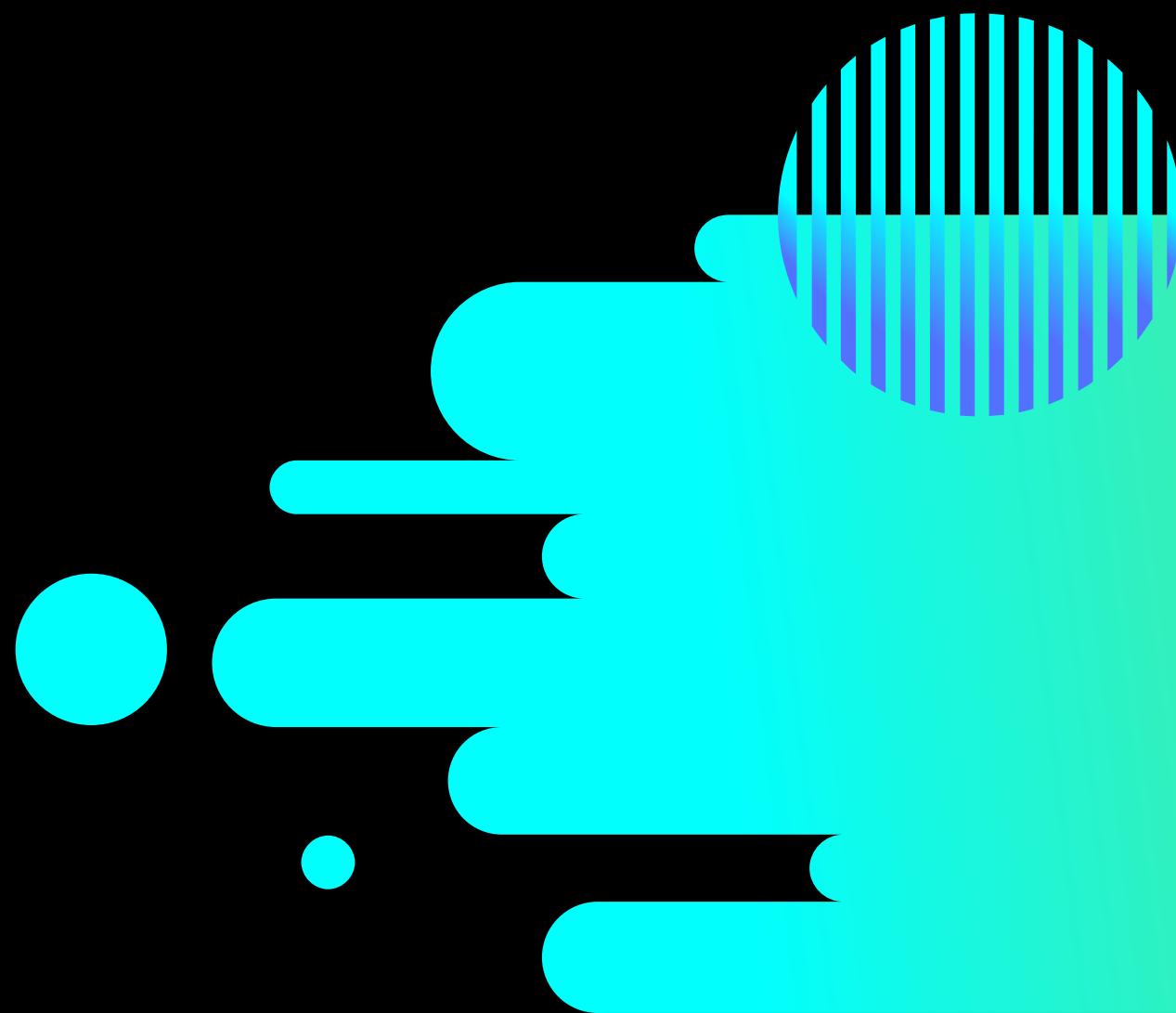
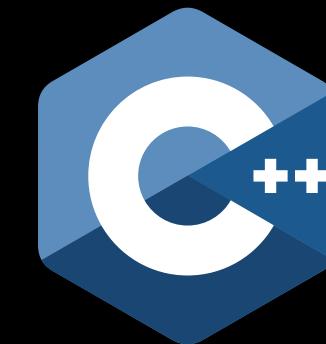
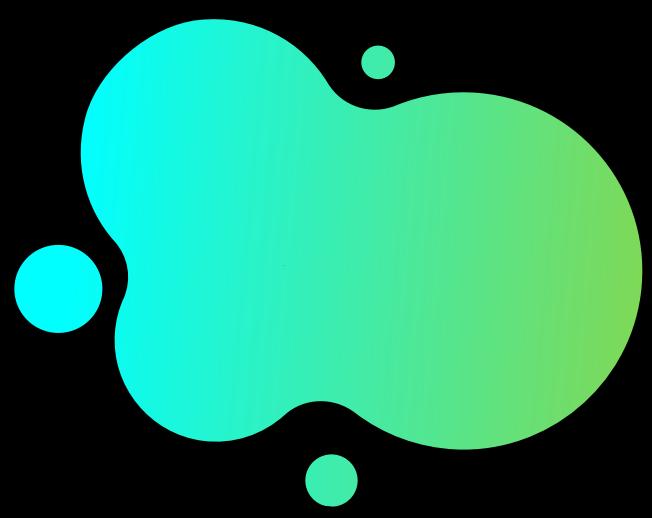


