UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
CENTRO UNIVERSITARIO DE OCCIDENTE
DIVISIÒN DE CIENCIAS DE LA INGENIERÍA

MANUAL TECNICO DE PRÁCTICA 1
ESTRUCTURA DE DATOS I

PRESENTADO POR:

DANIEL EDUARDO BAUTISTA FUENTES 201930588

DOCENTE
ING. OLIVER SIERRA
AUXILIAR
…

QUETZALTENANGO - QUETZALTENANGO - GUATEMALA
06/03/2022

Complejidad de servicios críticos

Calculo de complejidad de servicios críticos del programa para simular carreras de caballos.

Información sobre la aplicación:

- Lenguaje de programación Java SWING
- Archivo generado en .jar
- Versión del manual: HorseRaces-hw0.9.4.2

Algunos métodos fueron recursivos por lo que se utilizó otro método distinto de Big O, pero para los modelos no recursivos se hizo uso de la notación Big O para el cálculo de complejidad.

Ingreso de apuestas antes del inicio de la carrera

Para esto, hubo métodos previos, como el de carga de datos a partir de un archivo de texto. Si se quiere ver el código utilizado para ello, se puede presionar [aquí](#).

Insertar apuesta a la lista

Cómo estamos utilizando una lista doblemente enlazada y en cada momento conocemos la posición del último nodo, podemos hacer un O(1) para la inserción de nueva información a la lista.

```
public void addAtHead(T data) {
        if (isEmpty()) { // 1st step
            head = tail = new Node<T>(data); // 2 steps, declaration and
initializations
        } else {
            head = new Node<T>(data, null, this.head); // 1 step,
delcaration and initizlization
            head.getPrevious().setNext(head); // 1 step, set previous
node's next to new node
        }
        this.size++;
    }
```

Podemos resumir la complejidad con la siguiente ecuacion

$O(n) = 1 + 1 + 1 + 1 + 1$

$O(n) = 1$

INSERTAR NOS TOMA O(1)

Verificación de apuestas

Para calcular la complejidad de este, vemos que llamamos a métodos resetSteps() que son O(1) al igual que this.time ... los tomamos como un O(1), sin embargo necesitamos calcular la complejidad del método validateSub()

Método principal para verificar apuestas

```
   /**
    * Validates the bets, if the bet is valid, the info of the bet is
stored in
    * the bet object is changed to false if any of the horses are
repeated,
    * otherwise, the bet is valid
    *
    * @param bets the bets to analyze
    * @return
    */
   public ReportStatus validateBets(NodeList<Bet> bets) {
       // actions
       resetSteps(); // reset vals for report
       // varibales
       validateSub(bets.getTail());
       // create a promedium
       this.time = this.time / bets.getSize();
       this.steps = this.steps / bets.getSize();
       this.realSteps = this.realSteps / bets.getSize();
       return this.error ? ReportStatus.FAILURE
               : (this.noValids > 0 ? ReportStatus.SOME_BETS_INVALID :
ReportStatus.SUCCESS);
   }
```

$$O(n)_t = O(1) + O(n)_{s1} + 1 + 1 + 1 + 1$$

$$O(n)_T = 1 + O(n)_{s1} + 1$$

$$O(n)_T = 1 + O(n)_{s1}$$

Ahora debemos calcular el $O(n)_{s1}$ para el método validateSub

ValidateSub

```
/**
 * Submethdo to validate all bets 1 by 1 recursively
 *
 * @param next the node to validate
 */
private void validateSub(Node<Bet> next) {
    if (next == null) {
        // end
    } else {
        int status = validate(next);
        // if the bet is valid, the info of the bet is stored in the
bet
        if (status == 1) {
            this.noValids++;
        } else if (status == -1) {
            this.error = true;
        }
        // step section -> add the steps into the method validate
        calcMostLessSteps();
        addSteps(this.stepsByLoop, this.realStepsByLoop);
        // recursivity
        validateSub(next.getNext());
    }
}
```

Puesto que es recursivo tenemos que calcular T(n). Pero primero calcularemos la complejidad del metodo dentro de else{}

$O(n) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + O(1) + O(1)$

$O(n) = 1 + O(1) + O(1) + O(n)_{s2}$

$O(n) = 1 + 1 + 1 + O(n)_{s2}$

$O(n)_T = 1 + O(n)_{s2}$

Si excluimos el llamado recursivo, sabemos que el valor de complejidad dentro de la sentencia else es un $O(1) \ + \ O(n)_{s2}$.

