Universidad de San Carlos de Guatemala.

Centro Universitario de Occidente.

División de Ciencias de la Ingeniería.

Laboratorio Estructura de Datos.

Ing. Oliver Sierra.

Primer Semestre 2022.

# “DOCUMENTACION TÉCNICA”

Diego José Maldonado Monterroso.

Carné: 201931811.

Quetzaltenango, Guatemala.

07 de marzo de 2022.

# METODOS USADOS EN EL PROGRAMA

## Lectura de Archivos.

```java
public class LectorDeArchivos {

    /**
     * Metodo para poder leer un archivo de texto
     * @param archivo   archivo previamente seleccionado
     * @param text  area de texto a donde agregamos los datos leidos de este archivo
     * @throws FileNotFoundException
     * @throws IOException
     */
    public void leerFichero(File archivo, JTextArea text) throws FileNotFoundException, IOException {
        FileReader fr = new FileReader(archivo);
        BufferedReader br = new BufferedReader(fr);

        String linea;

        while((linea = br.readLine()) != null){
            text.append(linea+"\n");
        }
        fr.close();
    }

}
```

Método de complejidad O(n) ya que va realizarse n veces, o sea, la cantidad de n líneas que contenga ese archivo.

```java
public void escribirArchivo(JFrame framePapa,Apuesta[] apuestas){

    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");


    File fichero = null;
    PrintWriter pw = null;

        try {
            fichero = new File("Files/ApuestaExportada.txt");
            pw = new PrintWriter(fichero);

            pw.println("Archivo de Apuestas Exportado con fecha: "+dtf.format(LocalDateTime.now()));

            for(Apuesta apuesta: apuestas){
                pw.println(apuesta.toString());
            }

            JOptionPane.showMessageDialog(framePapa, "Archivo generado con exito");
        }
        catch (Exception ex) {
                JOptionPane.showMessageDialog(framePapa, "Error al escribir el archivo");
                try{
                    if(null != fichero){
                        pw.close();
                    }
                }
                catch(Exception e2){
                    JOptionPane.showMessageDialog(framePapa, "Ha ocurrido un error mientras se exportaba el archivo");
                }
```

Método de complejidad O(n) ya que por cada objeto de tipo -apuesta- en el arreglo -apuestas- se escribirá una línea de código. Si yo tengo n apuestas, se repetirá n cantidad de veces.

**Manejadores del programa.**

**Clase: Constructor de Apuestas.**

```java
/**
 * Metodo estatico para construir una apuesta en el cual casteamos los campos enviados
 * @param campos    arreglo de campos pertenecientes a una linea
 * @return
 */
public static Apuesta  construirApuesta(String[] campos){
    Apuesta apuesta = null;

    apuesta = new Apuesta(campos[0], Integer.valueOf(campos[1]), arregloToInt(campos));

    return apuesta;
}
```

Método de complejidad O(1), ya que solo hacemos una instancia de un objeto de tipo Apuesta el cual retornamos.

```java
/**
 * Metodo para castear un arreglo de tipo string a uno de tipo int
 * @param campos
 * @return
 */
public static int[]  arregloToInt(String[] campos){

    int[] intArray = new int[campos.length-2];

    for (int i = 2; i < campos.length; i++) {
        try {
            intArray[i-2] = Integer.parseInt(campos[i]);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "No ha sido posible convertir de string a int: " + e.getMessage());
        }
    }
    return intArray;
}
```

Método de complejidad O(n), ya que mientras yo tenga n campos se realizará el casteo de n elementos.

**Clase: Ingreso de Apuestas. Servicio Critico.**

```java
/**
 * Metodo para ingresar las apuestas a nuestro sistema
 * @param text  tomamos el texto del textArea el cual contiene todas las apuestas
 * @return
 */
public  Apuesta[] ingresarApuestas(JTextArea text) {

    Verificador verificador = new Verificador();

    Apuesta[] apuestas = new Apuesta[text.getLineCount()];


    String[] lineas = text.getText().split("\n");


    for(int i=0; i<lineas.length; i++){
        String[] campos = separarLinea(lineas[i]);
        Apuesta apuesta = ConstructorDeApuestas.construirApuesta(campos);
        apuesta.setIsCorrecta(verificador.verificarArreglo(apuesta.getPosicionesCaballos()));
        apuestas[i] = apuesta;
    }

    return apuestas;
}
```

Método de complejidad O(n), ya que es una iteración que se repetirá n veces, según la cantidad de líneas que yo tenga, ya que cada línea contendrá una apuesta.

```java
/**
 * Metodo para separar en campos cada linea perteneciente al JTextArea
 * @param linea
 * @return
 */
public static String[] separarLinea(String linea){
    String[] campos = linea.split(",");
    return campos;
}
```

Método de complejidad O(1) ya que solamente realizamos una sola asignación para obtener los campos de una línea y almacenarlos en un arreglo de tipo String.