SEGMENT TREE

Análisis y Aplicaciones en Algoritmos
y Estructuras de Datos

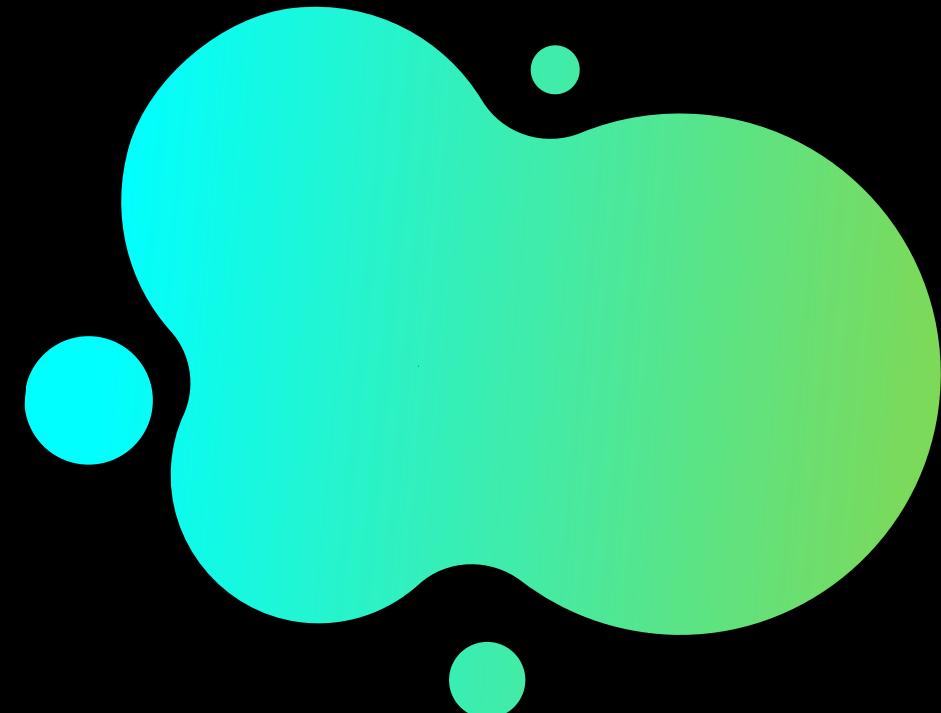
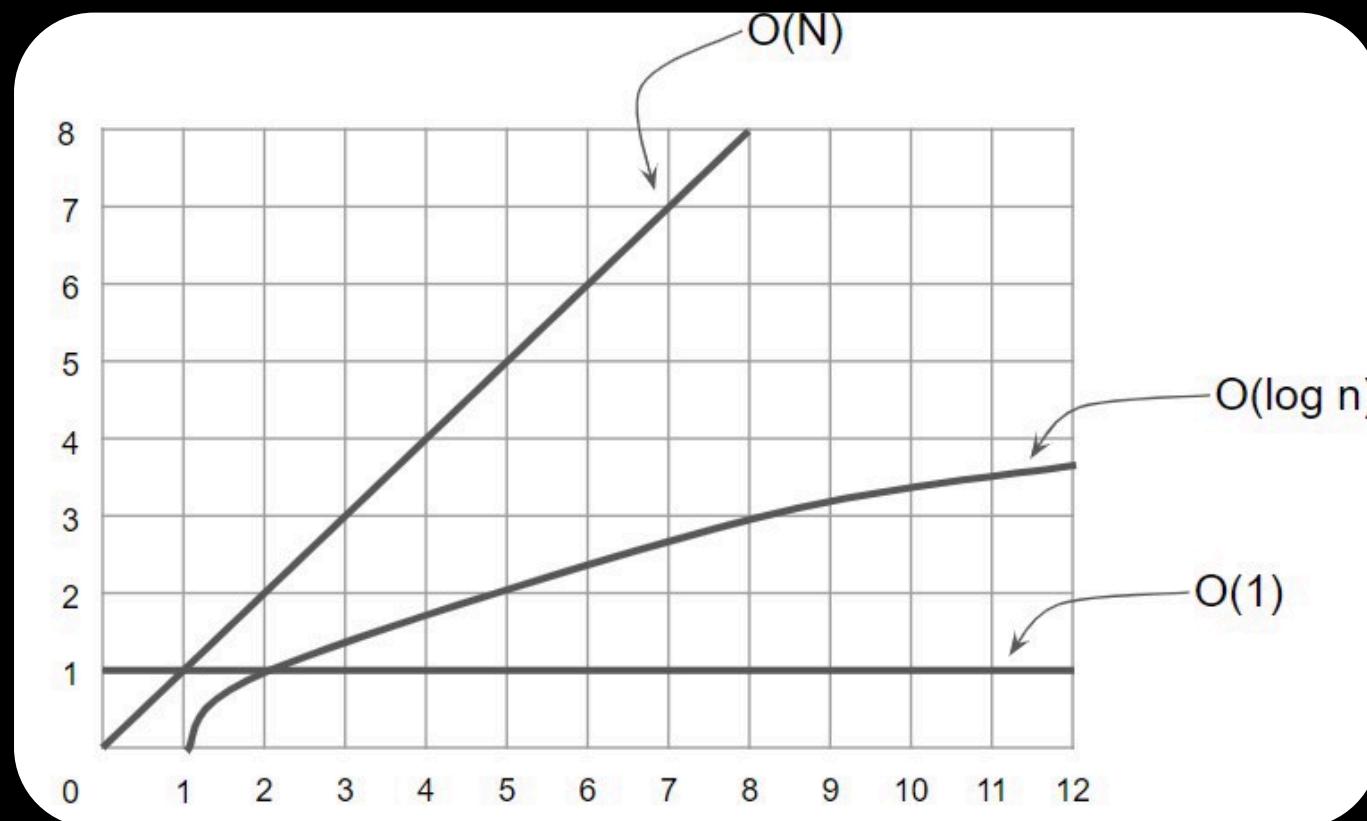
- Andy Raí Berrú Tenorio
- Raúl Edgardo Janampa Salvatierra





