Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

COMPLEJIDADES DE ALGORITMOS

Clase HashTable

- Método:

```
// constructor que recibe el tamaño del arreglo
@SuppressWarnings("unchecked")
public HashTable(int tamaño) {

    this.tamaño = tamaño;                                          //1
    // inicializando array de lista con tamaño
    tabla = new LinkedList[tamaño];                                //1
    for (int i = 0; i < tamaño; i++) {                             //n
        // recorriendo cada posicion del array e inicializando una lista
        tabla[i] = new LinkedList<>();                             //n
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad O(n), porque itera una n cantidad de veces según sea el numero recibido por parámetro.

- Método:

```
private int funcionHash(String placa) {
    int sumaAscii = 0;                                             //1
    for (char c : placa.toCharArray()) {                           //n
        // sumando todos los valores ascii de la placa
        sumaAscii += c;                                            //n
    }
    // devolviendo el modulo entre la suma y el tamaño
    return sumaAscii % tamaño;                                     //1
}
```

Justificación de complejidad:

Este metodo tiene la complejidad O(n) ya que recorre un String n veces según sea el tamaño de la placa del vehiculo que se esta recibiendo como parámetro.

- Método:

```
public boolean insertar(Vehiculo vehiculo) {
    if (placaDuplicada(vehiculo.getPlaca())) {                                          //1
        System.out.println("\nError, el vehiculo con la placa: " + vehiculo.getPlaca() + ", ya existe.");   //1
        return false;                                                                   //1
    }
    // obteniendo el indice del vehiculo
    int indice = funcionHash(vehiculo.getPlaca());                                      //1
    // agregando al vehiculo a la tabla con el indice que devolvio la funcion hash
    tabla[indice].add(vehiculo);                                                        //1
    return true;                                                                        //1
}
```

Justificación de complejidad:

Este metodo tiene la complejidad O(1) ya que todas las acciones que realiza no dependen de un tamaño de entrada, y se ejecutan una sola vez.

- Método:

```
private boolean placaDuplicada(String placa) {
    int indice = funcionHash(placa);                               //1
    boolean existe = false;                                        //1
    if (tabla[indice].buscarPorPlaca(placa) != null) {             //1
        existe = true;                                             //1
    }
    return existe;                                                 //1
}
```

Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

Justificación de complejidad:

Este método tiene la complejidad O(1) ya que realiza una única búsqueda dada una placa recibida para verificar que no este duplicada en el sistema por parámetro, en la llamada del metodo.

- Método:

```
public Vehiculo buscar(String placa) {                              //1
    System.out.println("Buscando vehiculo con placa: " + placa);    //1
    // obteniendo el indice del vehiculo a buscar
    int indice = funcionHash(placa);                                //1
    // buscando y retornando al vehivulo por la placa
    return tabla[indice].buscarPorPlaca(placa);                     //1
}
```

Justificación de complejidad:

Este método tiene la complejidad O(1) ya que realiza una única búsqueda dada una placa recibida por parámetro, en la llamada del metodo.

- Método:

```
public void mostrarVehiculosEnDestino(HashTable tablaVehiculos, int opcion) {
    int numero = 0;                                                     //1
    System.out.println();                                               //1
    for (int i = 0; i < tablaVehiculos.tamaño; i++) {                   //n

        // recorriendo cada bucket de la tabla
        LinkedList<Vehiculo> bucketActual = tablaVehiculos.tabla[i];    //n
        // recorriendo lista que esta en el bucket [i]
        Node<Vehiculo> actual = bucketActual.getHead();                 //n

        while (actual != null) {                                        //n^2
            Vehiculo vehiculoActual = actual.getData();                 //n^2
            if (opcion == 1) {                                          //n^2
                if (vehiculoActual.isEnDestino()) {                     //n^2
                    numero++;                                           //n^2
                    System.out.println(numero + ". " + vehiculoActual.toString()); //n^2
                }
            } else if (opcion == 2) {                                   //n^2
                if (!vehiculoActual.isEnDestino()) {                    //n^2
                    numero++;                                           //n^2
                    System.out.println(numero + ". " + vehiculoActual.toString()); //n^2
                }
            }
            actual = actual.getNext();                                  //n^2
        }
    }
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n^2) ya que recorre un arreglo de listas enlazadas, y en cada posición del arreglo o cada lista, recorre esa lista n cantidad de veces para mostrar cada vehiculo en cada posición de cada lista.

Clase LinkedList

- Método:

```
public void add(T data) {
    Node<T> newNode = new Node<>(data);         //1
    if (head == null) {
        head = newNode;                         //1
    } else {
        Node<T> current = head;                 //1
        while (current.getNext() != null) {     //n
            current = current.getNext();        //n
        }
        current.setNext(newNode);               //1
        newNode.setPrev(current);               //1
    }
    size++;                                     //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que recorre los nodos siguientes de un nodo una cantidad de n veces hasta que el nodo ya no tenga un siguiente para colocárselo como siguiente y aumentar el tamaño de la lista.

- Método:

```
public T find(T data) {
    Node<T> current = head;                    //1
    while (current != null) {                  //n
        if (current.getData().equals(data)) {  //n
            return current.getData();          //n
        }
        current = current.getNext();           //n
    }
    return null;                               //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que recorre un while n veces hasta que el nodo actual ya no tenga siguiente, o hasta que el dato que esta buscando sea igual al dato que se recibe por parámetro.

