

# Estructuras I + Tips (:



Redigonda Maximiliano

# Índice

1. Tips
2. Estructuras Básicas
3. Segment Tree
4. Problema

---

# TIPS

# Matemáticas

Pensamiento matemático es **fundamental** para un equipo de programación competitiva.

# Matemáticas

Es mejor usar 10 minutos en la demostración, que 50 minutos en una implementación que no sabemos si va a funcionar.

Cuando intentás demostrar algo y no funciona, a veces te el motivo por el que no funciona.

# **Solución = Observaciones**

La idea de un problema generalmente es un conjunto de observaciones.

**WA != 0**

Fallar un problema no significa que está 0% resuelto.

# AC $\neq$ 1

Acertar un problema no significa que está 100% resuelto.

Aprender otras soluciones para evitar las “**compensaciones**”.

Ejemplos: sumar de 1 a n, greedy no demostrado, SCC en vez de DFS, etc.



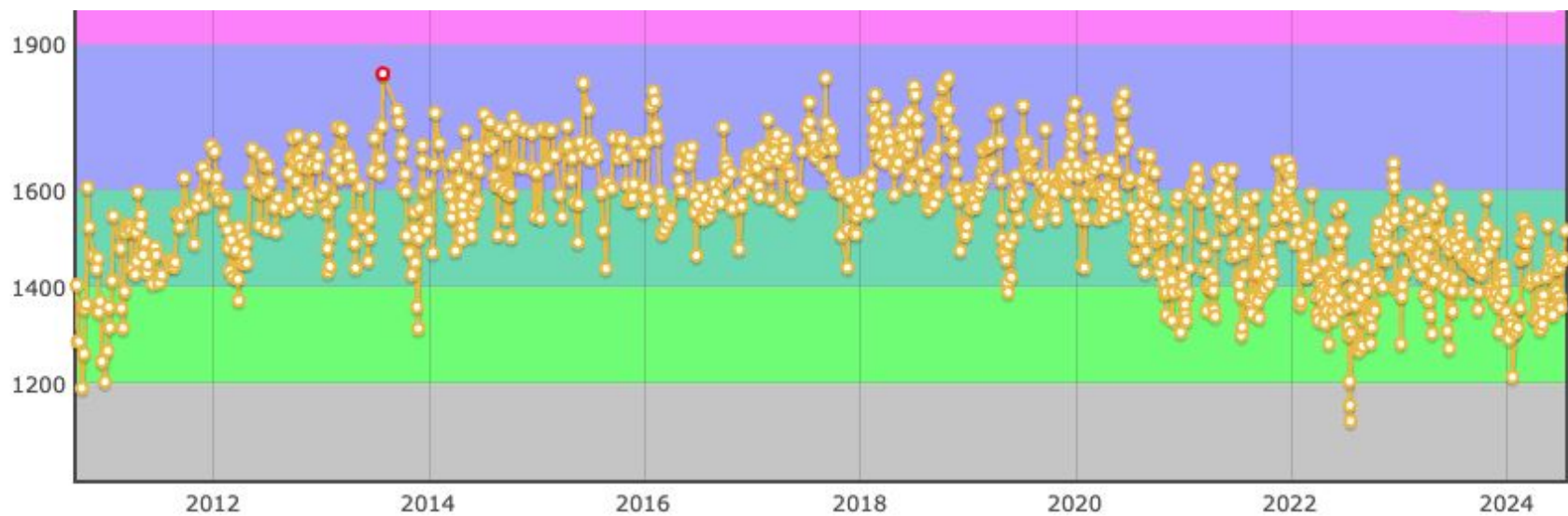
# Upsolving

En los simulacros, por definición uno resuelve lo que puede resolver.

Para mejorar, tenés que aprender a resolver lo que no podés resolver (aun).

# Entrenar != Simular

Los simulacros o las competencias, son básicamente una foto de las habilidades actuales.



# Estructuras Básicas

# Arrays

Tamaño fijo, y acceso en  $O(1)$  (rapidísimo).

Se guarda todo junto en la memoria así que puede aprovecharse la memoria cache (#dato).

```
int A[10] = {}; // inicializado a 0  
int B[200]; // inicializado con la memoria que ya estaba
```

Se pueden usar como “lookup tables”.

# Vector

Array con tamaño dinámico.

```
vector<int> A; // vector vacío
```

```
vector<int> A(9); // vector de 9 ints con valor 0
```

```
vector<int> A(9, 3); // vector de 9 ints, todos con valor 3
```

```
vector<vector<int>> A(10, vector<int>(10, -1));
```

# Vector

Vector ya tiene operaciones divertidas:

- `.size()`
- `.empty()`
- `.clear()`
- `.push_back()`
- `.pop_back()`

Las últimas dos operaciones no van a servir para simular una pila (stack).

También se pueden acceder igualito a los arrays, con `[i]`.