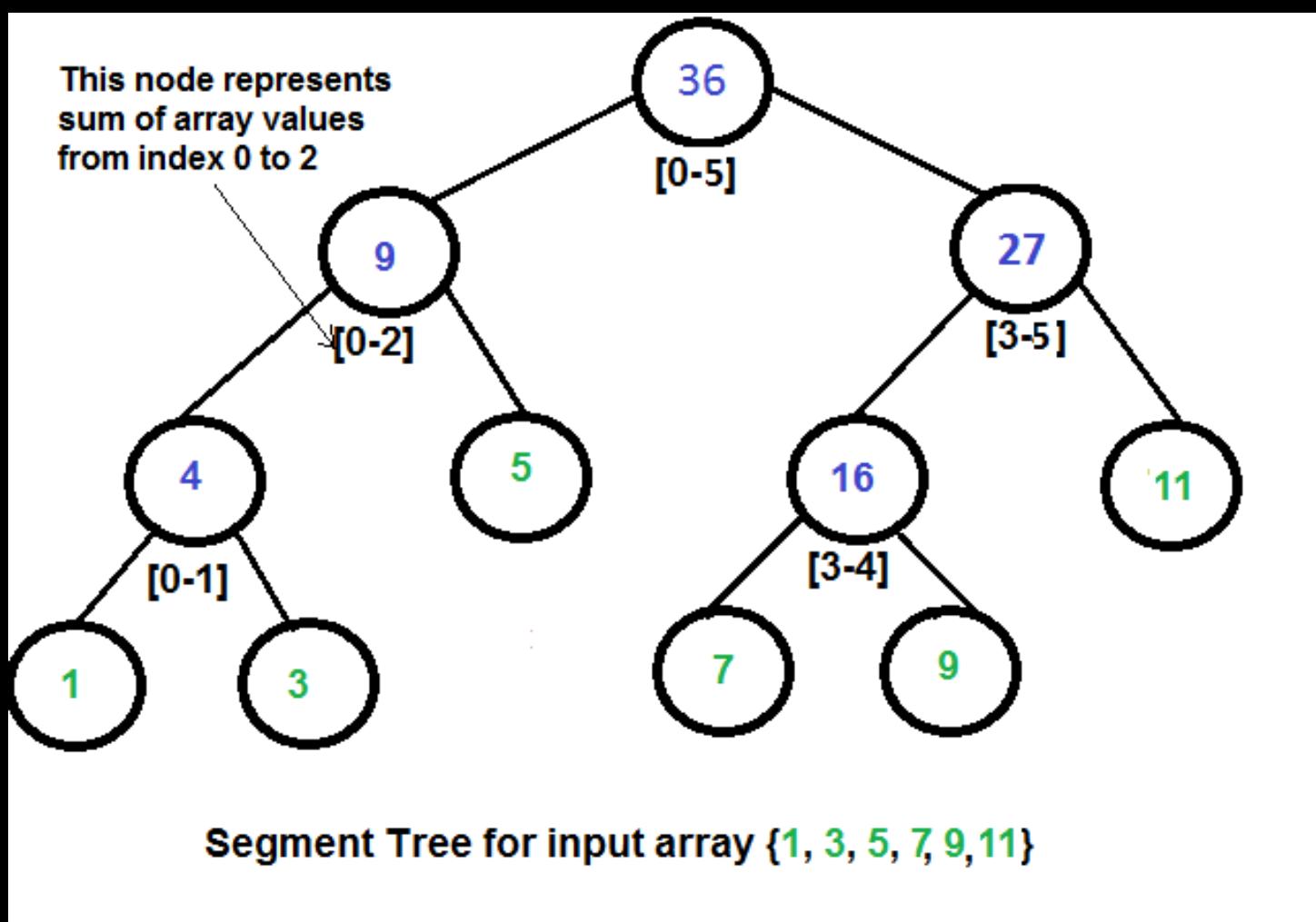
En problemas grandes, las consultas lineales $O(n)$ no son eficientes.

Necesitamos estructuras que permitan consultas rápidas, actualizaciones eficientes y manejo de datos en múltiples dimensiones.



¿Qué es un Segment Tree?

Es una estructura de datos basada en un árbol binario donde cada nodo representa un intervalo del arreglo original.



