# Fenwick Trees

Román Castellarin

# El problema: suma *dinámica* en rangos

Se tiene un arreglo `A[1..N]` con números enteros. Se quieren implementar dos operaciones:

- `get(a, b)`: devuelve la suma de los elementos en `A[a..b]`
- `update(x, v)`: realiza A[x] += v

Supongamos que tenemos que realizar Q *gets*, y U *updates*. Considerar que pueden intercalarse *gets* y *updates* (dinámica).

# Solución propuesta 1

```
int A[ MAXN ];

int get (int a, int b){
    int v = 0;
    for (int i = a; i <= b; ++i)
        v += A[x];
    return v;
}

void update (int x, int v) {
   A[x] += v;
}
```

# Solución propuesta 1

```
int A[ MAXN ];

int get (int a, int b){
    int v = 0;
    for (int i = a; i <= b; ++i)
        v += A[x];
    return v;
}

void update (int x, int v) {
   A[x] += v;
}
```

O(N)

O(1)

# Solución propuesta 2

Supongamos ahora que se hacen muchos *gets* y pocos *updates*. Querríamos que la operación más rápida sea get, ya que se realiza muchas veces, y la lenta sea update.

Por suerte, podemos lograr esto mediante la introducción de una **tabla aditiva**, `T`; de forma tal que

```
T[k] = A[1]+A[2]+...+A[k]
```

# Solución propuesta 2 (tabla aditiva)

```
int T[ MAXN ];

int get (int a, int b){
    return T[b] - T[a-1];
}

void update (int x, int v) {
   for (int i = x; i < MAXN; ++i)
       T[i] += v;
}
```

# Solución propuesta 2 (tabla aditiva)

```
int T[ MAXN ];

int get (int a, int b){
    return T[b] - T[a-1];
}

void update (int x, int v) {
   for (int i = x; i < MAXN; ++i)
       T[i] += v;
}
```

$O(1)$

$O(N)$

# Análisis

Q consultas
U actualizaciones

Solución 1:
$O(NQ + U)$

Solución 2:
$O(Q + NU)$

# Análisis

- Si Q, U son similares a N, ambas soluciones son cuadráticas $O(N^2)$
- Necesitamos algo más eficiente
- Un Segment Tree realiza ambas operaciones en O(log N)
- Complejidad total: O((Q+U) log N) ✓

# Solución propuesta 3 (Segment Tree)

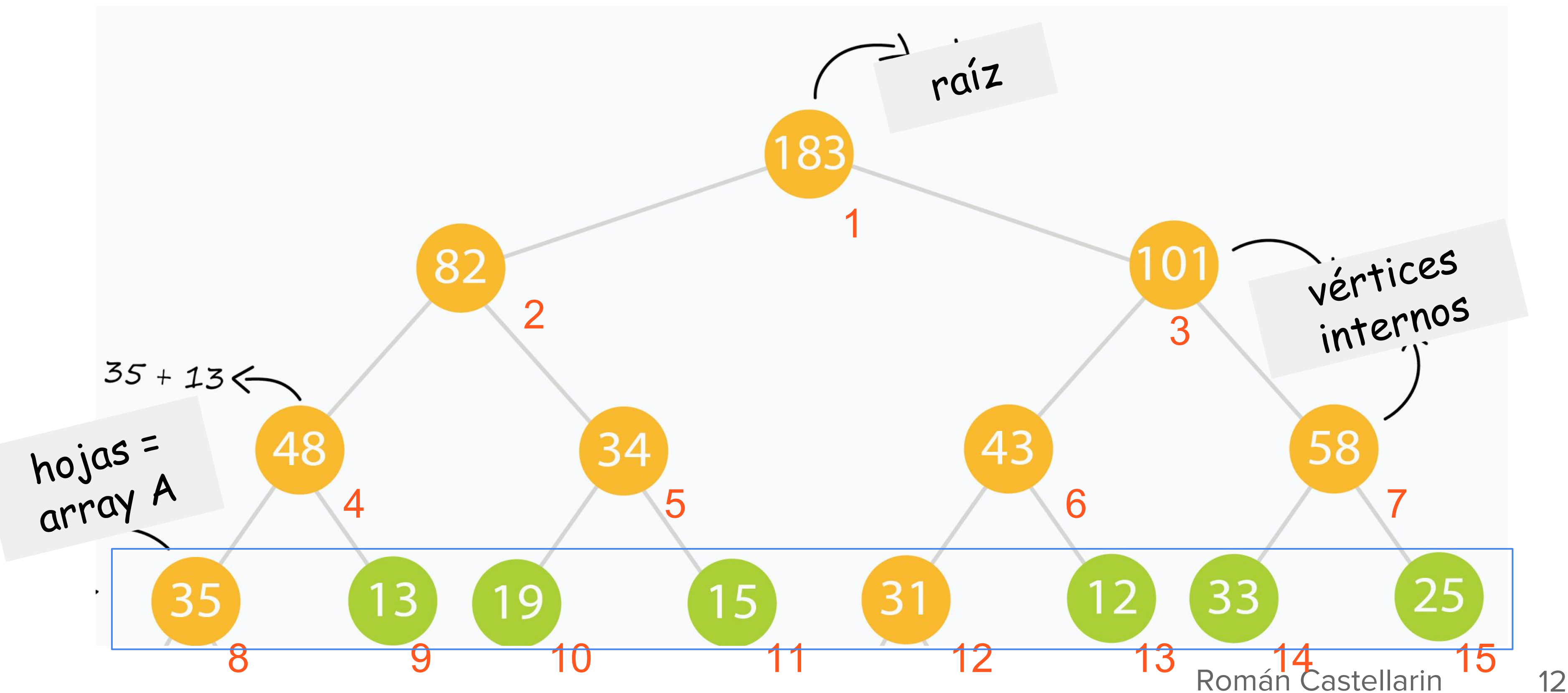
```
int ST[ MAXN * 2 ];

int get (int n, int s, int e, int a, int b){
    if( b < s || e < a ) return 0;
    if( a <= s && e <= b ) return ST[n];
    int m = (s + e) / 2;
    int v1 = get(2*n, s, m, a, b);
    int v2 = get(2*n+1, m+1, e, a, b);
    return v1 + v2;
}
```