**Clase: Resultados. Servicio Critico.**

```java
/**
 * Metodo con el cual creamos un resultado
 * @param texto texto tomado de un textArea donde ingresamos las posiciones de los caballos
 * @return
 */
public int[] crearResultado(String texto){
    String[] campos = texto.split(",");

    int[] resultado = new int[campos.length];


    for(int i=0; i<campos.length; i++){
        resultado[i] = Integer.valueOf(campos[i]);
    }

    return resultado;
}
```

Método con complejidad O(n), que nos sirve para castear cada campo obtenido al ingresar un resultado. La iteración se repetirá n veces, o sea, la cantidad n de campos que tenga.

```java
/**
 * Con este metodo enviamos los arreglos de posiciones de cada apuesta para calcular el puntaje del apostador
 * @param apuestas
 * @param resultados
 */
public void calculoResultados(Apuesta[] apuestas, int[] resultados){
    for(Apuesta a: apuestas){
        if(a.isIsCorrecta()){
            a.setPuntaje(otorgarPuntaje(a.getPosicionesCaballos(), resultados));
        }
    }
}
```

Método de complejidad O(n), ya que se repetirá el numero n de apuestas que se ingresen al sistema.

```java
/**
 * Metodo para otorgar puntaje a cada apostador dependiendo de su arreglo de posiciones de caballos
 * @param posicionesCaballos
 * @param resultados
 * @return
 */
public static int otorgarPuntaje(int[] posicionesCaballos, int[] resultados){
   int puntos=10;
   int totalPuntos=0;
   for(int i=0; i<10; i++){
      if(posicionesCaballos[i]==resultados[i]){
          totalPuntos = totalPuntos + (puntos-i);
      }
   }

   return totalPuntos;
}
```

Método de complejidad O(k), ya que las iteraciones se repetirán solamente 10 veces.

**Clase: Verificador. Servicio Critico.**

```java
/**
 * Metodo para verificar que un arreglo no contenga elementos repetidos
 * @param caballos
 * @return
 */
public Boolean verificarArreglo(int[] caballos){

    boolean correcta=true;

    for(int i=0; i<caballos.length-1; i++){
        if(caballos[i]==caballos[i+1]){
            correcta = false;
        }
    }
    return correcta;
}
```

Método de complejidad O(n), ya que realizaremos las iteraciones y comparaciones n veces, o sea, la cantidad de caballos que tengamos. Pero como ya sabemos que solo hay 10 caballos, podemos reducir la complejidad este método a un O(k), donde k = 10.

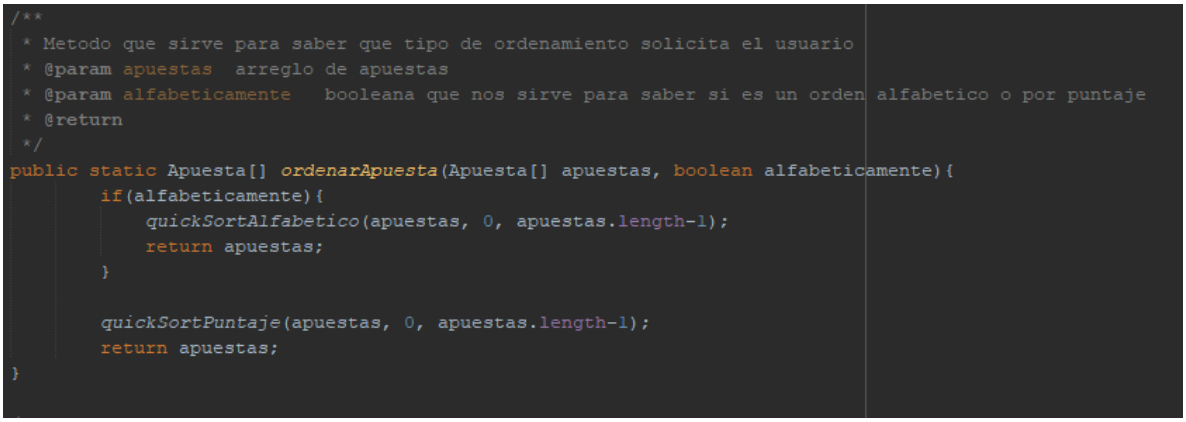
**Clase: Apuesta.**

```
public String getNombreApostador() {
    return nombreApostador;
}

public void setNombreApostador(String nombreApostador) {
    this.nombreApostador = nombreApostador;
}

public int getMonto() {
    return monto;
}

public void setMonto(int monto) {
    this.monto = monto;
}

public int[] getPosicionesCaballos() {
    return posicionesCaballos;
}

public void setPosicionesCaballos(int[] posicionesCaballos) {
    this.posicionesCaballos = posicionesCaballos;
}

public boolean isIsCorrecta() {
    return isCorrecta;
}
```