- ✓ Consultas rápidas sobre intervalos
- ✓ Actualizaciones eficientes
- ✓ Estructura flexible y adaptable

Construcción

“Divide y vencerás”

Arreglo completo de n elementos → [0, n-1]

2 mitades → [0, m] y [m+1, n-1]

Se repite el proceso recursivamente

Cuando el intervalo se reduce a un solo índice → nodo hoja

**CADA NODO REPRESENTA UN INTERVALO Y SU VALOR ES LA “AGREGACIÓN” DE
ESE INTERVALO**

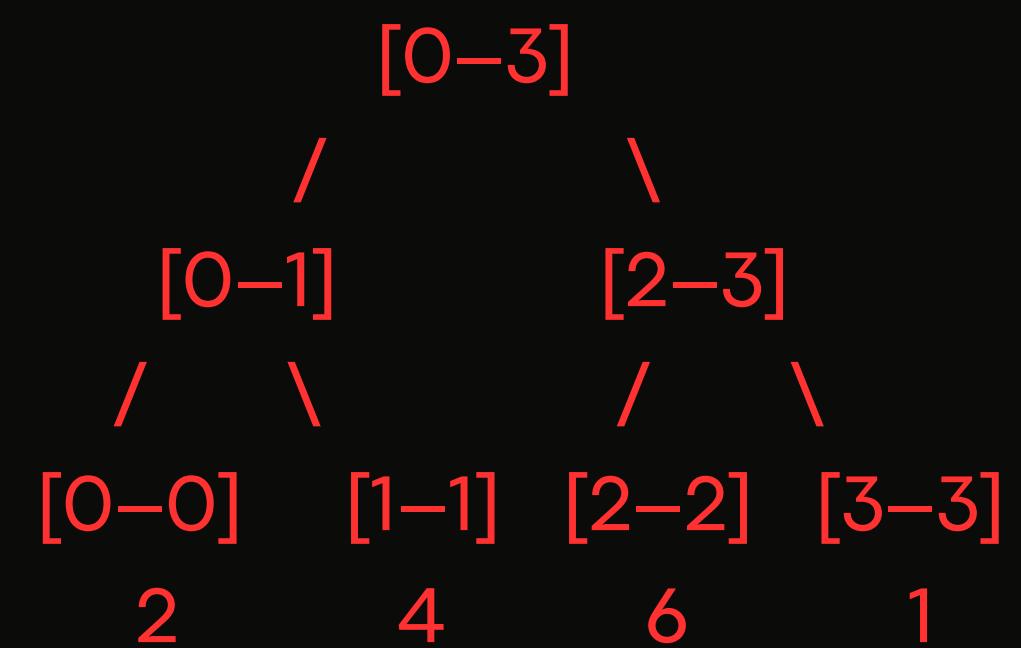
Array: [2, 4, 6, 1], intervalo [0, 3]

Divisiones:

- Izquierda → [0–1]
- Derecha → [2–3]

Luego:

- [0–1] → divide en [0–0] y [1–1]
- [2–3] → divide en [2–2] y [3–3]



Time: O(n)

Space: O(4n)

