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

COMPLEJIDADES DE ALGORITMOS
CLASE LINKEDLIST

- 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
            }
        }
    }
}
```

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 imprimir:

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

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

    cout << "\n";                                                  //1
    for (int i = sangria; i < espacio; i++) {                      //n
        cout << " ";                                               //n
    }
    cout << nodo->getData() << "\n";                               //1
    imprimirDescendente( 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.

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.

```
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
}
```

-Método obtenerNodo:

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
            }
            cout << "\n";                                                           //n²
        }
```

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() {
    this->generarTesoro();          //1
    this->generarJugador();         //1
    this->generarObjetosTablero();  //1
}
```

Justificación de complejidad: este método es de complejidad constante porque cada acción se realiza una sola vez por cada ejecución de la instancia de la clase partida.

-Metodo generarTesoro:

```
void Partida::generarTesoro() {
    tesoroX = rand() % ancho;                                                   //1
    tesoroY = rand() % alto;                                                    //1
    tesoroZ = rand() % profundidad;                                             //1

    Tesoro tesoro;                                                              //1
    tesoro.setPosicionX(tesoroX);                                               //1
    tesoro.setPosicionY(tesoroY);                                               //1
    tesoro.setPosicionZ(tesoroZ);                                               //1

    string ubicacionTesoro = "Tesoro ubicado en: (" + to_string(tesoroX) + ", " +
                             to_string(tesoroY) + ", " + to_string(tesoroZ) + ")";  //1
    registroTrayectoria->insertar(ubicacionTesoro);                             //1
    tableroDeJuego->insertar(tesoroX, tesoroY, tesoroZ, tesoro);                //1
}
```

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

Justificacion de complejidad: este método es de complejidad constante porque cada acción se realiza una sola vez por cada ejecución de la instancia de la clase partida, cuando se genera un tesoro.

-Metodo generarJugador:

```
void Partida::generarJugador() {
    int jugadorX, jugadorY, jugadorZ;                                          //1
    do {
        jugadorX = rand() % ancho;                                             //n
        jugadorY = rand() % alto;                                              //n
        jugadorZ = rand() % profundidad;                                       //n
    } while (jugadorX == tesoroX && jugadorY == tesoroY && jugadorZ == tesoroZ);    //n

    jugador.setPosicionX(jugadorX);                                            //1
    jugador.setPosicionY(jugadorY);                                            //1
    jugador.setPosicionZ(jugadorZ);                                            //1
    tableroDeJuego->insertar(jugadorX, jugadorY, jugadorZ, jugador);           //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada acción se realiza una sola vez por cada ejecución de la instancia de la clase partida, cuando se genera al jugador de la partida.

-Metodo generarObjetosTablero:

```
void Partida::generarObjetosTablero() {
    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 == tesoroX && y == tesoroY && z == tesoroZ) ||          //n³
                    (x == jugador.getPosicionX() && y == jugador.getPosicionY() && z == jugador.getPosicionZ())) {   //n³
                    continue;                                                  //n³
                }
                int random = rand() % 100;                                     //n³
                if (random < 15) {                                             //n³
                    generarEnemigo(x, y, z);                                   //n³
                } else if (random < 35) {                                      //n³
                    generarTrampa(x, y, z);                                    //n³
                } else if (random < 60) {                                      //n³
                    generarPocima(x, y, z);                                    //n³
                } else {                                                       //n³
                    generarPista(x, y, z);                                     //n³
                }
            }
        }
    }
}
```

Justificacion de complejidad: este método tiene una complejidad de O(n^3) ya que se hace un recorrido de un for dentro de otro for que esta dentro de otro for, haciendo 3 fors anidados.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
-Metodo generarEnemigo:

```
void Partida::generarEnemigo(int x, int y, int z) {
    Enemigo enemigoGenerado;                                    //1
    enemigoGenerado.setPosicionX(x);                            //1
    enemigoGenerado.setPosicionY(y);                            //1
    enemigoGenerado.setPosicionZ(z);                            //1
    enemigoGenerado.setEfecto(enemigoGenerado.getVida());       //1
    tableroDeJuego->insertar(x, y, z, enemigoGenerado);      //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada acción se un único enemigo en la casilla con coordenadas que se han enviado por parametro.

-Metodo 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
            partidaAbandonada = true;                           //1
            break;                                              //1
        }
        default: {                                              //1
            cout << "Ingresa una opcion valida";                //1
            break;                                              //1
        }
    }
}
```