Por lo que podemos empezar a calcular T(n)

Talla	n
Base case(n=0)	T(n)=1
General case(n>0)	T(n)=1 + T(n-1)

Recurrency ecc:

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 + T(n-1) & n > 0 \end{cases}$$

-> recurrency deploy

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 + T(n-1) & n > 0 \end{cases}$$

T(n)= 1 + T(n-1)
= 1 + (1 + T(n-2)) = 2 + T(n-2)
= 2 + (1 + T(n-3)) = 3 + T(n-3)
= ...
= i + T(n-i)

at last call n - i = 0 <-> i = n -> T(n) = n + T(0) = n + 1

T(n) = n + 1
T(n) ∈ Q(n)

$$O(n) = O(n)_{s1} = n + 1 + O(n)_{else}$$
$$O(n) = O(n)_{s1} = n + 1 + 1 + O(n)_{s2}$$
$$O(n)_{s1} = n + 1 + O(n)_{s2}$$

Por lo tanto solo nos hace falta calcular la complejidad de validate

Validate

```
    /**
     * This method validate a sibgle node and see if bet is valid
     *
     * @param current the node to analyze
     * @return 0 if no error, 1 if no valid, -1 if error
     */
    private int validate(Node<Bet> current) {
        int valid = 0;
        try {
            // variables
            this.repeatValidator.resetSteps();
            this.timer.run();
            // save valid info
            if (current.getData().getHorses().length != 10 ||
current.getData().getAmount() < 0) {
                current.getData().setValid(false);
                valid = 1;
                addSteps(0, 1);
            } else if (current.getData().getGambler().getName().isEmpty()
                    || current.getData().getGambler().getName().isBlank())
{
                valid = 1;
            } else {

current.getData().setValid(this.repeatValidator.isNumRepeat(current.getDat
a().getHorses()));
                if (!current.getData().isValid()) {
                    valid = 1;
                    addStepLoop(0, 1);
                }
                // add steps from method numrepeat
                addStepLoop(this.repeatValidator.getSteps(),
this.repeatValidator.getRealSteps() + 1);
            }
            // add values to promedium
            this.timer.stopTimer();
            this.time += this.timer.getTotalTime();
        } catch (Exception e) {
```

```
            valid = -1;
        }
        addStepLoop(1, 7);
        return valid;
    }
```

```
/**
```

$O(n)_2 = 1 + 1 + O(n)_{s3} + 1 ... + 1$

$O(n)_2 = 1 + 1 + O(n)_{s3} + 1 ... + 1$

$O(n)_2 = 1 + O(n)_3$

$O(n)_T = 1 + O(n)_{s2}$

$O(n)_T = 1 + + 1 + O(n)_{s3}$

$O(n)_T = 1 + O(n)_{s3}$

En este método, obtendremos siempre un $1 + O(n)_{s3}$, sin embargo, llama a un submetido, por lo que debemos calculator la complejidad del sub metodo <<isNumRepeat>>

IsNumRepeat

```
    /**
     * This method validate if a String is repeated, if the string is
repeated,
     *
     * @param values the sting of values to be validated
     * @return true if there's nos tring repeated, this method is Case
sensitive
     */
    public boolean isNumRepeat(int[] values) {
        intDeclared(values, 0, values[values.length - 1], 0);
        boolean tmp = this.isValid; // save data
        this.isValid = true; // reset value of object
        addRealSteps(1); // if, bool asign, valid asign, return
        return tmp;
    }
```