SUMA

MÍNIMO

MÁXIMO

FREC

$O(\log n)$

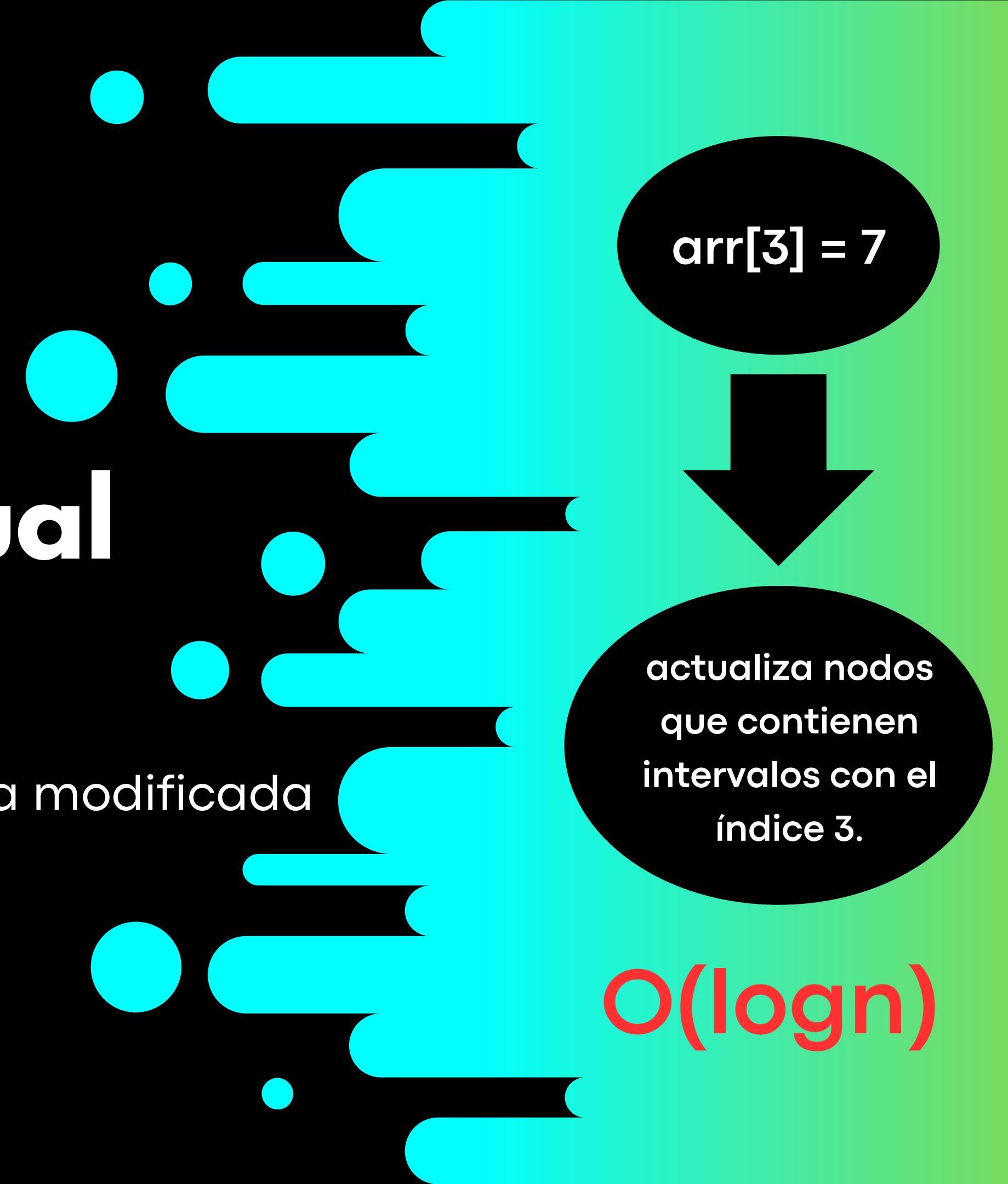
Querys de rango

Una consulta de rango devuelve información
de un intervalo $[l, r]$

Actualización puntual

Modifica un elemento del arreglo

Solo afecta el camino desde raíz a la hoja modificada



Actualización por rango

¡¡ Problema !!

Cuando se quiere modificar un intervalo completo
se tendría que actualizar cada elemento uno por uno ...

Costo: $O(k \log n)$

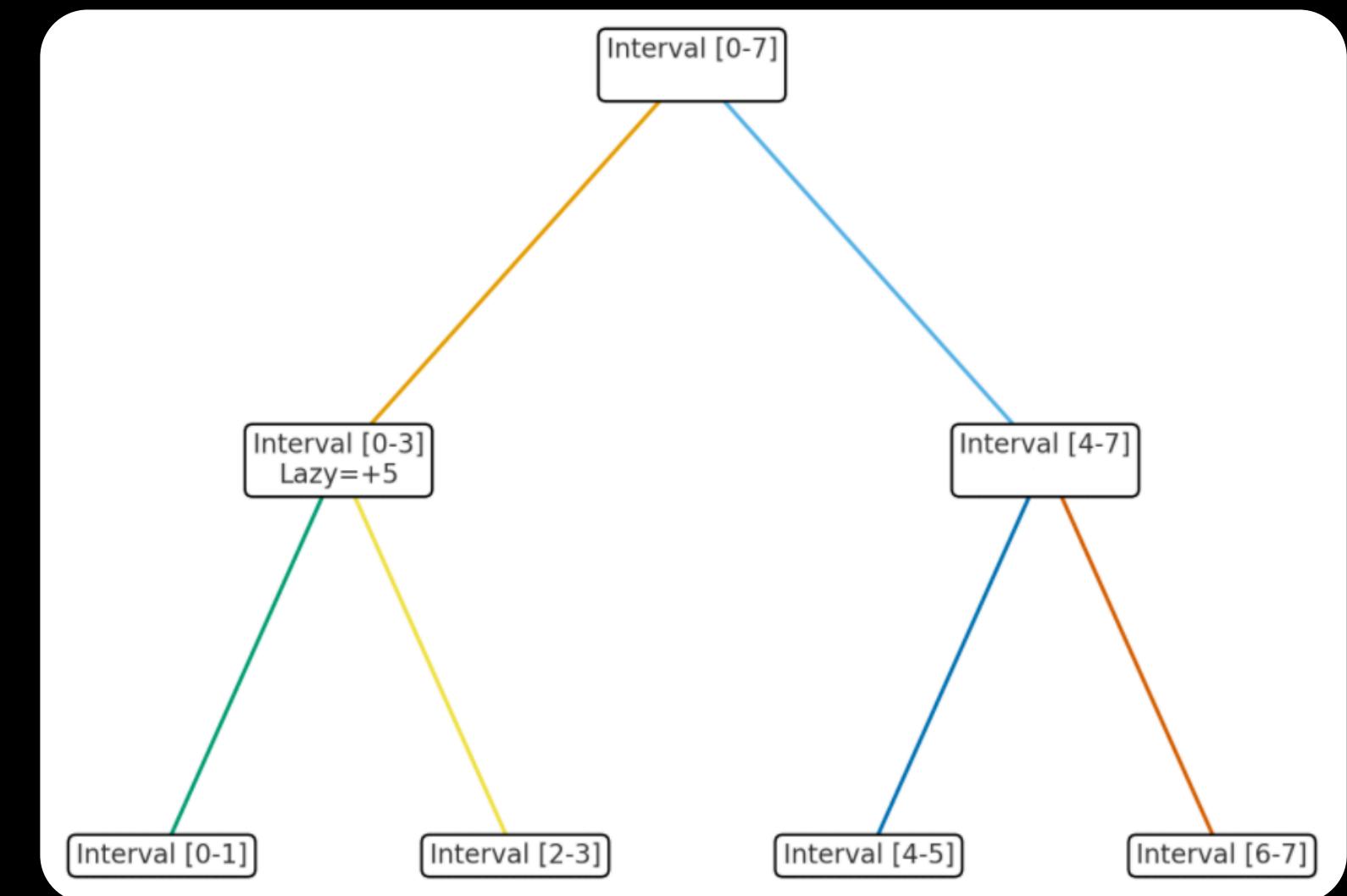
actualización uno por uno

Caro, pero hay una solución ...