Justificacion de complejidad: este método es de complejidad constante el jugador se puede mover a una sola dirección por cada turno que este tenga.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
-Metodo moverJugador:

```
void Partida::moverJugador(int tipoMovimiento) {
    int nuevoX, nuevoY, nuevoZ;                                                                    //1
    string direccion;                                                                              //1
    calcularPosicionMovimiento(tipoMovimiento, nuevoX, nuevoY, nuevoZ, direccion);                 //1

    if (direccion == "invalida") {                                                                 //1
        return;                                                                                    //1
    }
    if (nuevoX < 0 || nuevoX >= ancho || nuevoY < 0 || nuevoY >= alto || nuevoZ < 0 || nuevoZ >= profundidad) {     //1
        cout << "Movimiento fuera de los limites del tablero" << endl;                             //1
        return;                                                                                    //1
    }

    Node<Casilla>* nodoDestino = tableroDeJuego->obtenerNodo(nuevoX, nuevoY, nuevoZ);              //1
    if (!nodoDestino) {                                                                            //1
        cout << "Nodo no encontrado" << endl;                                                      //1
        return;                                                                                    //1
    }
    Casilla& casillaDestino = nodoDestino->getData();                                              //1

    procesarEfectoCasilla(casillaDestino, direccion);                                          //1

    actualizarPosicionJugador(nuevoX, nuevoY, nuevoZ);                                             //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque el movimiento del jugador se hace de casilla en casilla una a la vez por cada turno que este tenga.

-Metodo calcularPosicionMovimiento:

```
void Partida::calcularPosicionMovimiento(int tipoMovimiento, int &nuevoX, int &nuevoY, int &nuevoZ, string &direccion) {
    nuevoX = jugador.getPosicionX();                //1
    nuevoY = jugador.getPosicionY();                //1
    nuevoZ = jugador.getPosicionZ();                //1
    direccion = "";                                 //1

    switch (tipoMovimiento) {                       //1
        case 1: // Arriba                           //1
            nuevoY++;                               //1
            direccion = "arriba";                   //1
            break;                                  //1
        case 2: // Abajo                            //1
            nuevoY--;                               //1
            direccion = "abajo";                    //1
            break;                                  //1
        case 3: // Derecha                          //1
            nuevoX++;                               //1
            direccion = "derecha";                  //1
            break;                                  //1
        case 4: // Izquierda                        //1
            nuevoX--;                               //1
            direccion = "izquierda";                //1
            break;                                  //1
        case 5: // Adelante                         //1
            nuevoZ++;                               //1
            direccion = "adelante";                 //1
            break;                                  //1
        case 6: // Atras                            //1
            nuevoZ--;                               //1
            direccion = "atras";                    //1
            break;                                  //1
        default:                                    //1
            cout << "movimiento incorrecto" << endl;//1
            direccion = "no permitida";             //1
    }
}
```

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

Justificacion de complejidad: este método es de complejidad constante porque el avance de coordenadas del jugador se hace una a la vez.

-Metodo procesarEfectoCasilla:

```
void Partida::procesarEfectoCasilla(Casilla &casilla, const string &direccion) {
    const string& tipoCasilla = casilla.getTipoCasilla();                  //1

    if (tipoCasilla == "Enemigo") {                                        //1
        procesarEnemigo(casilla);                                         //1
    } else if (tipoCasilla == "Trampa") {                                  //1
        procesarTrampa(casilla);                                          //1
    } else if (tipoCasilla == "Pocima") {                                  //1
        procesarPocima(casilla);                                          //1
    } else if (tipoCasilla == "Pista") {                                   //1
        procesarPista(casilla);                                           //1
    } else if (tipoCasilla == "Tesoro") {                                  //1
        procesarTesoro(casilla, direccion);                               //1
    } else if (tipoCasilla == "Vacia") {                                   //1
        cout << "Estás a salvo en esta casilla, no hay nada." << endl;     //1
    }
    if (tipoCasilla != "Tesoro") {                                         //1
        registrarMovimiento(casilla, direccion);                           //1
    }