# Stack

Existe, pero usamos Vector que es más flexible!



# Queue

Simula una fila (en un banco, supermercado, etc.), donde el primero en ingresar, es el primero en salir.

Sirve para agregar “eventos” que queremos que se vayan procesando en el orden en que fueron agregados.

```
queue<int> Q;  
Q.push(5); Q.push(3); Q.push(-1);  
Q.front(); // 5  
Q.pop(); // chau 5  
Q.front(); // 3
```

# Priority Queue

Nada que ver con una queue, insertás elementos en cualquier orden, y se van removiendo los de mayor “prioridad” (por defecto, los “mayores”).

Usando en el algoritmo **Dijkstra**.

```
priority_queue<int> Q;  
Q.push(5);  
Q.push(10);  
Q.push(7);  
Q.top(); // 10  
Q.pop();  
Q.top(); // 7
```

# Priority Queue

**Atención:** agregar y remover elementos en las priority queue es una operación que toma  $O(\log n)$ .

Las priority queue son implementadas con una estructura llamada “heap” internamente.

**Tarea:** programá por tu cuenta un heap!

# Set

Es una colección **ordenada** de elementos **únicos**.

```
set<int> S;  
S.insert(5);  
S.insert(10);  
S.insert(5); // no da error, pero el elemento no se agrega  
S.size(); // 2  
S.erase(5);  
S.size(); // 1
```

# Set

Los sets se manejan con **punteros**, que son como flechitas a ciertos elementos del conjunto (no podemos usar `[i]`).

Muchas operaciones en los set son eficientes!

```
.insert(x)
```

```
.erase(x)
```

```
.count(x) // si, existe aunque pueda ser solo 0 o 1
```


```
.lower_bound(x) // primer elemento  $\geq x$ 
```

```
.upper_bound(x) // primer elemento  $> x$ 
```

Todas las anteriores son  $O(\log n)$

# MultiSet

Cuando necesitamos meter más de un solo elemento en un set, podemos usar **multiset**.

```
multiset<int> S;  
S.insert(3);  
S.insert(3);  
S.size(); // 2  
S.erase(3); //  Atención: esto borra los dos elementos!  
S.size(); // 0  
S.insert(10);  
S.insert(10);  
S.erase(S.find(10));  
S.size(); // 1
```

# Map

Como el set, pero a cada elemento (key) que se agrega se le “mapea” un valor.

```
map<int, string> M; // mapea de tipo `int` a tipo `string`
M[5] = "V"; // asociamos el 5 con la string "V"
M.size(); // 1
M[7] = "VII"; // asociamos el 7 con la string "VII"
M.size(); // 2
if (M[10] == "X") { ... }
// 🖐 este patrón es engañoso, sin querer agregamos el elemento 10
M.size(); // 3
if (M.count(3) > 0 && M[3] == "III") { ... }
// 🖐 este patrón si funciona como esperamos
```

# Unordered Set & Unordered Map

Uso casi idéntico al set y al map, solo que sus elementos no están ordenados.

Muchas operaciones pasan a ser  $O(1)$  en vez de  $O(\log n)$ .

Generalmente solo set/map funcionan suficientemente bien!



# Segment Tree

# Problema

Se tiene un array  $A$ , con  $N$  elementos, y se quiere soportar las siguientes operaciones de forma eficiente:

- $\text{set}(i, x): A[i] = x$
- $\text{query}(i, j): A[i] + A[i + 1] + \dots + A[j - 1]$

Existen dos soluciones que se enfocan en cada una de estas queries por separados, pero segment tree logra resolver ambas de forma eficiente al mismo tiempo.

# Problema - Solución con Array

- `set(i, x): A[i] = x`

Optimizar esta es sencillo, un simple array soporta esta operación en  $O(1)$ , pero luego la query de la suma de  $i$  hasta  $j$ , va a demorar  $O(n)$ .

# Problema - Solución con Prefix Sums

- $\text{query}(i, j): A[i] + A[i + 1] + \dots + A[j - 1]$

Optimizar esta es un poco más interesante, podemos usar una “tablita aditiva” o “prefix sums”.

# Problema - Solución con Prefix Sums

Se crea una tabla que en cada posición  $i$  tiene la suma de los elementos desde el índice 0 hasta el índice  $i$ .

Luego, la respuesta a la query( $i, j$ ) puede verse como  $ta[j] - ta[i]$  (+/- 1 dependiendo de la implementación).

Sin embargo, para actualizar una tabla como esta es requerido  $O(n)$  operaciones.

# Segment Tree

El segment tree es una estructura para responder consultas sobre rangos (no solo sumas!).

Sobre un array, se puede extender con elementos neutros hasta que tenga tamaño potencia de dos, y luego se crea un arbol binario completo cuyas hojas son los elementos del array.