Lazy Propagation

Permite realizar actualizaciones por rango sin recorrer todo el intervalo

- se marca el nodo con una etiqueta "lazy",
- se actualiza completamente solo cuando es necesario.



$O(\log n)$

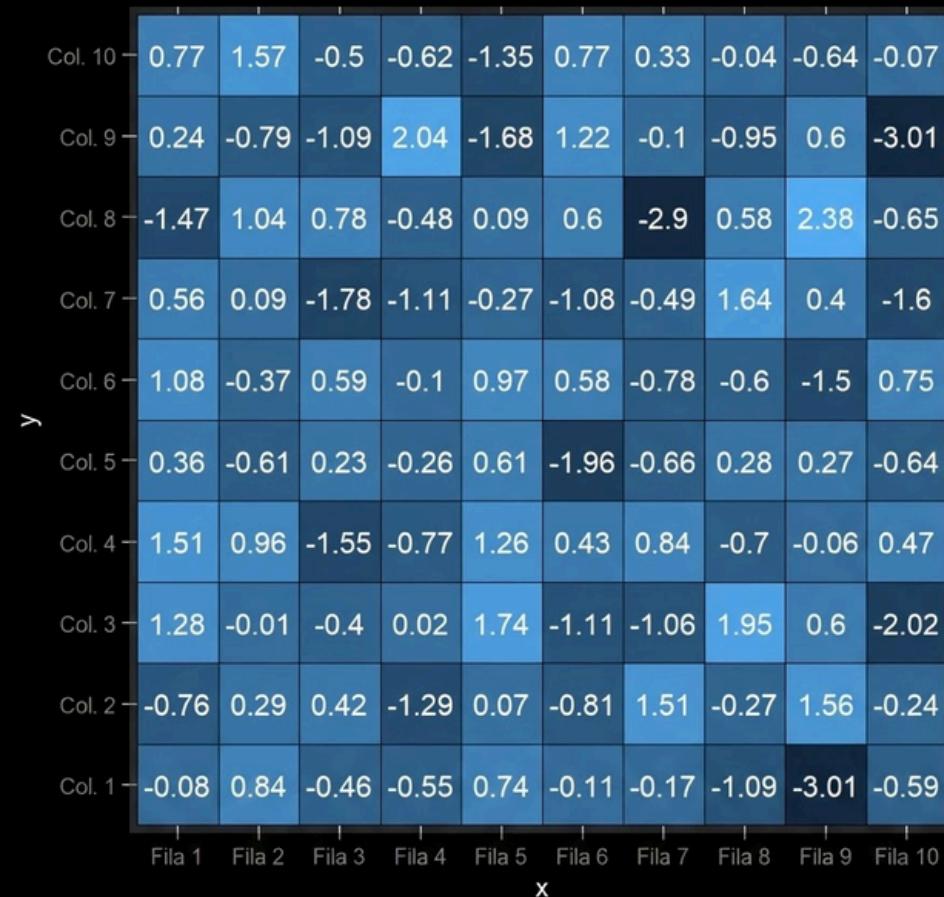
Comparación

Operación	Segment Tree clásico	Lazy Propagation
Query rango	$O(\log n)$	$O(\log n)$
Update puntual	$O(\log n)$	$O(\log n)$
Update por rango	$O(k \log n)$	$O(\log n)$
Complejidad total	Mayor	Menor
Uso típico	Principalmente Lecturas	Muchas actualizaciones

Segment Tree Multidimensional

Se extiende a matrices o incluso cubos

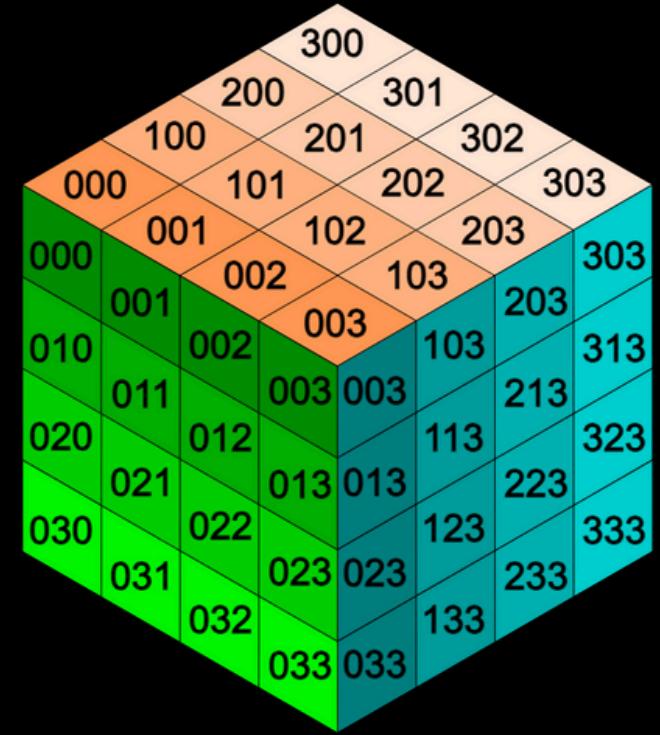
Los nodos cubren submatrices o subvolúmenes



Usos:

2D: mapas de calor, filtros

3D: simulaciones, análisis volumétrico



Complejidad:

Construcción: $O(n^d)$

Query: $O(\log^d n)$

Optimización para Segment Tree (Wang & Wang)

Versión del Segment Tree basada en heaps (heap-like indexing)

Ventaja crucial:
space $O(n)$

NO se usa un heap, se usa la misma representación en arreglo que
usa un heap binario

Solo se crean nodos que realmente existen...