- Método:

```
public boolean remove(T data) {
    Node<T> current = head;                                  //1

    while (current != null) {                                //n
        if (current.getData().equals(data)) {                //n
            if (current.getPrev() != null) {                 //n
                current.getPrev().setNext(current.getNext());//n
            } else {
                head = current.getNext();                    //n
            }

            if (current.getNext() != null) {                 //n
                current.getNext().setPrev(current.getPrev());//n
            }
            size--;                                          //n
            return true;                                     //n
        }
        current = current.getNext();                         //n
    }
    return false;                                            //n
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que recorre un while n veces hasta que el nodo actual ya no tenga siguiente, o hasta que el dato que esta buscando sea igual al dato que se recibe por parámetro y lo elimina de la lista.

- Método:

```
public void imprimir() {
    Node<T> current = head;                          //1
    while (current != null) {                        //n
        System.out.print(current.getData() + "\n");  //n
        current = current.getNext();                 //n
    }
    System.out.println();                            //1
}
```

Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

Justificación de complejidad:

Este método tiene una complejidad de O(n) ya que recorre un while n veces hasta que el nodo actual ya no tenga siguiente y muestra cada objeto en consola.

- Método:

```java
public Vehiculo buscarPorPlaca(String placa) {
    Node<T> current = head;                                          //1

    while (current != null) {                                        //n
        if (current.getData() instanceof Vehiculo) {                 //n
            Vehiculo vehiculo = (Vehiculo) current.getData();        //n
            if (vehiculo.getPlaca().equals(placa)) {                 //n
                return vehiculo;                                     //n
            }
        }
        current = current.getNext();                                 //n
    }
    return null;                                                     //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que recorre un while n veces hasta que el nodo actual ya no tenga siguiente, o hasta que el dato que esta buscando sea igual al dato que se recibe por parámetro usando como metodo de comparación la placa.

Clase OrthogonalMatrix

- Método:

```java
public OrthogonalMatrix(int ancho, int alto) {
    this.ancho = ancho;                                  //1
    this.alto = alto;                                    //1
    head = new Node<>(data:null);                        //1
    Node<T> nodoActual = head;                           //1

    for (int y = 0; y < alto; y++) {                     //n
        for (int x = 0; x < ancho; x++) {                //n^2
            if (x == 0 && y == 0) {                      //n^2
                continue;                                //n^2
            }
            Node<T> nuevoNodo = new Node<>(data:null);   //n^2
            if (x > 0) {                                 //n^2
                nuevoNodo.setLeft(nodoActual);           //n^2
                nodoActual.setRight(nuevoNodo);          //n^2
                nodoActual = nuevoNodo;                  //n^2
            } else {
                nodoActual = nuevoNodo;                  //n^2
            }
            if (y > 0) {                                       //n^2
                Node<T> nodoAbajo = obtenerNodo(x, y - 1);     //n^2
                nuevoNodo.setDown(nodoAbajo);                  //n^2
                nodoAbajo.setUp(nuevoNodo);                    //n^2
            }
        }
    }
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n^2) ya que recorre un for dentro de otro for dependiendo de las dimensiones de alto como de ancho que tiene la matriz y en cada posición va interconectando los nodos con sus vecinos.

- Método:

```java
public Node<T> obtenerNodo(int x, int y) {
    if (x < 0 || y < 0 || x >= ancho || y >= alto) {    //1
        return null;                                    //1
    }
    Node<T> actual = head;                              //1

    for (int i = 0; i < x && actual != null; i++) {     //n
        actual = actual.getRight();                     //n
    }
    for (int i = 0; i < y && actual != null; i++) {     //n
        actual = actual.getUp();                        //n
    }
    return actual;                                      //n
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que recorre un for n cantidad de veces hasta llegar a una poscion en x y es ahí cuando recorre otro for n cantidad de veces hasta llegar a una posición en y y retorna el dato en la posición.

- Método:

```java
public void insertarDato(int x, int y, T data) {
    Node<T> nodo = obtenerNodo(x, y);                           //1

    if (nodo != null) {                                         //1
        nodo.setData(data);                                     //1
    } else {
        throw new IndexOutOfBoundsException(s:"Posicion invalida"); //1
    }
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(1) ya que recibe un dato y lo inserta en la posición recibida como parámetro.

- Método:

```java
public T obtenerDato(int x, int y) {
    Node<T> nodo = obtenerNodo(x, y);                           //1
    if (nodo != null) {                                         //1
        T dato = nodo.getData();                                //1
        return dato;                                            //1
    } else {
        throw new IndexOutOfBoundsException(s:"Posicion invalida"); //1
    }
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(1) ya que encuentra un dato en la posición recibida como parámetro y lo devuelve.

- Método:

```java
public void imprimir() {
    System.out.println(x:"\nMapa de la ciudad:");                                    //1
    System.out.println(x:"*****************************");                          //1
    System.out.print(s:"   ");                                                       //1
    for (int x = 0; x < ancho; x++) {                                                //n
        System.out.print(" " + (x + 1) + " ");                                       //n
    }
    System.out.println();                                                            //1
    for (int y = 0; y < alto; y++) {                                                 //n
        System.out.print(letrasFilas[y] + "  ");                                      //n

        Node<T> fila = obtenerNodo(x:0, y);                                          //n
        for (int x = 0; x < ancho && fila != null; x++) {                            //n^2
            Interseccion interseccion = (Interseccion) fila.getData();               //n^2
            System.out.print(" " + interseccion.getRepresentacionConsola() + " ");//n^2
            fila = fila.getRight();                                                  //n^2
        }
        System.out.println();                                                        //n
    }
    System.out.println(x:"*****************************\n");                        //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n^2) ya que recorre un for dentro de otro for dependiendo de las dimensiones de alto como de ancho que tiene la matriz y en cada posición va imprimiendo en consola el dato que haya en la posición.

Clase PriorityQueueI

- Método:

```java
public void insertar(Interseccion interseccion) {
    Node<Interseccion> nuevo = new Node<>(interseccion);                             //1

    if (head == null || compararPorComplejidad(interseccion, head.getData()) > 0) {  //1
        nuevo.setNext(head);                                                         //1
        if (head != null) {                                                          //1
            head.setPrev(nuevo);                                                     //1
        }
        head = nuevo;                                                                //1
    } else {                                                                         //1
        Node<Interseccion> actual = head;                                            //1
        while (actual.getNext() != null &&
                compararPorComplejidad(interseccion, actual.getNext().getData()) <= 0) {//n
            actual = actual.getNext();                                               //n
        }

        nuevo.setNext(actual.getNext());                                             //1
        if (actual.getNext() != null) {                                              //1
            actual.getNext().setPrev(nuevo);                                         //1
        }
        actual.setNext(nuevo);                                                       //1
        nuevo.setPrev(actual);                                                       //1
    }
    size++;                                                                          //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n) ya que inserta un dato en la posicion n hasta que se encuentra la posición adecuada para colocarse, dada la complejidad de la intersección.

- Método:

```
public Interseccion desencolar() {
    if (head == null) {                          //1
        return null;                             //1
    }

    Interseccion interseccion = head.getData(); //1
    head = head.getNext();                       //1
    if (head != null) {                          //1
        head.setPrev(prev:null);                 //1
    }
    size--;                                      //1
    return interseccion;                         //1
}
```

Justificación de complejidad:

Este método tiene una complejidad de O (1) ya que toma el primer elemento de la cola y lo devuelve.

- Método:

```
public void actualizarInterseccion(String nombre) {
    if (head == null){                                                    //1
        return;                                                           //1
    }

    Node<Interseccion> actual = head;                                     //1

    while (actual != null) {                                              //n
        if (actual.getData().getNombre().equals(nombre)) {                //n
            // Encontramos la intersección, ahora la quitamos de la cola
            Interseccion interseccion = actual.getData();                 //n

            // Desenlazar el nodo actual
            if (actual.getPrev() != null) {                               //n
                actual.getPrev().setNext(actual.getNext());               //n
            } else {                                                      //n
                head = actual.getNext(); // era la cabeza                 //n
            }

            if (actual.getNext() != null) {                               //n
                actual.getNext().setPrev(actual.getPrev());               //n
            }
            size--;                                                       //n

            // Calcular nueva complejidad
            interseccion.calcularComplejidad();                           //n

            // Reinsertar en la cola
            insertar(interseccion);                                       //n
            return;                                                       //n
        }
        actual = actual.getNext();                                        //n
    }
}
```

Justificación de complejidad: Este metodo tiene una complejidad de O(n) ya que recorre una cola en búsqueda de una intersección usando el nombre como id, para desencolarlo, calcular su complejidad y volver a encolarlo.

- Método:

```
public void imprimir() {
    if (estaVacia()) {                                          //1
        System.out.println(x:"Cola de prioridad vacia\n");      //1
        return;                                                   //1
    }

    Node<Interseccion> actual = head;                          //1
    int posicion = 1;                                          //1
    while (actual != null) {                                   //n
        Interseccion i = actual.getData();                     //n
        System.out.println(posicion + ") " + i.getNombre() + " - complejidad: " + i.getComplejidad());
        actual = actual.getNext();                             //n
        posicion++;                                            //n
    }
    System.out.println(x:"====================================\n");     //1
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n), ya que recorre una lista n veces y en cada posición muestra en consola el dato del nodo en el que este iterando.

Clase Archivo

- Método:

```
public LinkedList<Vehiculo> leerArchivo(String ruta) {
    LinkedList<Vehiculo> listaVehiculos = new LinkedList<>();              //1
    File archivo = new File(ruta);                                         //1
    System.out.println("Abriendo archivo CSV: " + ruta);                   //1

    try (FileReader lector = new FileReader(archivo);                      //1
            BufferedReader buffer = new BufferedReader(lector)) {          //1

        System.out.println(x:"Leyendo datos de vehículos...");               //1
        String linea;                                                      //1

        while ((linea = buffer.readLine()) != null) {                      //n

            String[] datos = linea.split(regex:",");                       //n

            try {
                TipoVehiculo tipo = TipoVehiculo.valueOf(datos[0].trim());  //n
                String placa = datos[1].trim();                            //n
                String origen = datos[2].trim();                           //n
                String destino = datos[3].trim();                          //n
                int prioridad = Integer.parseInt(datos[4].trim());         //n
                int tiempoEspera = Integer.parseInt(datos[5].trim());      //n

                Vehiculo vehiculo = new Vehiculo(tipo, placa, origen, destino, prioridad, tiempoEspera);

                listaVehiculos.add(vehiculo);                              //n
            } catch (IllegalArgumentException e) {                         //n
                System.err.println("Error al procesar el dato " + e.getMessage());
            }
        }
    } catch (IOException e) {
        System.err.println("\nError al leer el archivo " + e.getMessage()); //1
    }
    return listaVehiculos;                                                 //1
}
```

Justificación de complejidad:

Este metodo tiene la complejidad O(n) ya que recorre un while n veces dependiendo de la cantidad de vehículos haya en el archivo y por cada vehiculo o línea crea la instancia y guarda el vehiculo en la lista.

Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

Clase InserscionDirecta

- Método:

```
public LinkedList<Vehiculo> ordenarPorPrioridad(LinkedList<Vehiculo> listaDesordenada) {
    LinkedList<Vehiculo> listaOrdenada = new LinkedList<>();        //1

    Node<Vehiculo> actual = listaDesordenada.getHead();             //1
    while (actual != null) {                                        //n
        Vehiculo vehiculoActual = actual.getData();                 //n
        insertarVehiculo(listaOrdenada, vehiculoActual);            //n
        actual = actual.getNext();                                  //n
    }
    return listaOrdenada;                                           //n
}
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n^2) ya que recorre una lista n veces e inserta cada elemento de manera ordenada para asi retornar otra lista con los datos ya ordenados, pero en el proceso de ordenación recorre otra lista para buscar la posición correcta.

- Método:

```
private void insertarVehiculo(LinkedList<Vehiculo> lista, Vehiculo vehiculo) {
    Node<Vehiculo> nuevoNodo = new Node<>(vehiculo);        //1

    if (lista.isEmpty()) {                                  //1
        // si la lista esta vacia se inserta al inicio
        lista.add(vehiculo);                                //1
        return;
    }

    Node<Vehiculo> actual = lista.getHead();                //1
    Node<Vehiculo> anterior = null;                         //1

    // se busca la poscion para insertar
    while (actual != null && actual.getData().getPrioridad() >= vehiculo.getPrioridad()) {
        anterior = actual;                                  //n
        actual = actual.getNext();                          //n
    }

    // si no hay un anterior se inserta al inicio
    if (anterior == null) {                                 //1
        nuevoNodo.setNext(lista.getHead());                 //1
        if (lista.getHead() != null) {                      //1
            lista.getHead().setPrev(nuevoNodo);             //1
        }
        lista.setHead(nuevoNodo);                           //1
    } else {
        // si hay anterior se inserta en el medio o final de la lista
        nuevoNodo.setNext(actual);                          //1
        nuevoNodo.setPrev(anterior);                        //1
        anterior.setNext(nuevoNodo);                        //1

        if (actual != null) {                               //1
            actual.setPrev(nuevoNodo);                      //1
        }
    }
    lista.setSize(lista.getSize() + 1);                     //1
}
```

Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

Justificación de complejidad:
este metodo tiene la complejidad de O(n) ya que inserta un vehiculo en una lista, pero recorre la lista n veces en busca de su mejor posición, y lo coloca ahí.

Clase MetodoDeLaSacudida

- Método:

```java
public void ordenarPorTiempoDeEspera(LinkedList<Vehiculo> lista) {
    // bandera para finalizar proceso
    boolean ordenado;                                       //1
    // limite inferior
    Node<Vehiculo> primero = lista.getHead();               //1
    // limite superior
    Node<Vehiculo> ultimo = null;                           //1
    do {
        ordenado = true;                                    //n
        Node<Vehiculo> actual = primero;                    //n
        // recorriendo de izquierda a derecha
        while (actual.getNext() != ultimo) {                //n^2
            if (actual.getData().getTiempoDeEspera() < actual.getNext().getData().getTiempoDeEspera()) {
                // si el actual es menor que el siguiente, se intercambian los datos del nodo
                Vehiculo tmp = actual.getData();            //n^2
                actual.setData(actual.getNext().getData()); //n^2
                actual.getNext().setData(tmp);              //n^2
                // se cambia la bandera para seguir iterando
                ordenado = false;                           //n^2
            }
            actual = actual.getNext();                      //n^2
        }
        // actualizando limite superior
        ultimo = actual;                                    //n
        if (ordenado) {                                     //n
            break;                                          //n
        }
        ordenado = true;                                    //n
        // recorriendo de derecha a izquierda
        actual = ultimo;                                    //n
        while (actual != primero && actual.getPrev() != null) {
            if (actual.getPrev().getData().getTiempoDeEspera() < actual.getData().getTiempoDeEspera()) {
                // si el anterior es menor que le actual, se intercambian los datos del nodo
                Vehiculo tmp = actual.getPrev().getData();  //n^2
                actual.getPrev().setData(actual.getData()); //n^2
                actual.setData(tmp);                        //n^2
                ordenado = false;                           //n^2
            }
```

Justificación de complejidad:

Este metodo tiene la complejidad de O(n^2) ya que itera con un while las veces que sea necesaria, pero en cada iteración hace un recorrido de izquierda a derecha y si es necesario de derecha a izquierda n veces hasta que la lista se encuentre totalmente ordenada.

