# Segment Tree: Estructura de Datos para Consultas de Rango

## Introducción

Un Segment Tree (Árbol de Segmentos) es una estructura de datos avanzada que permite realizar consultas eficientes sobre intervalos y actualizaciones de segmentos en un arreglo. Esta estructura es particularmente útil en situaciones donde necesitamos realizar consultas repetidas de suma, mínimo, máximo, o cualquier otra operación asociativa sobre un rango de elementos en un arreglo.

## Descripción del Algoritmo

Un Segment Tree se construye dividiendo el arreglo original en segmentos y almacenando la información sobre estos segmentos en nodos de un árbol binario. Cada nodo del árbol representa un intervalo (o segmento) del arreglo y almacena el resultado de una operación (como suma o mínimo) para ese intervalo.

Las operaciones clave que se pueden realizar en un Segment Tree son:

1. **Construcción**: Crear el Segment Tree a partir de un arreglo dado.
2. **Consulta de Rango**: Obtener el resultado de una operación (suma, mínimo, etc.) en un intervalo dado.
3. **Actualización**: Modificar un valor en el arreglo y actualizar el Segment Tree en consecuencia.

## Implementación en C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

class SegmentTree {
    vector<int> tree;
    int n;

    void buildTree(const vector<int>& arr, int start, int end, int node) {
        if (start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            buildTree(arr, start, mid, 2 * node + 1);
            buildTree(arr, mid + 1, end, 2 * node + 2);
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    int rangeQuery(int start, int end, int l, int r, int node) {
        if (l > end || r < start) {
            return 0; // rango fuera del intervalo actual
        }
```

```
        if (l <= start && r >= end) {
            return tree[node]; // rango completamente dentro del intervalo
        }
        int mid = (start + end) / 2;
        return rangeQuery(start, mid, l, r, 2 * node + 1) + rangeQuery(mid + 1, end,
l, r, 2 * node + 2);
    }

    void updateTree(int start, int end, int idx, int value, int node) {
        if (start == end) {
            tree[node] = value;
        } else {
            int mid = (start + end) / 2;
            if (idx <= mid) {
                updateTree(start, mid, idx, value, 2 * node + 1);
            } else {
                updateTree(mid + 1, end, idx, value, 2 * node + 2);
            }
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

public:
    SegmentTree(const vector<int>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        buildTree(arr, 0, n - 1, 0);
    }

    int rangeQuery(int l, int r) {
        return rangeQuery(0, n - 1, l, r, 0);
    }

    void update(int idx, int value) {
        updateTree(0, n - 1, idx, value, 0);
    }
};

int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11};
    SegmentTree st(arr);

    cout << "Suma del rango (1, 3): " << st.rangeQuery(1, 3) << endl;

    st.update(1, 10);
    cout << "Suma del rango (1, 3) después de actualizar: " << st.rangeQuery(1, 3) <<
endl;

    return 0;
}
```

**Ejemplo de Salida**

```
Suma del rango (1, 3): 15
Suma del rango (1, 3) después de actualizar: 22
```

## Ejemplos de Aplicaciones del Segment Tree

1. **Sistemas de Consulta en Bases de Datos**: Segment Tree se puede utilizar para manejar consultas sobre intervalos en bases de datos, como la suma de un rango de filas.
2. **Compresión de Datos**: En algoritmos de compresión de datos, un Segment Tree puede ayudar a calcular eficientemente la suma o el valor mínimo/máximo en segmentos de datos.
3. **Procesamiento de Imágenes**: Segment Tree puede ser útil en el procesamiento de imágenes donde se necesitan realizar operaciones rápidas en intervalos específicos de píxeles, como encontrar el valor máximo en una subregión de la imagen.

## Conclusión

El Segment Tree es una estructura de datos versátil y poderosa que permite realizar consultas y actualizaciones en intervalos de manera eficiente. Es particularmente útil en aplicaciones donde las operaciones sobre rangos de datos son frecuentes y críticas en términos de rendimiento.