Metodología y Resultados

Segment Tree clásico

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using namespace chrono;
4
5 struct SegmentTree {
6     int n;
7     vector<long long> tree;
8
9     SegmentTree(const vector<int> &arr) {
10         n = arr.size();
11         tree.assign(4 * n, 0);
12         build(arr, node: 1, l: 0, r: n - 1);
13     }
14
15     void build(const vector<int> &arr, int node, int l, int r) {
16         if (l == r) {
17             tree[node] = arr[l];
18             return;
19         }
20         int mid = (l + r) / 2;
21         build(arr, node * 2, l, mid);
22         build(arr, node: node * 2 + 1, l: mid + 1, r);
23         tree[node] = tree[node * 2] + tree[node * 2 + 1];
24     }
25 }
```

↑	Tamaño del arreglo	build (ms)	query (us)	update (us)
↑	100000	2	0	1
↓	500000	12	0	3
☰	10000000	232	0	4
☷	20000000	499	0	5
☷	300000000	17857	224	29

Process finished with exit code 0

Metodología y Resultados

Lazy Propagation

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using namespace chrono;
4
5 struct LazySegTree {
6     int n;
7     vector<long long> tree, lazy;
8
9     LazySegTree(const vector<int>& arr) {
10         n = arr.size();
11         tree.assign(4 * n, val: 0);
12         lazy.assign(4 * n, val: 0);
13         build(arr, node: 1, l: 0, r: n - 1);
14     }
15
16     void build(const vector<int>& arr, int node, int l, int r) {
17         if (l == r) {
18             tree[node] = arr[l];
19             return;
20         }
21         int m = (l + r) / 2;
22         build(arr, node * 2, l, r: m);
23         build(arr, node: node * 2 + 1, l: m + 1, r);
24         tree[node] = tree[node * 2] + tree[node * 2 + 1];
25     }
}
```

Tamaño	Build (ms)	Query (us)	Update (us)
100000	1	0	0
500000	9	0	0
1000000	162	0	1
2000000	304	0	1
30000000	8838	0	8