$O(n)_{s3} = 1 + 1 + 1... + O(n)_{s4}$

$O(n)_{s3} = 1 + O(n)_{s4}$

$O(n)_T = 1 + O(n)_{s3}$

$O(n)_T = 1 + 1 + O(n)_{s4}$

$O(n)_T = 1 + O(n)_{s4}$

Con este método seguimos obteniendo un valor $1 + O(n)_{s4}$ por lo que debemos volver a calculator él $O(n)_{s1}$ del sub metodo <<intDeclared>> y sumarlo a la complejidad total.

```
    /**
     * validate if a String is actually defined, if we have a conj
{A,B,C,D} it
     * works comparing A with B, A with C, A with D, B with C, B with D, C
with
     * D...
     *
     * @param values the values array of ints
     * @param pos the position of the actual value to compare, when call
1st
     * time send 0
```

```java
     * @param toCompare the value to compare, when call the method, sand
the
     * last item on the array
     * @param index the number of main groups changed, when call 1st time,
send
     * 0
     * @return the value compared
     */
   private int intDeclared(int[] values, int pos, int toCompare, int
index) {
       // end of recursivity
       if (index >= values.length - 1) {
           addSteps(1, 1); // if, return
           return values[pos];
       } else {
           try {
               // validate if the actual char to compare is the last in
the array
               if (pos >= values.length - index - 1) {
                   pos = 0; // reset the position
                   index++; // increases the group index analyzed
                   toCompare = values[values.length - index - 1]; // next
value to analyze
                   addRealSteps(1);
               }
           } catch (Exception e) {
               addRealSteps(1); // error management
           }
           if (values[pos] == toCompare && index < values.length - 1) { //
at this point A always is the same as A, so
               // validate index isn't the last
               this.isValid = false;
               addRealSteps(1); // if, set valid
           }

           // recursive call
           addSteps(1, 1); // recursive call
           return intDeclared(values, pos + 1, toCompare, index);
       }
```

```
    }
```

Puesto que es un método recursivo, la complejidad es de n + 1 tras demostrarlo mediante T(n)

Talla: values: n
Base case: T(n) = 1
General case: T(n) = 1 + T(n + k -l - 1) => pos:k index:l values:n
Recurrency ecc:

T(n) = 1 n = 0
1 + T(n + k -l - 1) n > 0

-> recurrency deploy

T(n) = 1 n = 0
1 + T(n + k - l - 1) n > 0

T(n) = 1 + T(n + k - l - 1) n > 0
= 1 + (1 + T(n + k - l - 2)) = 2 + T(n + k - l - 2)
= 2 + (1 + T(n + k - l - 3)) = 3 + T(n + k - l - 3)
= ...
= i + T(n + k - l - i)

-> at last call n + k -l -i = 0 <=> i + l = n + k -> T(n) = n + T(0) = n + 1

T(n) = n + 1
T(n) ∈ Q(n)

? = IS_STRING_DEC_COMPLEX
RESULT = n + 1 + n + 1
RESULT = n
Q(n) = n // EXPECTED COMPLEX

$O(n)_{s4} = n$

Una vez hemos calculator tanto $O(n)_{s1}$ como $O(n)_{s}$ como $O(n)_{T}$, puesto que no existen bucles en $O(n)_{T}$ todos los resultados se suman.

$$O(n)_T = 1 + O(n)_{s4}$$
$$O(n)_T = 1 + n$$
$$O(n)_T = n$$

VALIDAR APUESTAS NOS TOMA N

Cálculo de resultados

Este método al igual que al validar apuestas, nos debe tomar n para todas las apuestas, asi que iremos calculando la complejidad de los sub métodos.

Método principal para calcular puntos

```
    /**
     * This method add points to bet's Gambler based on the result of the
horses
     * positon, if Gambler guesses correctly the 1st horse gains 10 points,
if
     * guesses correctly the 2nd horse wins
     *
     * @param bets            the NodeList of bets to analyze
     * @param horsesPosition the name of the horses sorted by positon from
1st
     *                        to last
     * @return
     */
    public ReportStatus calculate(NodeList<Bet> list, int[] horsePositions)
{
        // variables
        resetSteps();
        calculateSub(list.getTail(), horsePositions);
        this.steps = this.steps / list.getSize();
        this.realSteps = this.realSteps / list.getSize();
        this.time = this.time / list.getSize();
        return this.error ? ReportStatus.FAILURE : ReportStatus.SUCCESS;

    }
```

$$O(n)_T = O(1) + O(n)_{s1} + 1 + 1 + 1 + 1$$
$$O(n)_T = O(1) + O(n)_{s1} + 1$$
$$O(n)_T = 1 + O(n)_{s1} + 1$$
$$O(n)_T = 1 + O(n)_{s1}$$

Ahora debemos calculator la compljedidad de $O(n)_{s1}$

CalculateSub

```
    /**
     * Submethod to calculate the points of the bets
     *
     * @param next            the next bet to analyze
     * @param horsePositions the horses podium
     */
    private void calculateSub(Node<Bet> next, int[] horsePositions) {
        if (next == null) {
            // end
        } else {
            // action
            this.timer.run();
            this.stepLoop++;
            this.realStepLoop += 2;
            try {
                if (next.getData().isValid()) {
                    setPoints(next.getData(), horsePositions, 0);
                }
            } catch (Exception e) {
                this.error = true;
                realStepLoop++;
            }
            this.timer.stopTimer();
            this.time += this.timer.getTotalTime();
            calcMostLessSteps(stepLoop);
            addSteps(stepLoop, realStepLoop);// run timer, if, stop timer,
get time, return
            this.stepLoop = 0; // reset loop vals
            this.realStepLoop = 0;
            calculateSub(next.getNext(), horsePositions); // recursive call
        }
    }
```

