Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

COMPLEJIDADES DE ALGORITMOS

CLASE NODE

- Método insertar:

```
template<typename T>
void LinkedList<T>::insertar(const T &data) {
    Node<T> *nuevoNodo = new Node<T>(data);          //1
    if (!head) {                                     //1
        head = nuevoNodo;                            //1
    } else {                                         //1
        Node<T> *temp = head;                        //1
        while (temp->getNext()) {                    //n
            temp = temp->getNext();                  //n
        }
        temp->setNext(nuevoNodo);                    //1
    }
    size++;                                          //1
}
```

Justificación de complejidad: O(n) esta complejidad se da cuando se recorre los elementos de una lista una cantidad de n veces en el peor de los casos y 1 vez en el mejor de los casos.

- Método eliminar:

```
template<typename T>
bool LinkedList<T>::eliminar(const T &data) {
    if (!head) {                                     //1
        return false;                                //1
    }
    if (head->getData() == data) {                   //1
        Node<T> *temp = head;                        //1
        head = head->getNext();                      //1
        delete temp;                                 //1
        size--;                                      //1
        return true;                                 //1
    }
    Node<T> *actual = head;                          //1
    while (actual->getNext() && actual->getNext()->getData() != data) { //n
        actual = actual->getNext();                                     //n
    }
    if (!actual->getNext()) {                        //1
        return false;                                //1
    }
    Node<T> *temp = actual->getNext();               //1
    actual->setNext(temp->getNext());                //1
    delete temp;                                     //1
    size--;                                          //1
    return true;                                     //1
}
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

Justificación de complejidad: O(n) al igual que el método anterior esta complejidad se da cuando se debe recorrer los elementos de una lista una cantidad de n veces hasta que el elemento a eliminar sea igual al elemento encontrado en x posición.

-Método Buscar:

```
template<typename T>
bool LinkedList<T>::buscar(const T &data) const {
    Node<T> *actual = head;                          //1
    while (actual) {                                 //n
        if (actual->getData() == data) return true; //n
        actual = actual->getNext();                  //n
    }
    return false;                                    //1
}
```

Justificación de complejidad: O(n) este método o algoritmo tiene esta complejidad dado que recorre una lista nodo por nodo una cantidad de n veces hasta que el elemento recibido sea igual al elemento que este en el nodo actual.

-Método obtenerPorIndice:

```
template<typename T>
Node<T> *LinkedList<T>::obtenerPorIndice(int indice) {
    if (indice < 0) {                                //1
        throw out_of_range("indice invalido");       //1
    }
    Node<T> *actual = head;                          //1
    int contador = 0;                                //1

    while (actual != nullptr) {                      //n
        if (contador == indice) {                    //n
            return actual;                           //n
        }
        actual = actual->getNext();                  //n
        contador++;                                  //n
    }
    throw out_of_range("indice fuera de rango");     //1
}
```

Justificación de complejidad: O(n) la complejidad de este método o algoritmo esta definida por el ciclo que se repite una cantidad de n veces en búsqueda del índice que es recibido para buscar el nodo que está en ese índice de la lista.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

-Método imprimir:

```
template<typename T>
void LinkedList<T>::imprimir() const {
    Node<T> *actual = head;                                  //1
    int i = 0;                                               //1
    while (actual) {                                         //n
        cout << i << ") " << actual->getData() << " \n";//n
        i++;                                                 //n
        actual = actual->getNext();                          //n
    }
}
```

Justificación de complejidad: O(n) esta complejidad esta definida por el recorrido de n veces que se le hace a una lista para ir imprimiendo cada nodo que pertenece a la lista.

CLASE BST

-Método insertar:

```
void insertar(const T &data) {
    // si el arbol esta vacio, crear un nuevo nodo como raiz
    if (raiz == nullptr) {                                   //1
        raiz = new Node<T>(data);                            //1
        return;                                              //1
    }
    Node<T> *actual = raiz;                                  //1
    while (true) {                                           //n
        if (data < actual->getData()) {                      //n
            // si el valor es menor, se guarda en el subarbol izq
            if (actual->getLeft() == nullptr) {              //n
                //si se llega al final del arbol se inserta
                actual->setLeft(new Node<T>(data));          //n
                break;                                       //n
            } else {
                // si no, se avanza al siguiente nodo izquierdo  //n
                actual = actual->getLeft();                  //n
            }
        } else {                                             //n
            // si el valor es mayor, se guarda en el subarbol der
            if (actual->getRight() == nullptr) {             //n
                //si se llega al final de arbol se inserta
                actual->setRight(new Node<T>(data));         //n
                break;                                       //n
            } else {
                // si no se avanza al siguiente subarbol derecho
                actual = actual->getRight();                 //n
            }
        }
    }
}
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

Justificación de complejidad: la complejidad O(n) está dada por el ciclo while que indica que se recorrerá el subárbol izquierdo o derecho en búsqueda de un espacio adecuado para que el dato se inserte.

-Método buscar:

```
bool buscar(const T &data) const {
    // se comienza desde la raiz
    Node<T> *arbol = raiz;                                                  //1
    while (arbol != nullptr) {                                              //n
        // guardando el valor de la raiz del arbol actual
        T valorRaiz = arbol->getData();                                     //n
        if (data == valorRaiz) {                                            //n
            // devolviendo true si el valor buscado esta en la raiz del arbol
            return true;                                                    //n
        } else if (data < valorRaiz) {                                      //n
            // si el valor es menor al valor de la raiz, se busca en el subarbol izquierdo
            arbol = arbol->getLeft();                                       //n
        } else {                                                            //n
            // si el valor es mayor al valor de la raiz, se busca en el subarbol derecho
            arbol = arbol->getRight();                                      //n
        }
    }
    // si se recorre el BST y no se encuentra el valor, se devuelve false
    return false;                                                           //n
}
```

Justificación de complejidad: Esta complejidad O(n) está definida también por el recorrido del árbol que puede tener una cantidad de n subárboles, esto en búsqueda del nodo que se ha recibido en la llamada del método.

-Método eliminar:

```
Node<T> *eliminarPrivado(Node<T> *arbol, const T &data) {
    if (arbol == nullptr) {                                                 //1
        // si el BST esta vacio, no se puede eliminar nada
        return nullptr;                                                     //1
    }
    if (data < arbol->getData()) {                                          //1
        // si el valor es menor que la raaz, se busca en el subarbol izquierdo
        arbol->setLeft(eliminarPrivado( arbol: arbol->getLeft(), data));    //log n
    } else if (data > arbol->getData()) {                                   //1
        // si el valor es mayor que la raiz, se busca en el subarbol derecho
        arbol->setRight(eliminarPrivado( arbol: arbol->getRight(), data));  //log n
    } else {                                                                //1
        // cuando el valor sea igual a la raiz, se elimina el nodo dependiendo del escenario
        if (arbol->getLeft() == nullptr && arbol->getRight() == nullptr) {  //1
            // escenario 1 - nodo sin hijos (solo se borra el nodo)
            delete arbol;                                                   //1
            return nullptr;                                                 //1
        } else if (arbol->getLeft() == nullptr) {                           //1
            // Escenario 2 - nodo solo con hijo derecho
            // Se guarda el subarbol derecho y se borra el arbol
            Node<T> *temp = arbol->getRight();                              //1
            delete arbol;                                                   //1
            return temp;                                                    //1
        } else if (arbol->getRight() == nullptr) {                          //1
            // escenario 2 - nodo solo con hijo izquierdo
            // se guarda el subarbol izquierdo y se borra el arbol
            Node<T> *temp = arbol->getLeft();                               //1
            delete arbol;                                                   //1
            return temp;                                                    //1
        } else {                                                            //1
            // escenario 3 - nodo con sus dos hijos
            // se encuentra el minimo del subarbol derecho
            Node<T> *temp = encontrarMinimoSubarbolDerecho( arbol: arbol->getRight());  //1
            // en la raiz del arbol se reemplaza el dato del minimo derecho
            arbol->setData(temp->getData());                                //1
            // eliminando el minimo de hasta abajo del árbol
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

Justificación de complejidad: este método tiene una complejidad de O (log n) dado que se aplica una recursión que envía el mismo árbol, pero con un tamaño reducido o a la mitad, esto en búsqueda de un nodo coincidente con el dato que nos han enviado.

-Método encontrarMinimoSubarbolDerecho:

```
Node<T> *encontrarMinimoSubarbolDerecho(Node<T> *arbol) const {
    // encontrando el minimo del arbol recibido
    while (arbol->getLeft() != nullptr) {                                    //n
        // mientras haya un subárbol izquierdo, se obtiene
        arbol = arbol->getLeft();                                            //n
    }
    // retornando el arbol donde ya no haya un subarbol izquierdo
    return arbol;                                                            //1
}
```

Justificación de complejidad: O(n) este método tiene una complejidad lineal dado que se hacer un recorrido de n subárboles hasta que se llegue al último nodo del árbol.

-Método imprimir:

```
void imprimirPrivado(Node<T> *nodo, int espacio){
    if (nodo == nullptr){                                                    //1
        return;                                                              //1
    }
    int sangria = 5;                                                         //1
    espacio += sangria;                                                      //1

    imprimirPrivado( nodo: nodo->getRight(), espacio);                       //log n

    cout << "\n";                                                            //1
    for (int i = sangria; i < espacio; i++) {                                //n
        cout << " ";                                                         //n
    }
    cout << nodo->getData() << "\n";                                         //1
    imprimirPrivado( nodo: nodo->getLeft(), espacio);                        //log n
}
```

Justificación de complejidad: O (log n) esta complejidad esta justificada en la llamada recursiva que se aplica de este método, pero en el momento que se hace esta llamada el árbol que se envía es cada vez mas pequeño, tanto del lado izquierdo como derecho, hasta que se llega a imprimir el árbol en su totalidad.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

CLASE THREEDIMENSIONALMATRIX

-Constructor:

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

```
ThreeDimensionalMatrix(int ancho, int alto, int profundidad) {
    if (ancho < 2 || alto < 2 || profundidad < 2) {                          //1
        throw invalid_argument("Las dimensiones deben ser al menos 2x2x2");  //1
    }
    this->ancho = ancho;                                                     //1
    this->alto = alto;                                                       //1
    this->profundidad = profundidad;                                         //1
    // creando nodo cabeza, esta en posicion (0, 0, 0)
    cabeza = new Node<T>(T());                                               //1
    Node<T> *nodoActual = cabeza;                                            //1

    //recorriendo cada columna, de cada fila, de cada profundidad
    for (int z = 0; z < profundidad; ++z) {                                  //n
        for (int y = 0; y < alto; ++y) {                                     //n²
            for (int x = 0; x < ancho; ++x) {                                //n³
                if (x == 0 && y == 0 && z == 0) {                            //n³
                    //saltando el nodo en la posicion (0, 0, 0), ya existe
                    continue;                                                //n³
                }
                // creando nuevo nodo en la posicion actual
                Node<T> *nuevoNodo = new Node<T>(T());                       //n³

                if (x > 0) {                                                 //n³
                    //si x >0, significa que hay un izquierdo
                    //al nuevo nodo se le manda el actual como izquierdo
                    nuevoNodo->setLeft(nodoActual);                          //n³
                    //al actual nodo se le manda el nuevoNodo como derecho
                    nodoActual->setRight(nuevoNodo);                         //n³
                    //actualizando nodoActual, para llevar la secuencia de la fila
                    nodoActual = nuevoNodo;                                  //n³
                } else {                                                     //n³
                    nodoActual = nuevoNodo;                                  //n³
                }
                if (y > 0) {                                                 //n³
                    //si y >0, significa que hay un nodo abajo, se obtiene
                    Node<T> *nodoAbajo = obtenerNodo(x,  y: y - 1, z);       //n³
                    //al nuevo nodo se le conecta al de abajo
                    nuevoNodo->setDown(nodoAbajo);                           //n³
                    //al de abajo se le conecta el arriba
                    nodoAbajo->setUp(nuevoNodo);                             //n³
                }
                if (z > 0) {                                                 //n³
                    //si z>0, significa que hay un nodo atras, se obtiene
                    Node<T> *nodoAtras = obtenerNodo(x, y,  z: z - 1);       //n³
                    //al nuevo nodo se le conecta al de atras
                    nuevoNodo->setPrev(nodoAtras);                           //n³
                    //al de atras se le conecta el de adelante
                    nodoAtras->setNext(nuevoNodo);                           //n³
                }
            }
```

Justificación de complejidad: O (n^3) este método tiene esta complejidad para todos los casos, ya que se recorre un for dentro de otro for que esta dentro de un último for (3 fors en total), cada for representa una dimensión para el tablero y en cada iteración se van conectando los nodos.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

-Método obtenerNodo:

```
Node<T> *obtenerNodo(int x, int y, int z) {
    //verificando limites
    if (x < 0 || y < 0 || z < 0 || x >= ancho || y >= alto || z >= profundidad) {   //1
        return nullptr;                                                             //1
    }
    Node<T> *actual = cabeza;                                                       //1
    //moviendo el actual hacia la derecha
    for (int i = 0; i < x && actual; ++i) {                                         //n
        actual = actual->getRight();                                                //n
    }
    //moviendo el actual hacia abajo
    for (int j = 0; j < y && actual; ++j) {                                         //n
        actual = actual->getUp();                                                   //n
    }
    //moviendo el actual hacia adelante
    for (int k = 0; k < z && actual; ++k) {                                         //n
        actual = actual->getNext();                                                 //n
    }
    return actual;                                                                  //1
}
```

Justificación de complejidad: La complejidad O (n) esta dada por los recorridos de los ciclos for que realizan cada iteración n veces dado una coordenada x, y o z.

-Método insertar:

```
void insertar(int x, int y, int z, T valor) {
    //obteniendo nodo en la posicion indicada
    Node<T> *nodo = obtenerNodo(x, y, z);                                           //1
    if (nodo) {                                                                     //1
        //si existe el nodo se le manda un nuevo valor
        nodo->setData(valor);                                                       //1
    } else throw out_of_range("Posición inválida");                                 //1
}
```

Justificación de complejidad: O (1) este método tiene esta complejidad constante ya que dadas unas coordenadas inserta un dato en el nodo que coincide con estas coordenadas.

-Método imprimir:

```
void imprimir() {
    cout << "********************" << endl;                                         //1
    for (int z = 0; z < profundidad; ++z) {                                         //n
        cout << "Tablero en z = " << z << ":\n";                                    //n
        for (int y = alto - 1; y >= 0; --y) {                                       //n²
            Node<T> *fila = obtenerNodo( x: 0, y, z);                               //n²
            for (int x = 0; x < ancho && fila; ++x) {                               //n³
                cout << fila->getData() << " ";                                     //n³
                fila = fila->getRight();                                            //n³
            }
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

Justificación de complejidad: Este método tiene una complejidad de O (n^3) ya que se hace un recorrido de 3 fors anidados y por cada iteración se hace una impresión en el nodo que se esté posicionado.

CLASE PARTIDA

-Método iniciarPartida:

```
void Partida::iniciarPartida() {
    tiempoPartida = time( timer: nullptr);                                          //1
    while (!jugadorEliminado && !tesoroEncontrado && !partidaAbandonada) {          //n
        tableroDeJuego->imprimir();                                                 //n
        realizarTurno( opcionTurno: jugador.mostrarOpcionesTurno());               //n
    }
    jugador.setTiempoJugado(time( timer: nullptr) - tiempoPartida);                 //1
    cout << jugador.getNombre() << ", tus estadisticas finales fueron: " << endl;  //1
    mostrarEstadisticas();                                                          //1
    cout << "Actualizando reportes..." << endl;                                     //1
}
```

Justificación de complejidad: Este método tiene una complejidad de O (n) dado que se repite un while n veces mientras que la partida no haya sido abandonada, el tesoro no haya sido encontrado o el jugador no haya sido eliminado.