# Solución propuesta 3 (Segment Tree)

Un Segment Tree es una estructura de árbol binario, cuyas hojas son los elementos del array A, y donde cada vértice interno guarda la suma de sus dos hijos.

Además, nombramos los vértices por niveles, comenzando con la raíz la cual etiquetaremos con el 1.

# Solución propuesta 3 (Segment Tree)



# Solución propuesta 3 (Segment Tree)

```
int ST[ MAXN * 2 ];

int get (int n, int s, int e, int a, int b){
    if( b < s || e < a ) return 0;
    if( a <= s && e <= b ) return ST[n];
    int m = (s + e) / 2;
    int v1 = get(2*n, s, m, a, b);
    int v2 = get(2*n+1, m+1, e, a, b);
    return v1 + v2;
}
```

doble memoria

código *largo*

*(relativo a Fenwick)*

propenso a errores (?)

# Fenwick Trees

## (Binary-indexed Trees)

# Fenwick trees

Son una estructura de datos que soporta las siguientes dos operaciones sobre un array *implícito* `A[1..N]`:

- `getFT(b)`: devuelve la suma de los elementos en `A[1..b]`
- `setFT(x, v)`: realiza `A[x]+=v`

¡Ambas operaciones son O(log N)!

# ¿Cómo funcionan?

# Nadie sabe.

# ...bueno, quizás sí. Pero es complicado.

- El bit prendido menos significativo de un entero x, puede obtenerse haciendo LSB(x) = (x & -x).
  *Demostración: ejercicio*

- Un **Fenwick Tree es un árbol infinito**, con raíz en el vértice cero, donde el vértice x tiene la suma de los elementos en A[ x-LSB(x)+1 .. x ].

# ...bueno, quizás sí. Pero es complicado.

Veamoslo con un ejemplo:

```
...000000001100 =   12, 
...111111110100 =  -12, 
...000000000100 =    4 ← 
```

`...000000001100 = 12`, tiene 2 bits prendidos (4 y 8)
`...111111110100 = -12`, ya que al sumarle 12 da 0
`...000000000100 = 4` ← bit menos significativo