}
```

Justificacion de complejidad: este método es de complejidad constante porque en cada casilla solo hay un posible objeto y se procesa solo este.

-Metodo procesarEnemigo:

```
void Partida::procesarEnemigo(Casilla &casilla) {
    jugador.setVida(jugador.getVida() - casilla.getEfecto());                          //1
    jugador.setPuntos(jugador.getPuntos() + 15);                                       //1
    cout << "Encontraste un enemigo D:!! pierdes " << casilla.getEfecto()
         << " puntos de vida." << endl;                                                //1
    cout << "Y ganas 15 puntos!!!" << endl;                                            //1

    string mensaje = "Se ha encontrado un enemigo y ha realizado: " +
                     to_string(casilla.getEfecto()) + " de daño al jugador, en la posición: (" +
                     to_string(casilla.getPosicionX()) + ", " +
                     to_string(casilla.getPosicionY()) + ", " +
                     to_string(casilla.getPosicionZ()) + ")";                          //1

    registroEnemigosYTrampas->insertar(mensaje);                                       //1
    enemigosPartida->insertar(casilla.getUbicacion());                                //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque se procesa solo un enemigo, trampa, pócima o pista por cada cambio de casilla que haga el jugador.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
-Metodo registrarMovimiento:

```
void Partida::registrarMovimiento(const Casilla &casilla, const string &direccion) {
    string trayectoria = "Movimiento en dirección hacia: " + direccion +
                         ", a la posición: (" + to_string(casilla.getPosicionX()) +
                         ", " + to_string(casilla.getPosicionY()) + ", " +
                         to_string(casilla.getPosicionZ()) + ")";                    //1

    registroTrayectoria->insertar(trayectoria);                                      //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque se registra un movimiento a la vez por cada turno que el jugador haga.

-Metodo actualizarPosicionJugador:

```
void Partida::actualizarPosicionJugador(int nuevoX, int nuevoY, int nuevoZ) {
    Casilla casillaVacia;                                                //1
    casillaVacia.setTipoCasilla("Vacia");                               //1
    tableroDeJuego->insertar(jugador.getPosicionX(),
                             jugador.getPosicionY(),
                             jugador.getPosicionZ(),
                             casillaVacia);                              //1

    jugador.setPosicionX(nuevoX);                                        //1
    jugador.setPosicionY(nuevoY);                                        //1
    jugador.setPosicionZ(nuevoZ);                                        //1
    tableroDeJuego->insertar(nuevoX, nuevoY, nuevoZ, jugador);          //1

    jugador.setMovimientos(jugador.getMovimientos() + 1);                //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
    }
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada que el jugador se mueve una casilla a cualquier dirección se registra un único movimiento.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
-Metodo reemplazarCasillaVacia:

```
void Partida::reemplazarCasillaVacia(int nuevoX, int nuevoY, int nuevoZ) {
    Casilla casillaVacia;                                                                   //1
    casillaVacia.setTipoCasilla("Vacia");                                                   //1
    tableroDeJuego->insertar(jugador.getPosicionX(), jugador.getPosicionY(), jugador.getPosicionZ(), casillaVacia);  //1
    jugador.setPosicionX(nuevoX);                                                           //1
    jugador.setPosicionY(nuevoY);                                                           //1
    jugador.setPosicionZ(nuevoZ);                                                           //1
    tableroDeJuego->insertar(nuevoX, nuevoY, nuevoZ, 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
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada que el jugador abandona una casilla, se crea una sola casilla para reemplazarlo en su lugar.

-Metodo calcularDistanciaPista:

```
int Partida::calcularDistanciaPista(int nuevoX, int nuevoY, int nuevoZ) {
    int distanciaX = nuevoX - tesoroX;                                  //1
    int distanciaY = nuevoY - tesoroY;                                  //1
    int distanciaZ = nuevoZ - tesoroZ;                                  //1
    //convirtiendo distancias a valores positivos (si los son)
    if (distanciaX < 0) {                                               //1
        distanciaX *= -1;                                               //1
    }
    if (distanciaY < 0) {                                               //1
        distanciaY *= -1;                                               //1
    }
    if (distanciaZ < 0) {                                               //1
        distanciaZ *= -1;                                               //1
    }
    //calculando distancia de posicion actual a la del tesoro
    int distancia = distanciaX + distanciaY + distanciaZ;               //1

    cout << "Encontraste una Pista :D, esta dice:" << endl;             //1
    return distancia;                                                   //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada que el jugador encuentra una pista se calcula la distancia de la pista con respecto al tesoro una sola vez.