Calculamos primero la complejidad dentro de la sentencia else

$$O(n) = 1 + 1 + 1 + 1 + O(n)_{s2} + 1 + 1 + 1 + 1 + 1 + O(1)$$

$$O(n) = 1 + O(n)_{s2} + 1 + O(1)$$

$$O(n) = 1 + O(n)_{s2} + 1$$

$O(n) \; = \; 1 \; + \; O(n)_{s2}$

AHora calculamos el T(n) para el método calculateSub

Talla: values: n

Base case: T(n) = 1

General case: T(n) = 1 + T(n - 1) => possible next Nodes:n

Recurrency ecc:

T(n) = 1 n = 0
1 + T(n - 1) n > 0

-> recurrency deploy

T(n) = 1 n = 0
1 + T(n - 1) n > 0

T(n) = 1 + T(n - 1) n > 0
= 1 + (1 + T(n - 2)) = 2 + T(n - 2)
= 2 + (1 + T(n - 3)) = 3 + T(n - 3)
= ...
= i + T(n - i)

-> at last call n - i = 0 <=> i = n -> T(n) = n + T(0) = n + 1

T(n) = n + 1 + Q(n)s2
T(n) ∈ Q(n)

$O(n) \; = \; 1 \; + \; O(n)_{s2} \; + \; n$

$O(n) \; = \; n \; + \; O(n)_{s2}$

$O(n)_{T} \; = \; 1 \; + \; O(n)_{s1}$

$O(n)_{T} \; = \; 1 \; + \; n \; + \; O(n)_{s2}$

Ahora calculamos s2

SetPoints

```
    /**
     * This method set the points to the bet gambler, only if the name of
the
     * horse matches with the result horse postion
     *
     * @param next
     * @param horsePositions
     * @param index
     */
    private void setPoints(Bet next, int[] horsePositions, int index) {
        if (next.getHorses()[index] == horsePositions[index]) {
            this.realStepLoop += 2;
            this.stepLoop++;
            next.getGambler().setPoints(next.getGambler().getPoints() + 10
- index);
        }
        if (index == next.getHorses().length - 1) {
            this.realStepLoop++; // end or rec
            return;
        } else {
            index++;
            this.stepLoop++;
            this.realStepLoop += 2; // index increase and recursive call
            setPoints(next, horsePositions, index);
        }
    }
```

Talla: values: n

Base case: T(n) = 1

General case: T(n) = 1 + T(n - 1) => possible next Nodes:n

Recurrency ecc:

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 + T(n-1) & n > 0 \end{cases}$$

-> recurrency deploy

T(n) = 1 n = 0
1 + T(n - 1) n > 0

T(n) = 1 + T(n - 1) n > 0
= 1 + (1 + T(n - 2)) = 2 + T(n - 2)
= 2 + (1 + T(n - 3)) = 3 + T(n - 3)
= ...
= i + T(n - i)

-> at last call n - i = 0 <=> i = n -> T(n) = n + T(0) = n + 1

T(n) = n + 1
T(n) ∈ Q(n)

Q(n) = n

$$O(n) = n$$

$$O(n)_T = 1 + n + O(n)_{s2}$$

$$O(n)_T = 1 + n + n$$

$$O(n)_T = 1 + n$$

$$O(n)_T = n$$

Hemos demostrado que calcular los puntos de una única apuesta nos lleva <<n>> pasos.

Ordenamiento de resultados

Se nos solicitaba una complejidad de n^2 para esta tarea, y se aplicò el método de la burbuja puesto que solo queríamos cambiar valores dentro de una lista enlazada.