-Método generarTablero:

```
void Partida::generarTablero() {
    tesoroX = rand() % ancho;                                                       //1
    tesoroY = rand() % alto;                                                        //1
    tesoroZ = rand() % profundidad;                                                 //1
    Tesoro tesoro;                                                                  //1
    tesoro.setPosicionX( posicion_x: tesoroX);                                      //1
    tesoro.setPosicionY( posicion_y: tesoroY);                                      //1
    tesoro.setPosicionZ( posicion_z: tesoroZ);                                      //1
    string ubicacionTesoro = "Tesoro ubicado en: (" + to_string( val: tesoroX) + ", " + to_string( val: tesoroY) + ", " +
                             to_string( val: tesoroZ) + ")";                        //1
    registroTrayectoria->insertar( data: ubicacionTesoro);                          //1
    tableroDeJuego->insertar( x: tesoroX,  y: tesoroY,  z: tesoroZ,  valor: tesoro);//1

    int jugadorX, jugadorY, jugadorZ;                                               //1
    do {
        //evitando que se cologue al jugador en el mismo lugar que el tesoro
        jugadorX = rand() % ancho;                                                  //n
        jugadorY = rand() % alto;                                                   //n
        jugadorZ = rand() % profundidad;                                            //n
    } while (jugadorX == tesoroX && jugadorY == tesoroY && jugadorZ == tesoroZ);    //n
    jugador.setPosicionX( posicion_x: jugadorX);                                    //1
    jugador.setPosicionY( posicion_y: jugadorY);                                    //1
    jugador.setPosicionZ( posicion_z: jugadorZ);                                    //1
    tableroDeJuego->insertar( x: jugadorX,  y: jugadorY,  z: jugadorZ,  valor: jugador);//1

    for (int z = 0; z < profundidad; z++) {                                         //n
        for (int y = 0; y < alto; y++) {                                            //n^2
            for (int x = 0; x < ancho; x++) {                                       //n^3
                if ((x == tesoroX && y == tesoroY && z == tesoroZ)
                    || x == jugadorX && y == jugadorY && z == jugadorZ) {           //n^3
                    //ignorando la posicion del tesoro y jugador
                    continue;                                                       //n^3
                }
```

```
                enemigoGenerado.setEfecto(enemigoGenerado.getVida());                                   //n^3
                tableroDeJuego->insertar(x, y, z, valor: enemigoGenerado);                              //n^3
                //enemigosPartida->insertar(enemigoGenerado.getUbicacion());                          //n^3
            } else if (random < 35) {                                                                   //n^3
                //20% de generacion para trampas
                Trampa trampaGenerada;                                                                  //n^3
                trampaGenerada.setPosicionX( posicion_x: x);                                             //n^3
                trampaGenerada.setPosicionY( posicion_y: y);                                             //n^3
                trampaGenerada.setPosicionZ( posicion_z: z);                                             //n^3
                string ubicacionTrampa = to_string( val: z) + to_string( val: y) + to_string( val: x);   //n^3
                trampaGenerada.setUbicacion(stoi( str: ubicacionTrampa));                               //n^3
                trampaGenerada.setEfecto(trampaGenerada.getDano());                                     //n^3
                tableroDeJuego->insertar(x, y, z, valor: trampaGenerada);                               //n^3
                //trampasPartida->insertar(trampaGenerada.getUbicacion());                            //n^3
            } else if (random < 60) {                                                                   //n^3
                //25% de generacion para pocimas
                Pocima pocimaGenerada;                                                                  //n^3
                pocimaGenerada.setPosicionX( posicion_x: x);                                             //n^3
                pocimaGenerada.setPosicionY( posicion_y: y);                                             //n^3
                pocimaGenerada.setPosicionZ( posicion_z: z);                                             //n^3
                pocimaGenerada.setEfecto(pocimaGenerada.getCuracion());                                 //n^3
                tableroDeJuego->insertar(x, y, z, valor: pocimaGenerada);                                //n^3
            } else {
                //40% de generacion para pistas
                Pista pistaGenerada;                                                                    //n^3
                pistaGenerada.setPosicionX( posicion_x: x);                                              //n^3
                pistaGenerada.setPosicionY( posicion_y: y);                                              //n^3
                pistaGenerada.setPosicionZ( posicion_z: z);                                              //n^3
                tableroDeJuego->insertar(x, y, z, valor: pistaGenerada);                                 //n^3
            }
        }
    }
}
```

Justificación de complejidad: Este método tiene una complejidad de O (n^3) ya que se recorre nuevamente los 3 fors anidados y en cada posición se generan entidades aleatorias, ya sea Enemigos, Trampas, Pócimas o Pistas, además del jugador y el tesoro.