Clase Simulador

- Método:

```java
private void generarMapaCiudad() {
    }
    for (int y = 0; y < filas; y++) {                                   //n
        char letraDeFila = letrasFilas[y];                              //n
        for (int x = 0; x < columnas; x++) {                            //n^2
            TipoInterseccion tipoInterseccion;                          //n^2
            String nombreInterseccion = letraDeFila + String.valueOf(x + 1);//n^2
            if (x == 0 && y == 0) {                                     //n^2
                // tipoInterseccion en esquina superior izquierda
                tipoInterseccion = TipoInterseccion.CRUCELVOLTEADAIZQUIERDA;//n^2
            } else if (x == columnas - 1 && y == 0) {                   //n^2
                // tipoInterseccion en esquina superior derecha
                tipoInterseccion = TipoInterseccion.CRUCELOPUESTA;      //n^2
            } else if (x == 0 && y == filas - 1) {                      //n^2
                // tipoInterseccion en esquina inferior izquierda
                tipoInterseccion = TipoInterseccion.CRUCEL;             //n^2
            } else if (x == columnas - 1 && y == filas - 1) {           //n^2
                // tipoInterseccion en esquina inferior derecha
                tipoInterseccion = TipoInterseccion.CRUCELVOLTEADADERECHA;  //n^2
            } else if (x == 0) {                                        //n^2
                // tipoInterseccion en columna afuera izquierda
                tipoInterseccion = TipoInterseccion.CRUCETVOLTEADAIZQUIERDA;//n^2
            } else if (x == columnas - 1) {                             //n^2
                // tipoInterseccion en columna afuera derecha
                tipoInterseccion = TipoInterseccion.CRUCETVOLTEADADERECHA;  //n^2
            } else if (y == 0) {                                        //n^2
                // tipoInterseccion en fila arriba
                tipoInterseccion = TipoInterseccion.CRUCET;             //n^2
            } else if (y == filas - 1) {                                //n^2
                // tipoInterseccion en fila abajo
                tipoInterseccion = TipoInterseccion.CRUCETOPUESTA;      //n^2
            } else {                                                    //n^2
                // tipoInterseccion en centro de matriz
                tipoInterseccion = TipoInterseccion.CRUCEMAS;           //n^2
            }
            Interseccion interseccion = new Interseccion(nombreInterseccion, tipoInterseccion);
            ciudad.insertarDato(x, y, interseccion);                    //n^2
        }
```

Justificación de complejidad:

Este metodo tiene una complejidad de O(n^2) ya que recorre un for dentro de otro for dependiendo de las dimensiones de alto como de ancho que tiene la matriz y en cada posición va insertando intersecciones según las posiciones en que se este iterando.

- Método:

```
private void repartirVehiculosCiudad(LinkedList<Vehiculo> listaVehiculos) {
    if (listaVehiculos == null || listaVehiculos.getSize() == 0) {          //1
        System.out.println("No hay vehiculos para repartir");               //1
        return;                                                             //1
    }
    int vehiculosRepartidos = 0;                                            //1
    int vehiculosPerdidos = 0;                                              //1

    while (!listaVehiculos.isEmpty()) {                                     //n
        Node<Vehiculo> nodoVehiculo = listaVehiculos.getHead();             //n
        Vehiculo vehiculo = nodoVehiculo.getData();                         //n

        Interseccion interseccionOrigen = obtenerInterseccion(vehiculo.getInterseccionOrigen());
        Interseccion interseccionDestino = obtenerInterseccion(vehiculo.getInterseccionDestino());

        if (interseccionOrigen != null && interseccionDestino != null) {//n
            PriorityQueueV cola = obtenerCola(interseccionOrigen);          //n

            if (cola != null) {                                             //n
                boolean insertado = tablaVehiculos.insertar(vehiculo);      //n
                if (!insertado) {                                           //n
                    listaDuplicados.add(vehiculo);                          //n
                } else {                                                    //n
                    cola.insertar(vehiculo);                                //n
                    listaGeneralVehiculos.add(vehiculo);                    //n
                    vehiculosRepartidos++;                                  //n
                    vehiculo.setInterseccionActual(interseccionOrigen.getNombre());
                    interseccionOrigen.setVehiculosCirculados(interseccionOrigen.getVehiculosCirculados() + 1);
                }
            }
        } else {                                                            //n
            System.out.println("Vehiculo con placa: " + vehiculo.getPlaca() +
                    ", tiene un origen o destino inexistente");             //n
            vehiculosPerdidos++;                                            //n
        }
        listaVehiculos.remove(vehiculo);                                    //n
    }
    System.out.println("Se repartieron: " + vehiculosRepartidos + "\nSe perdieron: " + vehiculosPerdidos);
}
```

Justificación de complejidad:

Este metodo tiene la complejidad de O (n) ya que recorre una lista de vehículos n veces hasta vaciarla para que con cada vehiculo lo inserte en la intersección que tiene como origen.

- Método:

```
private Interseccion obtenerInterseccion(String nombreInterseccion) {
    if (nombreInterseccion == null) {                       //1
        return null;                                        //1
    }

    int numeroDeFila = obtenerNumeroDeFila(nombreInterseccion); //1
    int numeroDeColumna = obtenerNumeroColumna(nombreInterseccion);//1

    if (numeroDeFila >= 0 && numeroDeFila < filas && numeroDeColumna >= 0 && numeroDeColumna < columnas) {
        Node<Interseccion> nodoInterseccion = ciudad.obtenerNodo(numeroDeColumna, numeroDeFila);    //1
        if (nodoInterseccion != null) {                     //1
            return nodoInterseccion.getData();              //1
        } else {                                            //1
            return null;                                    //1
        }
    } else {                                                //1
        // excediendo limites del tablero -> nodo inexistente
        return null;                                        //1
    }
}
```

Justificación de complejidad:

Este metodo tiene la complejidad de O(1) ya que dadas unas coordenadas retorna la intersección que se encuentre en esa posición de las coordenadas.

- Método:

```
private PriorityQueueV obtenerCola(Interseccion interseccion) {

    if (interseccion.getColaNorte() != null) {          //1
        return interseccion.getColaNorte();             //1
    } else if (interseccion.getColaSur() != null) {     //1
        return interseccion.getColaSur();               //1
    } else if (interseccion.getColaEste() != null) {    //1
        return interseccion.getColaEste();              //1
    } else if (interseccion.getColaOeste() != null) {   //1
        return interseccion.getColaOeste();             //1
    }
    return null;
}
```

Justificación de complejidad:

Este método tiene la complejidad do O (1) ya que retorna la 1ra cola que encuentre de la intersección recibida, recorrido que solo se hace una vez.

