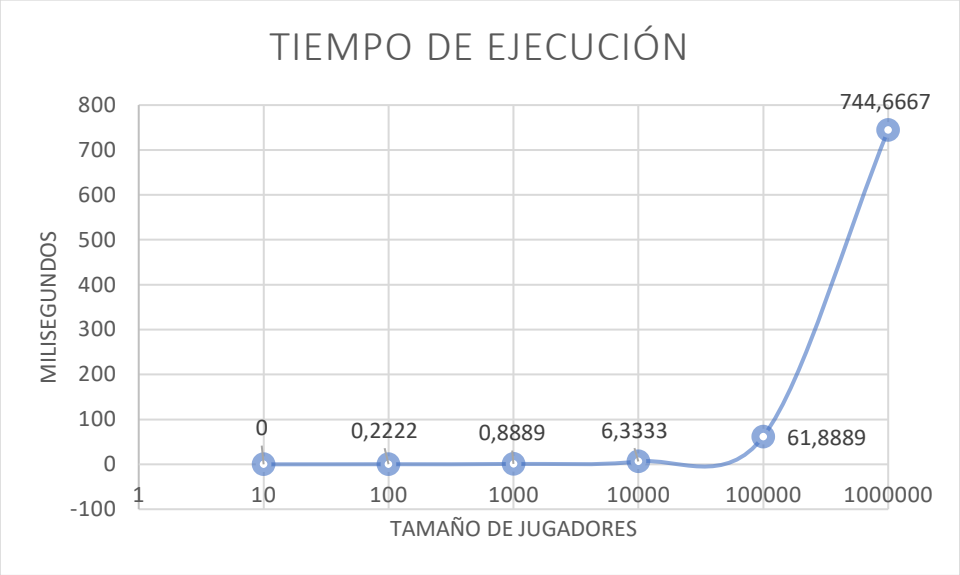


Figura 8. Gráfico con los tiempos de ejecución algoritmo base

Como podemos observar, el algoritmo es eficiente. Para un archivo con un millón de jugadores el algoritmo no tarda ni un segundo.

Tiempos de ejecución método mejorado

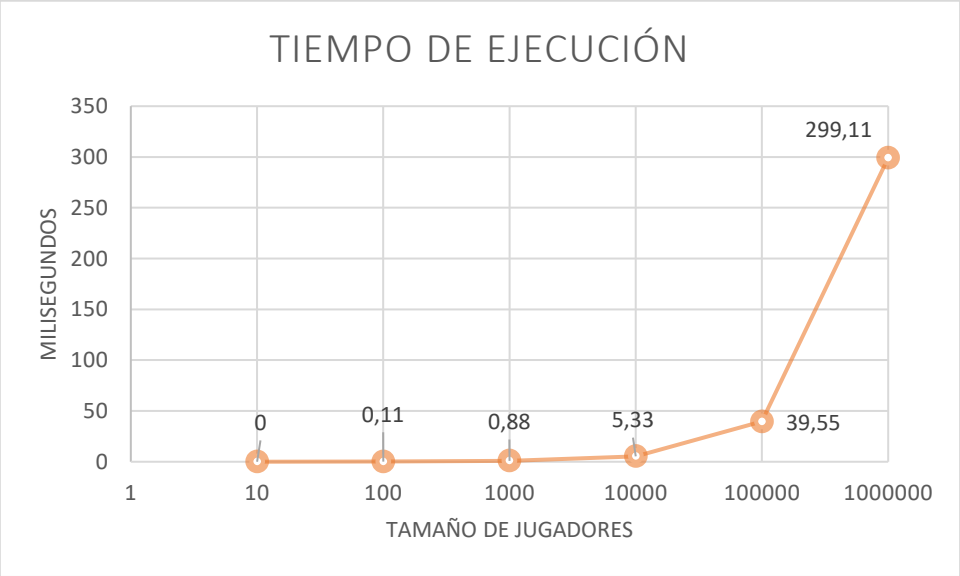


Figura 9. Gráfico con los tiempos de ejecución algoritmo mejorado

Comparación tiempos de ejecución

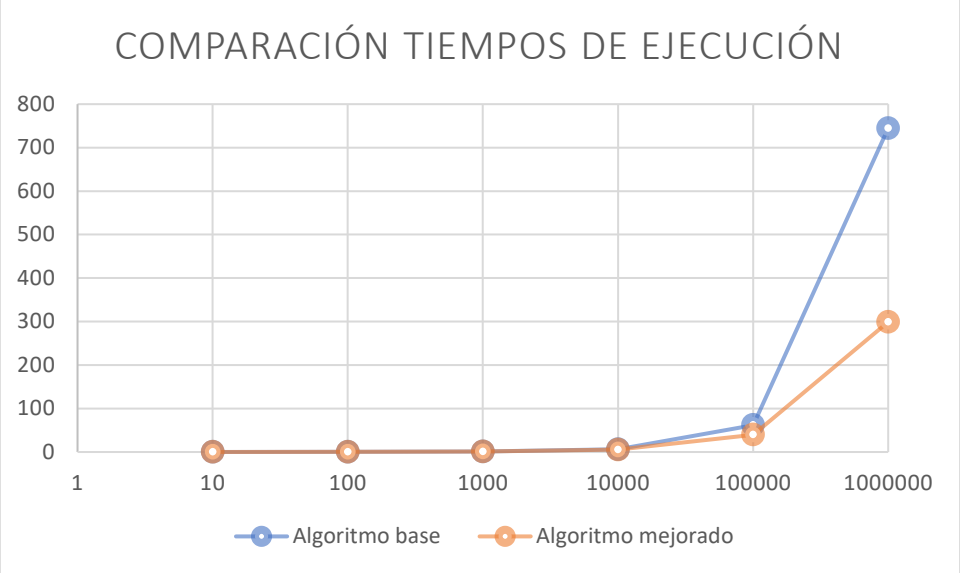


Figura 10.Gráfico con la comparación de los algoritmos

Como podemos comprobar en la gráfica, el algoritmo mejorado es bastante más rápido comparado con el método original. Por lo tanto, la mejora es efectiva.

Conclusión

En cuanto a las conclusiones y la valoración personal de esta práctica, se puede decir que se ha llevado a cabo una práctica muy completa en comparación a lo que estamos acostumbrados a realizar. Se ha tenido que elaborar el programa desde 0, con la información justa y necesaria, y se ha tenido que construir poco a poco, realizando diversas pruebas debido a la dimensión y la variedad que ofrece el programa.

Desde nuestro punto de vista, podemos decir que es una práctica que requiere trabajo y saber manejar adecuadamente los conocimientos del tema, ya que no es solo la realización del programa, sino también la elaboración de una memoria en la que hay que explicar detalladamente todo el proceso.