-Método realizarTurno:

```
void Partida::realizarTurno(int opcionTurno) {
    switch (opcionTurno) {                                  //1
        case 1: {                                           //1
            cout << "Ingresa la direccion del movimiento:" << endl;  //1
            cout << "1. Arriba" << endl;                    //1
            cout << "2. Abajo" << endl;                     //1
            cout << "3. Derecha" << endl;                   //1
            cout << "4. Izquierda" << endl;                 //1
            cout << "5. Adelante" << endl;                  //1
            cout << "6. Atras" << endl;                     //1
            int direccion;                                  //1
            cin >> direccion;                               //1
            moverJugador(direccion);                        //1
            break;                                          //1
        }
        case 2: {                                           //1
            cout << "Tus estadisticas son:" << endl;        //1
            mostrarEstadisticas();                          //1
            break;                                          //1
        }
        case 3: {                                           //1
            cout << "Abandonando la partida..." << endl;    //1
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

Justificación de complejidad: este método tiene la complejidad de O (1) ya que todo el algoritmo se ejecuta una sola vez, ya que el jugador solo puede hacer una acción por turno.

-Método moverjugador:

```
void Partida::moverJugador(int direccion) {
    int nuevoX = jugador.getPosicionX();                                                                    //1
    int nuevoY = jugador.getPosicionY();                                                                    //1
    int nuevoZ = jugador.getPosicionZ();                                                                    //1

    string direccionMovimiento = "";                                                                        //1
        cout << "Has encontrado el tesoro, felicidades!!!" << endl;                                         //1
        jugador.setMovimientos(jugador.getMovimientos() + 1);                                              //1
        casillaEncontrada = " (" + to_string( val: casillaDestino.getPosicionX()) + ", " +
                            to_string( val: casillaDestino.getPosicionY()) + ", " + to_string(
                                val: casillaDestino.getPosicionZ()) + ")";                                  //1
        string movimientoFinal = "El tesoro fue encontrado con un movimiento hacia: " + direccionMovimiento +
                                 ", en la posicion" +
                                 casillaEncontrada + ", otorgandole al jugador: 100 pts!!!";                //1
        registroTrayectoria->insertar( data: movimientoFinal);                                             //1
        return;                                                                                             //1
      }
      case 6: {                                                                                             //1
        cout << "Estas a salvo en esta casilla, no hay nada." << endl;                                      //1
        break;                                                                                              //1
      }
      default: {                                                                                            //1
        cout << "errorrrrrrr .-." << endl;                                                                  //1
        break;                                                                                              //1
      }
    }
    Casilla casillaVacia;                                                                                   //1
    casillaVacia.setTipoCasilla("Vacia");
    tableroDeJuego->insertar( x: jugador.getPosicionX(),  y: jugador.getPosicionY(),  z: jugador.getPosicionZ(),  valor: casillaVacia); //1
    jugador.setPosicionX( posicion_x: nuevoX);                                                               //1
    jugador.setPosicionY( posicion_y: nuevoY);                                                               //1
    jugador.setPosicionZ( posicion_z: nuevoZ);                                                               //1
    tableroDeJuego->insertar( x: nuevoX,  y: nuevoY,  z: nuevoZ,  valor: jugador);                           //1
    if (jugador.getVida() <= 0) {                                                                           //1
        jugador.setVida(0);                                                                                 //1
        jugadorEliminado = true;                                                                            //1
        cout << "Tu vida ha llegado a 0, has perdido la partida :(" << endl;                                //1
    } else {                                                                                                //1
        cout << "Aun sigues con vida, continua explorando..." << endl;                                      //1
    }
    jugador.setMovimientos(jugador.getMovimientos() + 1);                                                   //1

    string trayectoria = "Movimiento en direccion hacia: " + direccionMovimiento + ", a la posicion: " + "(" +
                         to_string( val: casillaDestino.getPosicionX()) + ", " +
                         to_string( val: casillaDestino.getPosicionY()) + ", " + to_string(
                             val: casillaDestino.getPosicionZ()) + ")";                                     //1
    registroTrayectoria->insertar( data: trayectoria);                                                      //1
}
```

Justificación de complejidad: este método tiene también una complejidad de O (1) ya que el movimiento del jugador a través del mapa se hace un paso a la vez, así mismo como las interacciones del jugador con la entidad que este en la casilla a donde se movió el jugador.

CLASE MOTORDEJUEGO

-Método cargarJugadores:

```
void MotorDeJuego::cargarJugadores() {
    string nombreArchivo;                                                    //1
    cout << "\nIngresa el nombre del archivo.csv: ";                         //1
    cin >> nombreArchivo;                                                    //1

    ifstream archivo( s: nombreArchivo);                                     //1

    if (!archivo.is_open()) {                                                //1
        cout << "No se pudo abrir el archivo " << nombreArchivo << endl;     //1
        return;                                                              //1
    }
    string linea;                                                            //1
    while (getline( &: archivo,  &: linea)) {                                //n
        stringstream ss( str: linea);                                        //n
        string nombre;                                                       //n
        string puntosStr;                                                    //n
        string movimientosStr;                                               //n

        getline( &: ss,  &: nombre,  delim: ',');                            //n
        getline( &: ss,  &: puntosStr,  delim: ',');                         //n
        getline( &: ss,  &: movimientosStr);                                 //n

        try {
            int puntos = stoi( str: puntosStr);                              //n
            int movimientos = stoi( str: movimientosStr);                    //n

            Jugador nuevoJugador(nombre);                                    //n
            nuevoJugador.setPuntos(puntos);                                  //n
            nuevoJugador.setMovimientos(movimientos);                        //n
            reporte.agregarJugador( jugador: nuevoJugador);                  //n
        } catch (const exception& e) {                                       //n
            cout << "Error al procesar linea: " << linea << " - " << e.what() << endl;  //n
        }
    }
    archivo.close();                                                         //1
    cout << "Se cargo a los jugadores desde el archivo" << endl;             //1
}
```

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
Justificación de complejidad: Este método tiene una complejidad de O (n) dado que se lee una línea una cantidad de n veces, hasta que el archivo ya no tenga líneas para leer la información.

CLASE REPORTE

-Método mostrarMenuReporteJugadores:

```
void Reporte::mostrarMenuReportesPartidas(int indiceJugador) {
    Node<Partida> *actual = partidas.obtenerPorIndice( indice: indiceJugador);                    //1
    if (!actual) {                                                                                //1
        cout << "No hay partida con este indice" << endl;                                         //1
        return;                                                                                   //1
    }
    int opcionPartida = 0;                                                                        //1
    do {
        cout << "\nSelecciona el reporte que quieres ver del jugador: " << actual->getData().getJugador().getNombre() <<
                endl;                                                                             //n
        cout << "1. Nombre del jugador, tiempo total, movimientos y puntuacion." << endl;         //n
        cout << "2. Ubicacion del tesoro y trayectora del jugador." << endl;                      //n
        cout << "3. Pistas encontradas y su distancia al tesoro." << endl;                        //n
        cout << "4. Enemigos enfrentados y tramapas activadas." << endl;                          //n
        cout << "5. Grafico de los arboles de enemigos y trampas." << endl;                       //n
        cout << "6. Regresar al menu de reportes" << endl;                                        //n
        cout << "Selecciona una opcion... ";                                                      //n
        cin >> opcionPartida;                                                                     //n
        cout << "-------------------------------------" << endl;                                 //n
        switch (opcionPartida) {                                                                  //n
            case 1: {                                                                             //n
                cout << "\nNombre del jugador: " << actual->getData().getJugador().getNombre() << endl;          //n
                cout << "Tiempo total: " << actual->getData().getJugador().getTiempoJugado() << " s" << endl;    //n
                cout << "Movimientos: " << actual->getData().getJugador().getMovimientos() << endl;              //n
                cout << "Puntuacion: " << actual->getData().getJugador().getPuntos() << endl;                    //n
                if (actual->getData().getJugador().getEncontroTesoro()){                                        //n
                    cout << "Encontro tesoro: Si"<< endl;                                                        //n
                } else {                                                                                         //n
                    cout << "Encontro tesoro: No"<< endl;                                                        //n
                }
                break;                                                                            //n
            }
            case 4: {                                                                             //n
                cout << "\nEnemigos enfrentados y trampas activadas: " << endl;                   //n
                actual->getData().getRegistroEnemigosYTrampas()->imprimir();                      //n
                break;                                                                            //n
            }
            case 5: {                                                                             //n
                cout << "\nGrafico de los arboles de enemigos y trampas:" << endl;                //n
                cout << "Grafico de arboles de enemigos (nivel alto a nivel bajo)" << endl;       //n
                actual->getData().getEnemigosPartida()->imprimir();
                cout << "Grafico de arboles de trampas (nivel bajo a nivel alto)" << endl;        //n
                actual->getData().getTrampasPartida()->imprimir();
                break;                                                                            //n
            }
            case 6: {                                                                             //n
                return;                                                                           //n
            }
            default: {                                                                            //n
                cout << "Opcion no valida." << endl;                                              //n
                break;                                                                            //n
            }
        }
        cout << "\n-------------------------------------" << endl;                               //n
    } while (opcionPartida != 6);                                                                 //n
}
```

Justificación de complejidad: Este método tiene una complejidad de O (n) ya que las opciones de los reportes que el jugador puede acceder se repiten n veces hasta que el jugador desee regresar al menú principal.

-Método mostrarTablaJugadores:

```
void Reporte::mostrarTablaJugadores() {
    if (!tablaJugadores.getCabeza()) {                                          //1
        cout << "No hay jugadores existentes" << endl;                          //1
        return;                                                                 //1
    }
    ordenarTablaJugadores();                                                    //1
    Node<Jugador> *actual = tablaJugadores.getCabeza();                         //1
    cout << "\n--- Tabla de Jugadores ---\n";                                   //1
    int indice = 0;                                                             //1
    while (actual) {                                                            //n
        indice++;                                                               //n
        cout << indice << ") Nombre: " << actual->getData().getNombre() << endl;//n
        cout << "   Puntuacion: " << actual->getData().getPuntos() << endl;     //n
        cout << "   Vida: " << actual->getData().getVida() << endl;             //n
        cout << "   Movimientos: " << actual->getData().getMovimientos() << endl;//n
        cout << "   Tiempo jugado: " << actual->getData().getTiempoJugado() << " s" << endl;//n
        if (actual->getData().getEncontroTesoro()){                             //n
            cout << "   Encontro el teosoro: Si" <<endl;                        //n
        } else {                                                                //n
            cout << "   Encontro el teosoro: No" <<endl;                        //n
        }
        cout << "-------------------------\n";                                 //n
        actual = actual->getNext();                                             //n
    }
}
```

Justificación de complejidad: este método tiene la complejidad de O (n) dado que realiza la impresión de datos de los jugadores n veces, dependiendo de la cantidad de jugadores que haya registrados en la ejecución actual.

-Método ordenarTablaJugadores:

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos

```
void Reporte::ordenarTablaJugadores() {
    if (!tablaJugadores.getCabeza()) {                                          //1
        return;                                                                 //1
    }
    bool intercambio;                                                           //1
    do {
        intercambio = false;                                                    //n
        Node<Jugador> *actual = tablaJugadores.getCabeza();                     //n

        while (actual->getNext() != nullptr) {                                  //n^2
            if (actual->getData().getPuntos() < actual->getNext()->getData().getPuntos()) {     //n^2
                Jugador temp = actual->getData();                               //n^2
                actual->setData(actual->getNext()->getData());                  //n^2
                actual->getNext()->setData(temp);                               //n^2
                intercambio = true;                                             //n^2
            }
            actual = actual->getNext();                                         //n
        }
    } while (intercambio);                                                      //n
}
```

Justificación de complejidad: este método de ordenamiento (Bubble-Sort Mejorado) tiene la complejidad de O (n^2) dado que por cada elemento de la lista recorre una vez la lista ordenando los valores por los puntos, hace una iteración de la lista por cada elemento de la lista que se está iterando.