Ordenamiento por nombre

```
/**
    * method to sort an array of Strings the array is part of an Object
named
    * "bets" the method sorts the array by the String "gambler"
    *
    * @param list the bet list
    * @return the status of the action
    */
   public ReportStatus sortByGambler(NodeList<Bet> list) {
       resetSteps();
       try {
           this.timer.run();
           Node<Bet> current = list.getTail();
           while (current != null) {
               addSteps(1, 2); // while loop and assignation
               Node<Bet> comparing = current;
               while (comparing != null) {
                   addSteps(1, 2); // while loop and assignation
                   if (comparing.getData().getGambler().getName()

.compareTo(current.getData().getGambler().getName()) > 0) {
                       Bet tmp = comparing.getData();
                       comparing.setData(current.getData());
                       current.setData(tmp);
                       addRealSteps(4);
                   }
                   comparing = comparing.getNext();
                   addRealSteps(1);
               }
               current = current.getNext();
               addRealSteps(1);
           }
```

```
            this.timer.stopTimer();
            this.time = this.timer.getTotalTime();
            addSteps(1, 4); // start timer and assignation
            // promedium
            this.steps = this.steps / list.getSize();
            this.realSteps = this.realSteps / list.getSize();
            return ReportStatus.SUCCESS;
        } catch (Exception e) {
            addRealSteps(1);
            return ReportStatus.FAILURE;
        }
    }
```

Para calcular O(n) tomamos en cuenta que el primer while nos genera n intentos y el segundo while n intentos. Todo lo que està dentro del segundo while es un O(1) por lo que nuestra ecuaciòn final es:

$$O(n) = n(n(1 + 1 + 1... + 1))$$
$$O(n) = n(n(1))$$
$$O(n) = n(n)$$
$$O(n) = n^2$$

Demostramos que ordenar los resultados nos llevan <<n>> pasos.

Ordenamiento por puntos

```
    /**
     * method to sort an array of Doubles the array is part of an Object
named
     * "bets" the method sorts the array by the Double "amount"
     *
     * @param bets the bet list
     * @return
     */
    public ReportStatus sortByPoints(NodeList<Bet> list) {
        try {
            this.timer.run();
            Node<Bet> current = list.getTail();
            // loop for item
```

```java
            while (current != null) {
                int stepByLoop = 1;
                int realStepByLoop = 2;
                Node<Bet> comparing = current;
                while (comparing != null) {
                    stepByLoop++;
                    realStepByLoop++;
                    if (((Bet)
comparing.getData()).getGambler().getPoints() >
current.getData().getGambler()
                            .getPoints()) {
                        Bet tmp = (Bet) comparing.getData();
                        comparing.setData(current.getData());
                        current.setData(tmp);
                        realStepByLoop += 4;
                    }
                    comparing = comparing.getNext();
                    realStepByLoop++;
                }
                current = current.getNext();
                calcMostLessSteps(stepByLoop);
                addSteps(stepByLoop, realStepByLoop);
            }
            this.timer.stopTimer();
            this.time = this.timer.getTotalTime();
            // promedium
            this.steps = this.steps / list.getSize();
            this.realSteps = this.realSteps / list.getSize();
            addSteps(1, 4);
            return ReportStatus.SUCCESS;
        } catch (Exception e) {
            return ReportStatus.FAILURE;
        }

    }
```

$O(n) = n(n(1 + 1 + 1\ldots + 1))$
$O(n) = n(n(1))$
$O(n) = n(n)$

$O(n) = n^2$

Demostramos que ordenar los resultados nos llevan <<n>> pasos.

Complejidad de metodos no críticos

Cálculo de máximo o mínimo de pasos

```
/**
 * Calculates if the actual steps are more or less than the saved
 *
 * @param steps
 */
private void calcMostLessSteps() {
    if (this.mostSteps == this.lessSteps && this.mostSteps == 0) {
        this.mostSteps = this.lessSteps = this.stepsByLoop;
    } else {
        if (this.stepsByLoop > this.mostSteps) {
            this.mostSteps = steps;
        } else if (this.stepsByLoop < this.lessSteps) {
            this.lessSteps = steps;
        }
    }
}
```

$O(n) = 1 + 1 + 1 + 1 + 1 + 1 + 1$
$O(n) = 1$

La complejidad de calculator la mayor o menor cantidad de Pasos es de 1

Iniciar y terminar reloj

```
public void startTimer() {
    this.startTime = System.currentTimeMillis();
    this.isRunning = true;
}

public void stopTimer() {
    this.endTime = System.currentTimeMillis();
    this.totalTime = this.endTime - this.startTime;
    this.isRunning = false;
}
```

Puesto que sólo llamamos a la variable y las inicializamos siempre tenemos O(1)