-Metodo mostrarEstadisticas:

```
void Partida::mostrarEstadisticas() {
    cout << "Vida: " << jugador.getVida() << endl;                                          //1
    cout << "Puntos: " << jugador.getPuntos() << endl;                                      //1
    cout << "Movimientos: " << jugador.getMovimientos() << endl;                            //1
    cout << "Tiempo Jugado: " << jugador.getTiempoJugado() << " s" << endl;                 //1
    cout << "Ubicacion: (" << jugador.getPosicionX() << ", " << jugador.getPosicionY() << ", " << jugador.getPosicionZ()
         << ")" << endl;                                                                    //1
}
```

Justificacion de complejidad: este método es de complejidad constante porque cada atributo del jugador se muestra una vez por cada solicitud de estadísticas.

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

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
}
```

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.

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

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
}
```

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

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.

Rony Mauricio Rojas Aguilar – 202031191
Proyecto 1 – Estructura de Datos
-Metodo mostrarTablaJugadores:

```
void Reporte::mostrarTablaJugadores(int opcionTabla) {
    if (!tablaJugadores.getCabeza()) {                                          //1
        cout << "No hay jugadores existentes" << endl;                          //1
        return;                                                                 //1
    }
    if (opcionTabla == 1) {                                                     //1
        //ordenando tabla de jugadores por nombre
        ordenarTablaJugadores(1);                                               //1
        cout << "Mostrando tabla ordenada por nombre (A-Z):" << endl;           //1
    } else if (opcionTabla == 2) {                                              //1
        //ordenando tabla de jugadores por puntuacion
        ordenarTablaJugadores(2);                                               //1
        cout << "Mostrando tabla ordenada por puntuacion (mayor a menor):" << endl;  //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
    }
}
```

Justificacion de complejidad: este método tiene la complejidad O(n) ya que muestra una lista de jugadores con todos sus atributos, dicha lista puede tener mas de un jugador en su interior, lo que hace que la dificultad en el peor de los casos llegue a ser lineal.

-Método ordenarTablaJugadores:

```
void Reporte::ordenarTablaJugadores(int opcionTabla) {
    if (!tablaJugadores.getCabeza()) {                                          //1
        return;                                                                 //1
    }
    bool intercambio;                                                           //1
    do {
        intercambio = false;                                                    //n
        Node<Jugador> *actual = tablaJugadores.getCabeza();                     //n
        if (opcionTabla==1) {                                                   //n
            intercambio = ordenarPorNombre(actual);                             //n
        } else if (opcionTabla==2) {                                            //n
            intercambio = ordenarPorPunteo(actual);                             //n
        }
    } while (intercambio);                                                      //n
}
```

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

Justificación de complejidad: este método tiene la complejidad de O(n) ya que cicla un while n veces hasta que la lista de jugadores este ordenada en su totalidad, lo que hace que tenga una complejidad lineal.

-Metodo ordenarPorNombre:

```
bool Reporte::ordenarPorNombre(Node<Jugador> *actual) {
    bool intercambio = false;                                                    //n
    while (actual->getNext() != nullptr) {                                       //n^2
        if (actual->getData().getNombre() > actual->getNext()->getData().getNombre()) {     //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
    }
    return 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 el nombre del jugador, hace una iteración de la lista por cada elemento de la lista que se está iterando.

-Metodo ordenarPorPunteo:

```
bool Reporte::ordenarPorPunteo(Node<Jugador> *actual) {
    bool intercambio = false;                                                    //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
    }
    return 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 el nombre del jugador, hace una iteración de la lista por cada elemento de la lista que se está iterando.