`12 - 4 + 1 = 9.`

Por lo tanto, el vértice `12` guarda la suma de `A[9..12].`

`FT[12] = A[9]+A[10]+A[11]+A[12]`

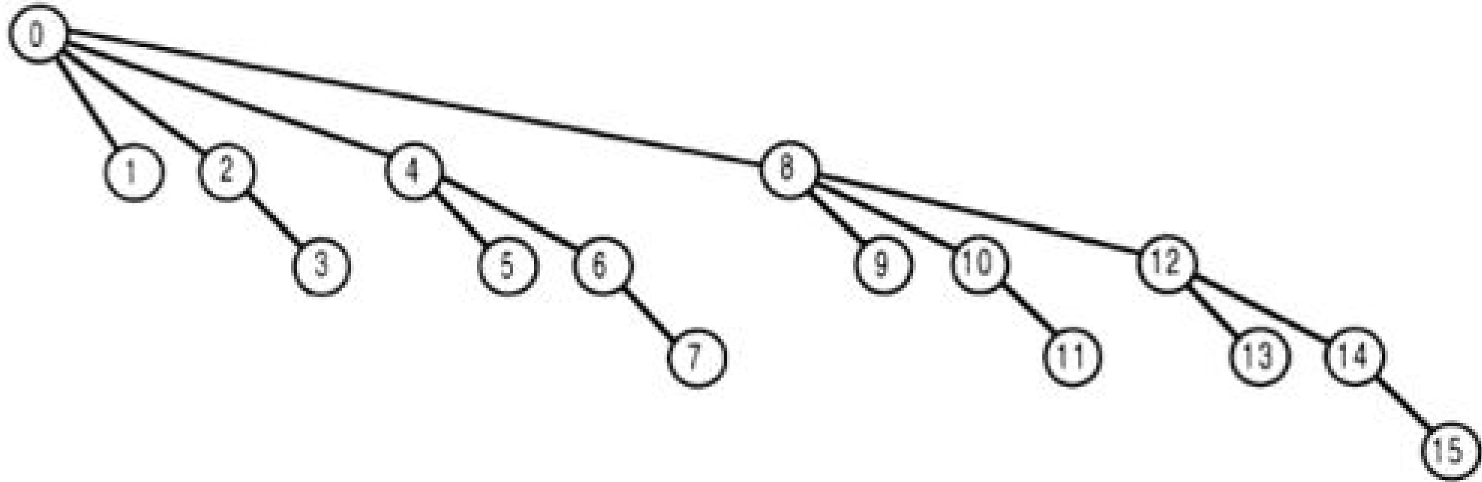
# …bueno, quizás sí. Pero es complicado.

- En un Segment Tree, el padre de un vértice x se podía encontrar haciendo $\lfloor x/2 \rfloor$ *(ver gráfico anterior)*

- En un Fenwick Tree, el padre de un vértice x se puede encontrar haciendo x - LSB(x)

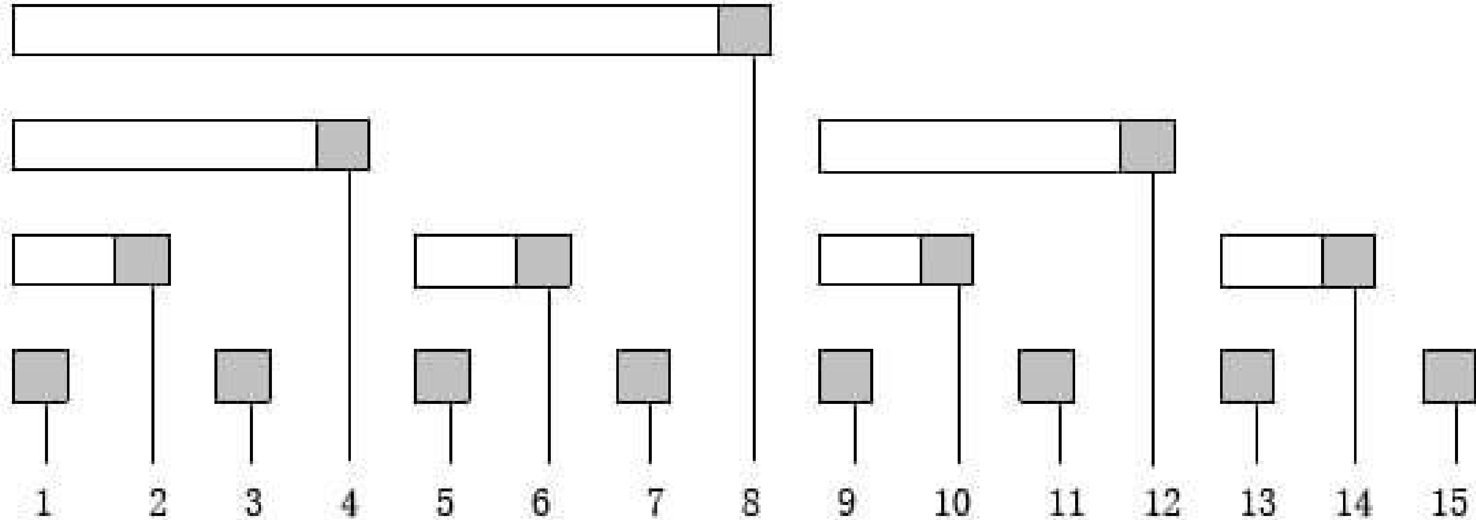
  Ej: El padre de 12 es 8, porque 12 - 4 = 8