- Método:

```
private void calcularComplejidades() {
    for (int y = 0; y < filas; y++) {                           //n
        for (int x = 0; x < columnas; x++) {                    //n^2
            Interseccion interseccion = ciudad.obtenerDato(x, y);   //n^2
            interseccion.calcularComplejidad();                 //n^2
            arbolIntersecciones.insertar(interseccion);         //n^2
        }
    }
}
```

Justificación de complejidad: Este método tiene la complejidad de O(n^2) ya que recorre la matriz 2D por cada fila y cada columna, y en cada posición va calculando la complejidad de cada intersección y la almacena en la cola de intersecciones.

- Método:

```java
private void moverTrafico() {
    // obteniendo interseccion de mayor prioridad (se elimina de la cola)
    Interseccion interseccionPrioritaria = arbolIntersecciones.desencolar();        //1
    if (interseccionPrioritaria.getComplejidad() == 0) {                            //1
        System.out.println(x:"Ya no hay trafico en la ciudad, buenas noches.");     //1
        simulacionTerminada = true;                                                 //1
        return;                                                                     //1
    }
    // verificando que no este bloqueada
    if (interseccionPrioritaria.isBloqueda()) {                                     //1
        verificarBloqueo(interseccionPrioritaria);                                  //1
        return;                                                                     //1
    }
    String nombreOrigen = interseccionPrioritaria.getNombre();                      //1
    String mensaje = "Moviendo trafico en interseccion: " + nombreOrigen
            + ", con prioridad: " + interseccionPrioritaria.getComplejidad();       //1

    System.out.println(mensaje);                                                    //1
    registroEventos.push(mensaje);                                                  //1
    // obteniendo numero de fila y columna de la interseccion a trabajar
    int filaActual = obtenerNumeroDeFila(nombreOrigen);                             //1
    int columnaActual = obtenerNumeroColumna(nombreOrigen);                         //1

    // iterando a cada direccion posible de la interseccion
    for (Direccion direccion : Direccion.values()) {
        PriorityQueueV colaActual = interseccionPrioritaria.getColaPorDireccion(direccion);
        if (colaActual == null) {                                                   //n
            continue;                                                               //n
        }
        moverVehiculosDeCola(colaActual, interseccionPrioritaria, filaActual, columnaActual);
    }
    // actualizando complejidad y volviendo a encolar la interseccion
    interseccionPrioritaria.calcularComplejidad();                                  //1
    arbolIntersecciones.insertar(interseccionPrioritaria);                          //1
}
```

Justificación de complejidad:

Este método tiene la complejidad para el peor de los casos de O(n) ya que realiza movimientos para n direcciones según las colas que tenga la intersección que se esté descongestionando.

- Método:

```java
private void moverVehiculosDeCola(PriorityQueueV colaActual, Interseccion origen, int filaOrigen,
        int columnaOrigen) {
    PriorityQueueV colaTemporal = new PriorityQueueV();                         //1

    while (!colaActual.estaVacia()) {                                           //n
        // recorriendo vehiculo a vehiculo y procesandolo
        Vehiculo vehiculo = colaActual.desencolar();                            //n
        procesarVehiculo(vehiculo, origen, filaOrigen, columnaOrigen, colaTemporal);//n
    }
    while (!colaTemporal.estaVacia()) {                                         //n
        // si hubo vehiculos que no se movieron se vuelven a guardar
        colaActual.insertar(colaTemporal.desencolar());                         //n
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad de O (n) ya que mueve n vehículos de la cola que se este trabajando, y si al final la cola temporal no esta vacia, vuelve a encolar n vehículos de vuelta.

- Método:

```java
private void procesarVehiculo(Vehiculo vehiculo, Interseccion origen, int filaActual, int columnaActual,
        PriorityQueueV colaTemporal) {
    String destino = vehiculo.getInterseccionDestino();
    System.out.println("\nVehiculo:" + vehiculo.getPlaca() + ", con destino: " + destino + ", intenta avanzar.");
    int filaDestino = obtenerNumeroDeFila(destino);                         //1
    int columnaDestino = obtenerNumeroColumna(destino);                     //1

    if (filaActual == filaDestino && columnaActual == columnaDestino) {     //1
        // validacion extra por si el vehiculo inicia en su destino
        System.out.println("El vehiculo: " + vehiculo.getPlaca() + ", ya esta en su destino.");
        vehiculo.setEnDestino(enDestino:true);                               //1
        return;                                                             //1
    }
    // calculando mocimineto
    int nuevaFila = filaActual;                                             //1
    int nuevaColumna = columnaActual;                                       //1

    int distanciaFila = filaDestino - filaActual;                           //1
    int distanciaColumna = columnaDestino - columnaActual;                  //1
    Direccion direccionMovimiento = calcularDireccionMovimiento(distanciaFila, distanciaColumna);
    switch (direccionMovimiento) {                                          //1
        case NORTE:                                                         //1
            nuevaFila--;                                                    //1
            break;                                                          //1
        case SUR:                                                           //1
            nuevaFila++;                                                    //1
            break;                                                          //1
        case ESTE:                                                          //1
            nuevaColumna++;                                                 //1
            break;                                                          //1
        case OESTE:                                                         //1
            nuevaColumna--;                                                 //1
            break;                                                          //1
    }
    System.out.println("El vehiculo se movera en direccion: " + direccionMovimiento);
    Interseccion interseccionDestino = ciudad.obtenerDato(nuevaColumna, nuevaFila);

    if (!interseccionDestino.isBloqueda()) {                                //1
```

Justificación de complejidad:

Este método tiene la complejidad de O (1) ya que calcula un solo movimiento para el vehiculo que se esta movilizando y obtiene la intersección donde se realizara este movimiento, según el valor recibido de la funcion calcular dirección movimiento.

- Método:

```java
private Direccion calcularDireccionMovimiento(int distanciaFila, int distanciaColumna) {

    if (distanciaColumna != 0) {            //1
        if (distanciaColumna < 0) {         //1
            return Direccion.OESTE;         //1
        } else {                            //1
            return Direccion.ESTE;          //1
        }
    } else if (distanciaFila != 0) {        //1
        if (distanciaFila < 0) {            //1
            return Direccion.NORTE;         //1
        } else {
            return Direccion.SUR;           //1
        }
    } else {                                //1
        return null;                        //1
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad de O (1) ya que calcula un solo movimiento para el vehiculo que se esta movilizando.

- Método:

```
private void moverVehiculo(Vehiculo vehiculo, Interseccion origen, Interseccion destino, int filaDestino,
        int colDestino, int nuevaFila, int nuevaColumna) {
    System.out.println(x:"El vehiculo se esta moviendo...");             //1
    // calculando siguiente movimiento si lo hay
    int distanciaRestanteX = filaDestino - nuevaFila;                     //1
    int distanciaRestanteY = colDestino - nuevaColumna;                   //1
    Direccion proximaDireccion = calcularProximaDireccion(distanciaRestanteX, distanciaRestanteY);

    if (proximaDireccion != null) {                                       //1
        // si hay proxima direccion se inserta en el carril hacia ella
        destino.getColaPorDireccion(proximaDireccion).insertar(vehiculo);   //1
        String mensaje = "Vehiculo " + vehiculo.getPlaca() + " se movio de " +
                origen.getNombre() + " a " + destino.getNombre() + ", al carril: " + proximaDireccion + ",";
        vehiculo.setTiempoDeEspera(vehiculo.getTiempoDeEspera() + 1);       //1
        vehiculo.setInterseccionActual(destino.getNombre());                //1
        System.out.println(mensaje);                                        //1
        registroEventos.push(mensaje);                                      //1
        destino.setVehiculosCirculados(destino.getVehiculosCirculados() + 1);//1
        arbolIntersecciones.actualizarInterseccion(destino.getNombre());    //1
    } else {                                                                //1
        // si no hay proxima direccion, el vehiculo ya esta en su destino
        String mensaje = "Vehiculo con placa: " + vehiculo.getPlaca()
                + ", llego a su destino: " + vehiculo.getInterseccionDestino();
        vehiculo.setTiempoDeEspera(vehiculo.getTiempoDeEspera() + 1);       //1
        vehiculo.setEnDestino(enDestino:true);                               //1
        System.out.println(mensaje);                                        //1
        registroEventos.push(mensaje);                                      //1
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad de O (1) ya que, dada la nueva intersección y el vehículo a mover, mueve al vehículo a su nueva dirección, después de realizar unas validaciones.

- Método:

```
private Direccion calcularProximaDireccion(int distanciaFila, int distanciaColumna) {
    if (distanciaColumna != 0) {                    //1
        if (distanciaColumna < 0) {                 //1
            return Direccion.OESTE;                 //1
        } else {                                    //1
            return Direccion.ESTE;                  //1
        }
    } else if (distanciaFila != 0) {                //1
        if (distanciaFila < 0) {                    //1
            return Direccion.NORTE;                 //1
        } else {                                    //1
            return Direccion.SUR;                   //1
        }
    } else {                                        //1
        return null;                                //1
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad de O (1) ya que dadas unas distancias en X y Y, calcula la nueva direccion del siguiente movimiento, calcula un solo movimiento, por eso la complejidad constante.

- Método:

```java
private void verEstadoInterseccion() {
    sn.nextLine();
    System.out.println(x:"Ingresa el nombre de la interseccion que deseas ver (A1, B3, C2, etc).");         //1
    String nombreInterseccion = sn.nextLine();                                                              //1
    Interseccion interseccionBuscada = obtenerInterseccion(nombreInterseccion);                             //1

    if (interseccionBuscada != null) {                                                                      //1
        System.out.println(x:"\n****Estado de la interseccion****");                                        //1
        System.out.println("Nombre: " + interseccionBuscada.getNombre());                                   //1
        System.out.println("Complejidad: " + interseccionBuscada.getComplejidad());                         //1
        System.out.println("Hay bloqueo?: " + interseccionBuscada.isBloqueda());                            //1
        System.out.println("Representacion en consola: " + interseccionBuscada.getRepresentacionConsola()); //1
        System.out.println("Vehiculos circulados: " + interseccionBuscada.getVehiculosCirculados());        //1
        if (interseccionBuscada.getColaNorte() != null) {                                                   //1
            System.out.println(x:"Cola norte:");                                                            //1
            interseccionBuscada.getColaNorte().imprimir();                                                  //1
        }
        if (interseccionBuscada.getColaSur() != null) {                                                     //1
            System.out.println(x:"Cola sur:");                                                              //1
            interseccionBuscada.getColaSur().imprimir();                                                    //1
        }
        if (interseccionBuscada.getColaEste() != null) {                                                    //1
            System.out.println(x:"Cola este:");                                                             //1
            interseccionBuscada.getColaEste().imprimir();                                                   //1
        }
        if (interseccionBuscada.getColaOeste() != null) {                                                   //1
            System.out.println(x:"Cola oeste:");                                                            //1
            interseccionBuscada.getColaOeste().imprimir();                                                  //1
        }
    }
}
```

Justificación de complejidad:

Este método tiene la complejidad de O (1) ya que, dado un nombre de intersección, muestra sus atributos en la consola para el usuario.