Luego, cada hoja representa el rango  $[i, i + 1)$ , y los nodos internos representan la unión de los rangos de los hijos.

# Segment Tree

El nodo raíz lo guardamos en el índice 1.

El hijo izquierdo se guarda en el hijo  $2*i$ , y el hijo derecho en el índice  $2*i+1$ .

El nodo padre naturalmente va a estar en el índice  $i/2$  (piso).

# Segment Tree

Sabiendo cómo navegar el árbol, la operación de actualización puede soportarse de la siguiente forma:

1. Actualizo el valor del nodo actual
2. Si hay padre, convertirlo en el nodo actual y volver al paso 1

Si no hay, padre, es porque terminamos de actualizar todos.



# Segment Tree

Luego, la operación de query puede responderse como una combinación de varios segmentos.

Comenzamos desde la raíz, repitiendo el proceso:

1. Si el rango del nodo está completamente incluido en la query, se retorna su valor.
2. Si el rango del nodo interseca con el de la query, se suma el subproblema en ambos sub-árboles hijos.
3. Si el rango no intersecta con el de la query, se retorna 0.

# Segment Tree

Cuando procesamos una query de esta forma, en cada nivel visitaremos **no más que cuatro nodos**.

## **Demostración por inducción:**

**Caso base:** nodo raíz, solo visitamos un nodo.

## **Caso inductivo:**

- Si visitamos  $\leq 2$  nodos en el paso anterior, sabemos que en el paso siguiente vamos a visitar  $\leq 4$  nodos, porque cada nodo tiene máximo dos llamadas recursivas
- Si visitamos 3 o 4 nodos en el paso anterior, si analizamos los vértices del medio con más atención, vamos a ver que corresponden a un rango completamente cubierto por la query (caso 1 en la diapositiva anterior) y por lo tanto no van a tener llamadas recursivas.

Luego, el algoritmo es  $O(\log n)$  (la constante  $\times 4$  no se cuenta!).

# Segment Tree - Código para construí-lo

```
int n, t[4*MAXN];
```

```
void build(int a[], int v, int tl, int tr) {  
    if (tl == tr) {  
        t[v] = a[tl];  
    } else {  
        int tm = (tl + tr) / 2;  
        build(a, v*2, tl, tm);  
        build(a, v*2+1, tm+1, tr);  
        t[v] = t[v*2] + t[v*2+1];  
    }  
}
```

# Segment Tree - Código para la query

```
int sum(int v, int tl, int tr, int l, int r) {  
    if (l > r)  
        return 0;  
    if (l == tl && r == tr) {  
        return t[v];  
    }  
    int tm = (tl + tr) / 2;  
    return sum(v*2, tl, tm, l, min(r, tm))  
        + sum(v*2+1, tm+1, tr, max(l, tm+1), r);  
}
```

# Segment Tree - Código para actualizar

```
void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

# Extra: Fenwick Tree

# Fenwick Tree

Existe, resuelve muchas de los problemas que segment tree resuelve, es más cortito de programar, pero a los propósitos de esta clase nos quedamos con segment tree por ser más flexible!

# Problema



# Problema

Obtener el primer índice de un array de números positivos en que la suma supera un cierto valor  $X$  (o  $-1$  si no existe).

Solución trivial:  $O(n)$

Solución binary search + segment tree:  $O(\log(n) * \log(n))$

Solución segment tree:  $O(\log n)$

# Solución Trivial

Recorrer el arreglo, sumando los valores, hasta que la suma supere el valor X, o si llego al final y no sucede, retornar -1.

# Solución Binary Search + Segment Tree

Sabemos que el segment tree, dado cualquier intervalo, nos da la suma en  $O(\log n)$ .

Sabemos que con Binary Search podemos fijar un índice y preguntar “la suma hasta acá, supera el valor  $X$ ”? Mientras más a la derecha esté el índice, mayor será la suma, y viceversa.

La pregunta “la suma hasta acá, supera el valor  $X$ ” se puede responder con segment tree!

Técnicamente la pregunta también se puede responder con “tablita aditiva”, pero dejaría de funcionar si queremos también soportar operaciones de actualización.

# Solución Segment Tree

Comenzamos en la raíz del árbol, que representa el intervalo completo. Si la suma no supera a  $X$ , entonces retornamos -1.

Luego, nos fijamos en sus hijos, digamos que la suma en el hijo de la izquierda es  $A$ , y la suma del hijo de la derecha es  $B$ .

Si la suma  $A$  es mayor que  $X$ , significa que el segmento que queremos está contenido en el hijo de la izquierda.

Si no, es equivalente a resolver el mismo problema en el sub-árbol de la derecha, pero con valor objetivo  $X - A$  (**recursión**).

Como se realiza solo una bajada en el árbol, que tiene altura  $O(\log n)$ , y solo hay una operación constante en cada nivel, el algoritmo es  $O(\log n)$  🎉.

# Gracias!