Process finished with exit code 0

Metodología y Resultados

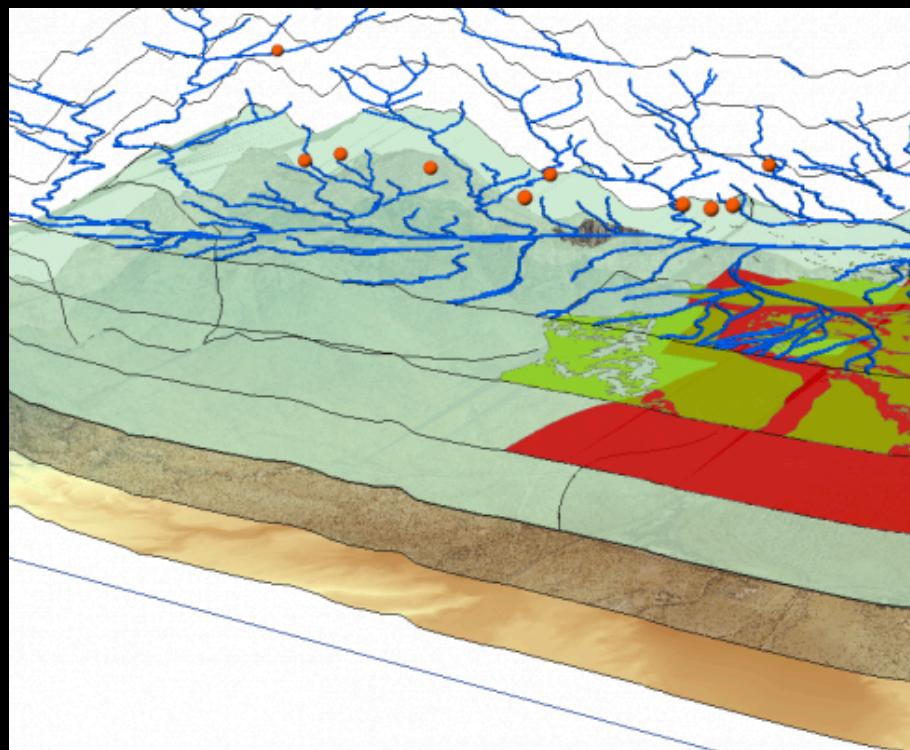
Segment Tree Optimizado

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using namespace chrono;
4
5 struct HeapSegTree {
6     int n;
7     vector<long long> seg;
8
9     HeapSegTree(const vector<int>& arr) {
10         n = arr.size();
11         seg.assign(2 * n, 0);
12
13         for (int i = 0; i < n; i++)
14             seg[n + i] = arr[i];
15
16         for (int i = n - 1; i > 0; --i)
17             seg[i] = seg[i << 1] + seg[i << 1 | 1];
18     }
19
20     long long query(int l, int r) {
21         long long res = 0;
22         for (l += n, r += n; l <= r; l >>= 1, r >>= 1) {
23             if (l & 1) res += seg[l++];
24             if (!(r & 1)) res += seg[r--];
25         }
26     }
27 }
```

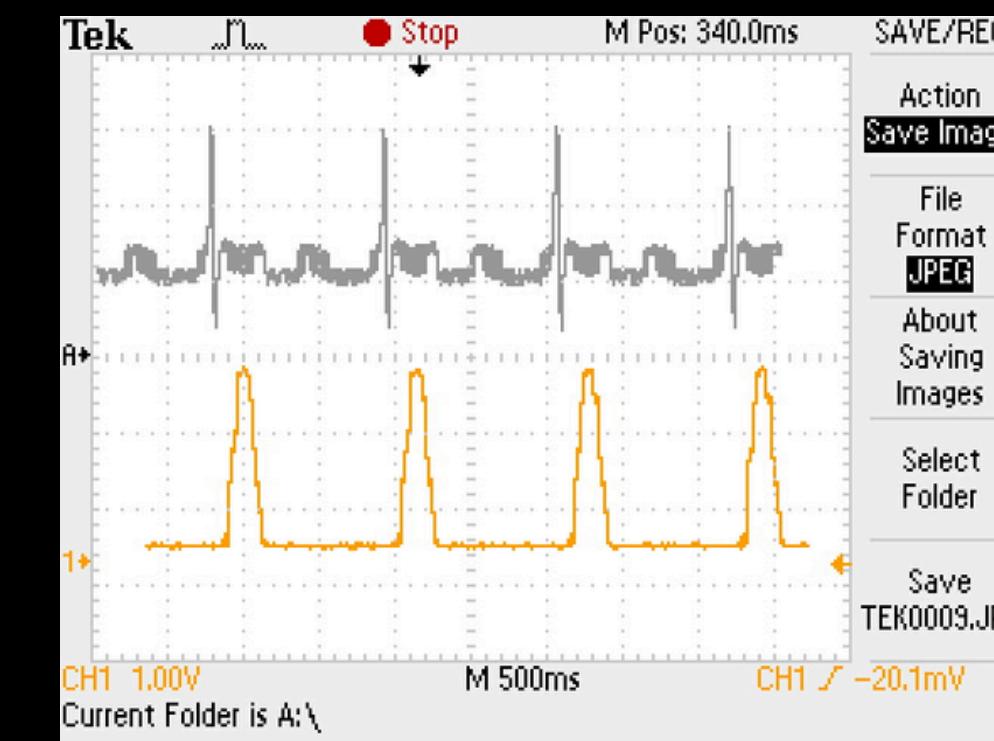
↑	Tamaño del arreglo	build (ms)	query (us)	update (us)	↓
↑	100000	0	0	0	0
↓	500000	4	0	0	0
🖨️	10000000	126	0	0	0
🗑️	20000000	200	0	0	0
↑	300000000	4215	0	1	0

Process finished with exit code 0

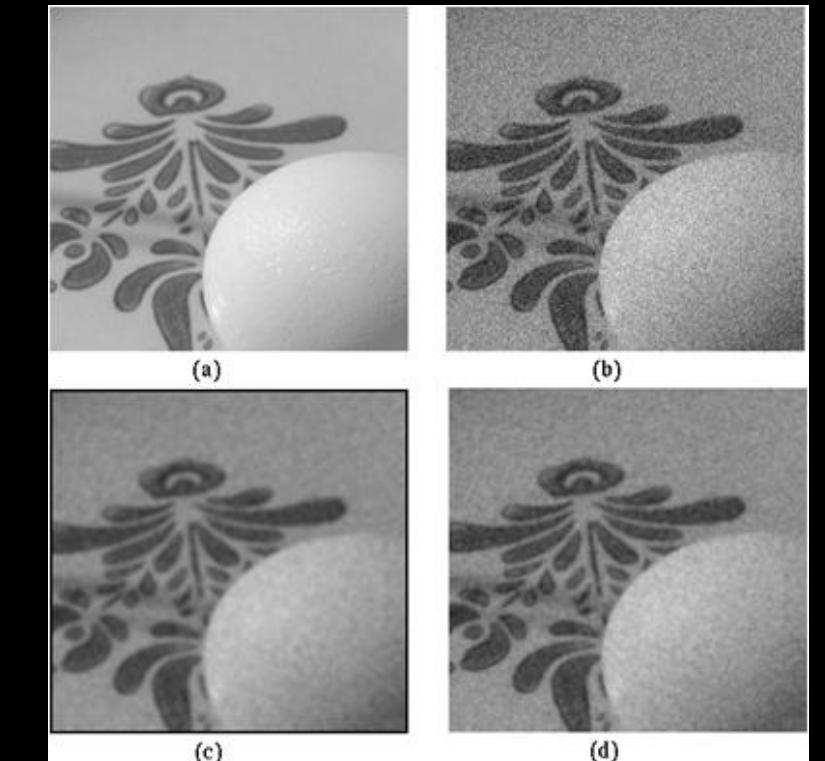
Aplicaciones para los Segment Trees



Sistemas de Información
Geográfica



Filtrado en tiempo real



Procesamiento de
imágenes

Conclusiones

- Un segment tree es invaluable para realizar consultas de rango de manera eficiente.
- La lazy propagation optimiza el rendimiento en situaciones con numerosas actualizaciones.
- Los modelos multidimensionales enriquecen el análisis al aplicarse sobre matrices.

Las versiones optimizadas disminuyen el uso de memoria sin sacrificar velocidad.



GRACIAS