## Estructura (literal, padres e hijos)




**Fenwick Tree** (sólo los vértices 0..15 están dibujados)

# Los intervalos a cargo de cada vértice (el vértice 0 no se usa)



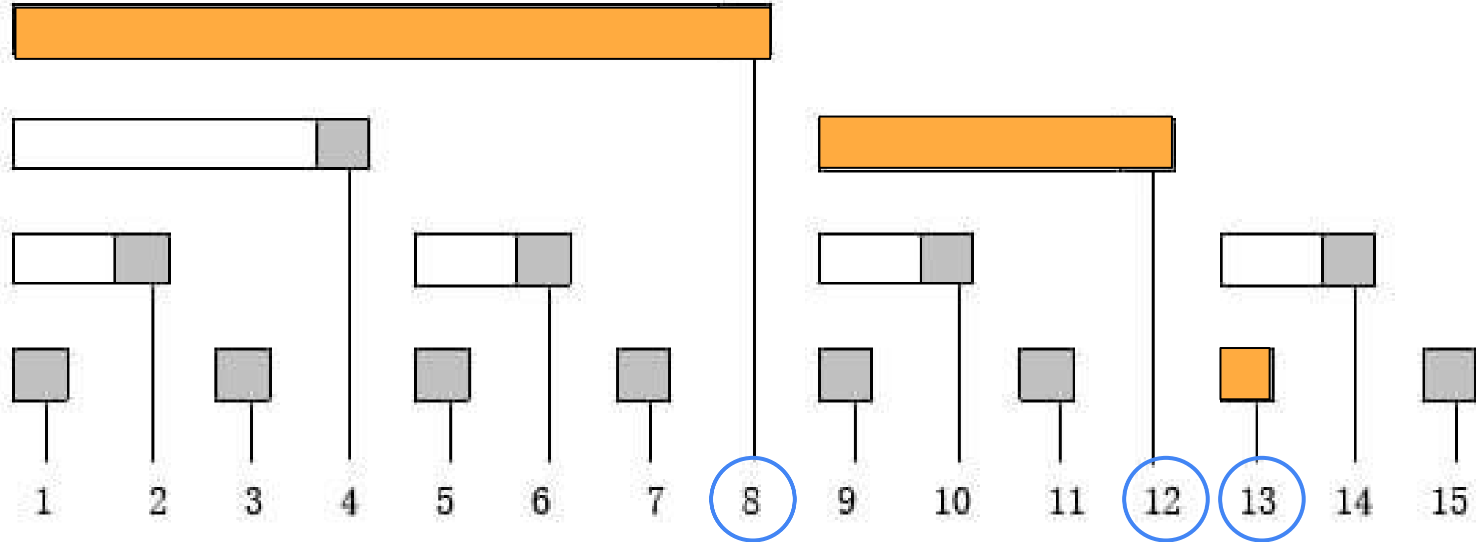

**Fenwick Tree** (sólo los vértices 0..15 están dibujados)

# ¿Es complicado?

- Por suerte, no nos interesa mucho saber cómo funciona, sino saberlo usar.

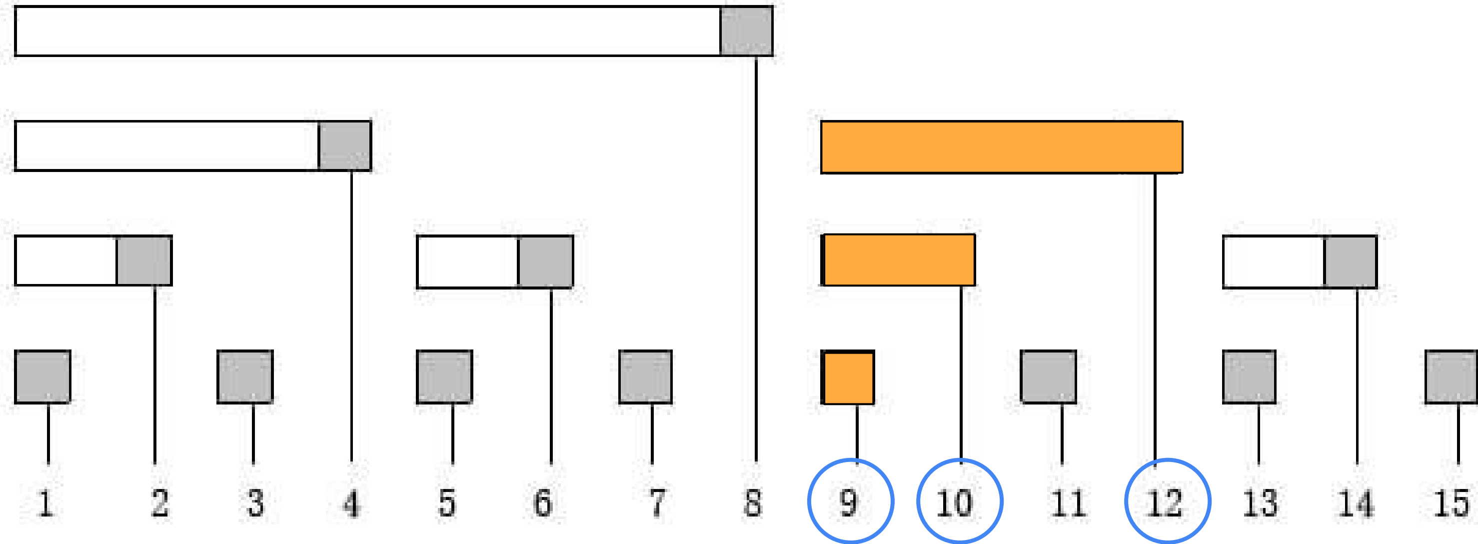
- Ambas operaciones son extremadamente **sencillas** de programar.