- Método:

```java
private void generarBloqueo() {
    Scanner sn = new Scanner(System.in);                                                                    //1
    System.out.print(s:"Ingresa el nombre de la interseccion donde quieres generar el bloqueo: ");         //1
    String nombreInterseccion = sn.next();                                                                  //1
    Interseccion interseccionABloquear = obtenerInterseccion(nombreInterseccion);                           //1
    if (interseccionABloquear != null) {                                                                    //1
        System.out.println(x:"Bloqueando interseccion");                                                    //1
        interseccionABloquear.setBloqueda(bloqueda:true);                                                   //1
        String mensaje = "Bloqueando interseccion: " + interseccionABloquear.getNombre();                   //1
        System.out.println(mensaje);                                                                        //1
        registroEventos.push(mensaje);                                                                      //1
    }
}
```

Justificación de complejidad:

Este método tiene una complejidad de O(1) ya que recibido un nombre de intersección, bloquea la intersección cambiando el atributo booleano de la intersección a true.

- Método:

Documentación técnica – Proyecto 2
Rony Mauricio Rojas Aguilar – 202031191

```
private void agregarVehiculo() {
    Scanner sn = new Scanner(System.in);                                    //1
    System.out.println(x:"Ingresa los datos del vehiculo manualmente!!!");  //1
    System.out.print(s:"Ingresa el tipo de vehiculo: ");                      //1
    String tipo = sn.next();                                                //1
    TipoVehiculo tipoVehiculo = TipoVehiculo.valueOf(tipo);                 //1
    System.out.print(s:"Ingresa la placa del vehiculo: ");                    //1
    String placa = sn.next();                                               //1
    System.out.print(s:"Ingresa la interseccion de origen: ");                //1
    String interSeccionOrigen = sn.next();                                  //1
    System.out.print(s:"Ingresa la interseccion de destino: ");               //1
    String interseccionDestino = sn.next();                                 //1
    System.out.print(s:"Ingresa la prioridad: ");                             //1
    int prioridad = sn.nextInt();                                           //1
    System.out.print(s:"Ingresa el tiempo de espera: ");                      //1
    int tiempoDeEspera = sn.nextInt();                                      //1
    Vehiculo nuevoVehiculo = new Vehiculo(tipoVehiculo, placa, interSeccionOrigen, interseccionDestino, prioridad,
            tiempoDeEspera);                                                //1
    colocarVehiculo(nuevoVehiculo);                                         //1
}
```

Justificación de complejidad:
Este método tiene la complejidad de O(1) ya que solicita una vez los datos de cada vehículo y si son validos los datos, lo coloca en la intersección de origen.

- Método:

```
private void colocarVehiculo(Vehiculo vehiculo) {          Vehiculo vehiculo - com.mycompany.sistema_de_trafi
    Interseccion interseccionOrigen = obtenerInterseccion(vehiculo.getInterseccionOrigen());       //1
    Interseccion interseccionDestino = obtenerInterseccion(vehiculo.getInterseccionDestino());     //1

    if (interseccionOrigen != null && interseccionDestino != null) {                               //1
        PriorityQueueV cola = obtenerCola(interseccionOrigen);                                     //1
        if (cola != null) {                                                                        //1
            boolean insertado = tablaVehiculos.insertar(vehiculo);                                 //1
            if (!insertado) {                                                                      //1
                listaDuplicados.add(vehiculo);                                                     //1
            } else {                                                                               //1
                cola.insertar(vehiculo);                                                           //1
                interseccionOrigen.setVehiculosCirculados(interseccionOrigen.getVehiculosCirculados() + 1);
                vehiculo.setInterseccionActual(interseccionOrigen.getNombre());                    //1
                String mensaje = "Agregando vehiculo: " + vehiculo.toString();                     //1
                System.out.println(mensaje);                                                       //1
                registroEventos.push(mensaje);                                                     //1
                listaGeneralVehiculos.add(vehiculo);                                               //1
            }
        }
    }
}
```

Justificación de complejidad:

Este metodo tiene la complejidad de O (1), ya que dado un vehiculo, se realizan validaciones y si cumplen coloca al vehiculo en su posición de salida.

- Método:

```
private void verVehiculo() {
    Scanner sn = new Scanner(System.in);                                    //1
    System.out.print(s:"Ingresa la placa del vehiculo que quieres ver: ");  //1
    String placaABuscar = sn.next();                                        //1
    Vehiculo vehiculoBuscado = tablaVehiculos.buscar(placaABuscar);         //1
    if (vehiculoBuscado != null) {                                          //1
        System.out.println(vehiculoBuscado.toString());                     //1
    } else {
        System.out.println("Vehiculo con placa: " + placaABuscar + ", no existe."); //1
        return;                                                             //1
    }
}
```

Justificación de complejidad:

Este metodo tiene la complejidad de O (1) ya que al recibir la placa de un vehiculo, si el vehiculo existe en la tabla hash, muestra los atributos en consola.