Getters y Setters del sistema, los cuales tienen una complejidad de O(1).

**Clase: Ordenamiento. Servicio Critico.**

Se eligió el método QuickSort, ya que me puse a investigar bastantes experimentos realizados en universidades que cuentan con la carrear de informática, y en sus resultados concluían que el método QuickSort era un algoritmo muy eficiente donde la complejidad del algoritmo la definía el pivote, dependiendo de que manera divide el arreglo original. Lo ideal es que se divida en partes iguales, en el peor de los casos lo dividirá en un arreglo muy grande y otro muy pequeño.



```java
/**
 * Metodo que sirve para saber que tipo de ordenamiento solicita el usuario
 * @param apuestas  arreglo de apuestas
 * @param alfabeticamente   booleana que nos sirve para saber si es un orden alfabetico o por puntaje
 * @return
 */
public static Apuesta[] ordenarApuesta(Apuesta[] apuestas, boolean alfabeticamente){
        if(alfabeticamente){
            quickSortAlfabetico(apuestas, 0, apuestas.length-1);
            return apuestas;
        }

        quickSortPuntaje(apuestas, 0, apuestas.length-1);
        return apuestas;
}
```

Método de complejidad O(1), donde simplemente realizamos una bifurcación y eso nos lleva a llamar a otros métodos, para luego retornar un arreglo de apuestas.

```java
public static void quickSortAlfabetico(Apuesta[] array, int inicio, int fin){

    int i = inicio;

    int j = fin;

    if (j - i >= 1)
    {

        String pivote = array[i].getNombreApostador();

        while (j > i)
        {
            while (array[i].getNombreApostador().compareTo(pivote) <= 0 && i < fin && j > i){
                i++;
            }
            while (array[j].getNombreApostador().compareTo(pivote) >= 0 && j > inicio && j >= i){
                j--;
            }
            if (j > i)
                cambio(array, i, j);
        }
        cambio(array, inicio, j);

        quickSortAlfabetico(array, inicio, j - 1);

        quickSortAlfabetico(array, j + 1, fin);
    }
}
```

Método de ordenamiento alfabético de complejidad O(n log n). En el mejor de los casos el pivote separa en dos arreglos de igual tamaño el arreglo original para reducir la complejidad del algoritmo. En el peor es un $O(n^2)$.

```java
public static void quickSortPuntaje(Apuesta[] array, int inicio, int fin){

    int i = inicio;

    int j = fin;

    if (j - i >= 1)
    {
        int pivote = array[i].getPuntaje();

        while (j > i)
        {
            while (array[i].getPuntaje() >= pivote && i < fin && j > i){
                i++;
            }
            while (array[j].getPuntaje() <= pivote && j > inicio && j >= i){
                j--;
            }
            if (j > i)
                cambio(array, i, j);
        }
        cambio(array, inicio, j);

        quickSortPuntaje(array, inicio, j - 1);

        quickSortPuntaje(array, j + 1, fin);
    }
}
```

Método de ordenamiento por puntaje de complejidad O(n log n). En el mejor de los casos el pivote separa en dos arreglos de igual tamaño el arreglo original para reducir la complejidad del algoritmo. En el peor es un $O(n^2)$.

```java
/**
 * Metodo para agilizar el cambio de elementos dentro del arreglo
 * @param array arreglo de apuestas
 * @param i iterador i
 * @param j iterador j
 */
private static void cambio(Apuesta[] array, int i, int j){
    Apuesta temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

Método que nos ayuda en los métodos de ordenamiento descritos anteriormente. Es un algoritmo de complejidad O(1), ya que solamente realizamos un cambio de posiciones.