Universidad de San Carlos de Guatemala USAC.
División de Ciencias de la Ingeniería.
Centro Universitario de Occidente CUNOC.
Estructura de Datos
Ing. Oliver

Estudiante	***Carnet***
Elvis Lizandro Aguilar Tax	*201930304*

Practica No. 1 – Documentación Técnica

Manejador de apuestas de carreras de Caballos

CÁLCULO DE COMPLEJIDAD DE LOS ALGORITMOS DE LOS SERVICIOS CRÍTICOS :

1. *Cálculo de Ingreso de apuesta*

```java
public class ControlIngreso {

    /**
     * Ingreso de apuesta O(1) al arreglo de apuestas
     * @param apuesta
     * @param apuestas
     * @param pos
     */
    public void ingresarApuesta(Apuesta apuesta, Apuesta[] apuestas) {
        apuestas[apuestas.length-1] = apuesta;
    }
```

O(1)

*En el algoritmo de de ingreso de las apuestas al arreglo es de **O(1)** ya la apuesta a ingresar se considera como una inserción al final del arreglo por lo tanto no afecta en los más mínimo al arreglo y no deber hacer nada más que insertarla al final.*

2. *Cálculo de Verificación de Apuestas*

```java
    /**
     * el primer For es de complejidad O(n) y el for anidado es constante de 10
     * maximo por lo tanto es O(1)
     * @param apuestas
     * @return apuestas verificadas
     */
    public Apuesta[] verficadorApuestas(Apuesta[] apuestas) {
        long tiempoInicial = System.nanoTime();
        this.tamaño = apuestas.length;
        for (int i = 0; i < apuestas.length; i++) {                          //O(n)
            int[] posTem = new int[10];                                      //O(1)
            for (int j = 0; j < 10; j++) {                                   //O(1)
                if (!posCorrecta(apuestas[i].getOrdenLlegada()[j], posTem)) {//O(1)
                    rechazadas += apuestas[i].toString();                    //O(1)
                    apuestas[i] = null;                                      //O(1)
                    tamaño--;
                    break;
                }
            }
            tiempoEjecucion += (System.nanoTime() - tiempoInicial);
        }
        tiempoPromedioVerificacion = tiempoEjecucion / apuestas.length;
        return apuestas;
    }
```

O(n)

*La complejidad del algoritmo de verificación es de O(n) ya que el **for** recorre N veces al arreglo por lo tanto hará n pasos en su peor caso no obstante existe un segundo **for** anidado por lo tanto su complejidad debe ser calculada.*

```java
            for (int j = 0; j < 10; j++) {                                   //O(1)
                if (!posCorrecta(apuestas[i].getOrdenLlegada()[j], posTem)) {//O(1)
                    rechazadas += apuestas[i].toString();                    //O(1)
                    apuestas[i] = null;                                      //O(1)
                    tamaño--;
                    break;
                }
```

*En este **for** anidado se puede notar que solo existe un corrimiento de 10 veces por lo tanto es independiente, y se considera un **O(1)** claro esta que se hará 10 como máximo es importante mencionar que esto podría llegar a ser menos pero se tomo 10 en su peor caso.*

ahora se verifica si la función dentro del for también cuenta con una complejidad de O(1)

```java
/**
 * verifica que la posicion no este repetida en un arreglo donde solo puede
 * existir 10 valores
 * @param index
 * @param pos
 * @return
 */
public boolean posCorrecta(int index, int pos[]) {                  //O(1)
    if (index > 0 && index <= 10 && pos[index - 1] == 0) {
        pos[index - 1] = index;
        return true;
    } else {
        return false;
    }
}
```

O(1)

esta función simplemente hace comparaciones y lo hace 1 vez, es importante que esto se realizará cada que ver que el for anidado, no obstante esto retorna como complejidad un O(1)

*Por lo tanto se concluye que el algoritmo de verificación es un O(n) ya que la suma en total de complejidad es O(n) primer **For,** O(1) segundo **for** y O(1) la funcion, esto hace una suma de **n+1+1=n+2** por notación BigO, tomamos el **O(n)***

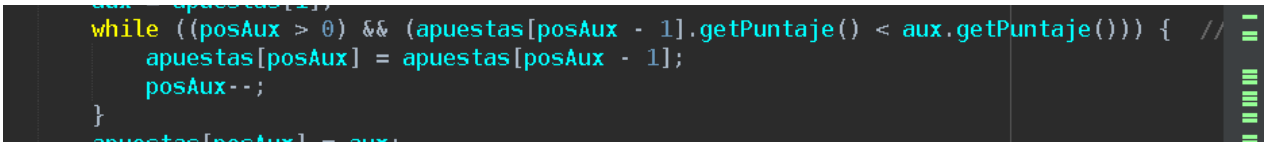
3. *Cálculo de ordenamiento*

```java
/**
 * algoritmo para ordenamiento por puntaje O(n^2) metodo de insersion, Puntajes
 * @param apuestas
 * @return
 */
public Apuesta[] ordenarPorPuntaje(Apuesta[] apuestas) {            //O(n^2)
    this.inicializarTiempos();
    Apuesta aux;
    int posAux;
    for (int i = 0; i < apuestas.length; i++) {                             //O(n)
        long tiempoI = System.nanoTime();                                   //O(1)
        posAux = i;
        aux = apuestas[i];
        while ((posAux > 0) && (apuestas[posAux - 1].getPuntaje() < aux.getPuntaje())) {
            apuestas[posAux] = apuestas[posAux - 1];
            posAux--;
        }
        apuestas[posAux] = aux;
        tiempoEjecucionPunaje += (System.nanoTime() - tiempoI);
    }
    this.tiempoPromedioOrdenamientoPuntaje = tiempoEjecucionPunaje / apuestas.length;
    return apuestas;
}
```

O(n^2)
O(n)

O(n)

*En este algoritmo es un ordenamiento por inserción, como se puede observar en la imagen este algoritmo utiliza un **For** principal que recorrerá n veces el arreglo por lo tanto es un O(n) sin embargo dentro de él existe un **While** que también depende del tamaño del arreglo por lo que en su pero caso recorrerá n veces el mismo retornando un O(n)*



usando un iterador que depende de n del arreglo esto implica que recorrerá n-1 veces en su peor caso no obstante como lo hará n vece es un O(n);

Por lo tanto el algoritmo de ordenamiento es un ***O(n^2).***

ARGUMENTACIÓN DEL METODO DE INSERCION PARA ORDENAMIENTO

El metodo de insercion cumple con la complejidad ***O(n^2),*** *mas sin embargo al compararlo con otro métodos como el de burbuja, este algoritmo en el peor de los casos es de* ***O(n^2)*** *y en el mejor de los casos logra reducirse a log2(n) ya que no recorre todo el while de adentro ya que va dejando un arreglo ordenado desde el primer apuntador que utiliza, ahora bien el de burbuja o de selección entre otros tiene que recorrer todo siempre ya que usan eso para verificar uno por uno en cada vuelta.*