$O(n) = 1 + 1$
$O(n) = 1$

Iniciar y terminar el reloj nos toma una complejidad de 1

Main

```
 public static void main(String[] args) {
     MainView1 view = new MainView1();
     view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     // fullscreen
     view.setLocationRelativeTo(null);
     view.setVisible(true);
 }
```

$O(n) = 1 + 1 + 1 + 1 + 1$
$O(n) = 1$

ANEXOS

Método para leer desde un archivo de texto

Lectura línea por línea del archivo de texto. Este método llama a un sub metodo para crear un arreglo con los valores ingresados para los caballos.

```
public ReportStatus analyzeFile(String fileName, NodeList<Bet> list) {
       File file = new File(fileName);
       try ( Scanner reader = new Scanner(file)) {
           while (reader.hasNextLine()) {
               String[] data = reader.nextLine().split(",");
               saveDataFromVector(data, list);
           }
           return ReportStatus.SUCCESS;
       } catch (FileNotFoundException e) {
           return ReportStatus.FILE_NOT_FOUND;
       }
```

```
    }
```

Obtener información de cada línea y llamar al método O(1) para insertar una nueva apuesta

```
private void saveDataFromVector(String[] data, NodeList<Bet> list) {
        try {
            Gambler gambler = new Gambler(data[0].trim()); // always the 1st val is the name
            Double ammount = data.length - 1 > 0 ? Double.valueOf(data[1].trim()) : 0; // always the 2nd val is the ammount
            int horsesLength = data.length - 2 < 0 ? (data.length - 1 < 0 ? 0 : data.length - 1) : data.length - 2;
            int[] horses = new int[horsesLength]; // there must be 10 horses\
            for (int i = 0; i < data.length - 2; i++) {
                try {
                    horses[i] = Integer.valueOf(data[i + 2].trim());
                } catch (NumberFormatException e) {
                    horses[i] = -1;
                }
            }
            list.addAtHead(new Bet(gambler, ammount, horses));
        } catch (NumberFormatException e) {
            System.out.println("Error in saveDataFromVector " + e.getMessage());
        }
    }
```

Exportar a CSV

Metodo comun

```
public void exportCSV(boolean exportValids, NodeList<Bet> bets) {
        String path = getPath();
        if (path == null) {
            JOptionPane.showMessageDialog(null,
                     "No se puede guardar el archivo en la ruta
especificada, si quieres volver a exportar los archivos vuelve a correr el
analisis");
        } else {
            writeToFile(bets, path + ".csv", exportValids);
        }
    }
```

$O(n) = 1 + 1 + O(n)_{s1} + O(n)_{s2}$

Para conocer la complejidad total debemos calculator la complejidad del metodo <<getPath>> y <<writeToFile>>

```
private String getPath() {
        try {
            JFileChooser fChooser = new JFileChooser();
            fChooser.showSaveDialog(null);
            File file = fChooser.getSelectedFile();
            return file.getAbsolutePath();
        } catch (Exception e) {
            return null;
        }

    }
```

$O(n) = 1 + 1 + 1 + 1$

$O(n) = 1$

$O(n)_{s1} = O(n) = 1$

```
private void writeToFile(NodeList<Bet> bets, String path, boolean
exportValids) {
        try {
```

```
            File file = new File(path);
            FileWriter writer;
            Node<Bet> current = bets.getTail();
            if (!file.exists()) {
                file.createNewFile();
            }

            writer = new FileWriter(file);
            while (current != null) {
                if (current.getData().isValid() == exportValids) {
                    String horses = "";
                    for (int i = 0; i <
current.getData().getHorses().length; i++) {
                        horses += "," + current.getData().getHorses()[i];
                    }
                    writer.write(
                            current.getData().getGambler().getName() + ","
+ current.getData().getAmount() + horses
                            + "\n");
                }
                current = current.getNext();
            }
            writer.close();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null,
                    "No se pudo guardar el archivo en la ruta especificada,
se usará la ruta por defecto");
        }
    }
```

$O(n) = 1 + 1 + 1... + 1 + n + 1$

$O(n) = n$

$O(n) = O(n)_{s2} = n$

Con esto podemos calcular la complejidad de <<exportCSV>> el cual es

$O(n) = 1 + 1 + O(n)_{s1} + O(n)_{s2}$

$O(n)_{s1} = 1$

$$O(n)_{s2} = n$$

$$O(n) = 1 + 1 + 1 + n$$

$$O(n) = n$$