Si queremos sumar A[1..13], debemos hacer FT[13] + FT[12] + FT[8]. Notemos que el padre de 13 es 12, y el de 12 es 8.




**Fenwick Tree (sólo los vértices 0..15 están dibujados)**

Si queremos modificar A[9], debemos actualizar FT[9], FT[10] y FT[12]. Notemos que 9 + LSB(9) = 10, y 10 + LSB(10) = 12




**Fenwick Tree (sólo los vértices 0..15 están dibujados)**

# Fenwick Trees: operaciones

```
int FT[ MAXN ];

int getFT (int b){
    int v = 0;
    for (int x = b; x; x -= x & -x)
        v += FT[x];
    return v;
}

void setFT (int p, int v) {
  for (int x = p; x < MAXN; x += x & -x)
        FT[x] += v;
}
```

# Fenwick Trees: operaciones

```
int FT[ MAXN ];

int getFT (int b){
    int v = 0;
    for (int x = b; x; x -= x & -x)
        v += FT[x];
    return v;
}

void setFT (int p, int v) {
  for (int x = p; x < MAXN; x += x & -x)
        FT[x] += v;
}
```

sólo N memoria

O(log N)

*(operaciones de bits == rapidez)*

O(log N)

# El problema: suma dinámica en rangos

Ahora, volviendo al problema original, podemos calcular:

- `get(a, b) ≡ getFT(b) - getFT(a-1)`
- `update(x, v) ≡ setFT(x, v)`

¡Escribiendo ~3 líneas por cada función del Fenwick Tree!
¡Cada función es O(log N), y es muy rápida ya que usa operaciones de bits!

# Ejemplo

```
setFT(4, 100);   → A[4]  = 100
setFT(10, 50);   → A[10] = 50
getFT(3);        → 0
getFT(4);        → 100
getFT(9);        → 100
getFT(70);       → 150
setFT(10, -20);  → A[10] = 30
getFT(70);       → 130
```

# Mejoras (augmentations)

# Mejora copada 1

# (Re)interpretaciones

# Interpretaciones

Notemos que en ningún momento almacenamos el array A, por eso decimos que está implícito.

Sin embargo, hay una razón más importante por la cual el array A está *implícito*:

Las operaciones del Fenwick Tree se pueden **interpretar** de dos maneras diferentes.

# Fenwick trees

Interpretación 1 (la vista hasta ahora):

- `getFT(b)`: devuelve la suma de los elementos en `A[1..b]`
- `setFT(x, v)`: realiza `A[x]+=v`

Interpretación 2:

- `getFT(b)`: devuelve `A[b]`
- `setFT(x, v)`: realiza `A[i]+=v para i en [x..∞]`

# Fenwick trees

Interpretación 1: point update, range query

- `getFT(b)`: devuelve la suma de los elementos en `A[1..b]`
- `setFT(x, v)`: realiza `A[x]+=v`

Interpretación 2: range update, point query

- `getFT(b)`: devuelve `A[b]`
- `setFT(x, v)`: realiza `A[i]+=v para i en [x..∞]`

# Ejemplo (interpr. 2)

```
setFT(4, 100);
setFT(10, -100);   } → A[4..9] = 100

getFT(3);          → 0
getFT(4);          → 100
getFT(6);          → 100
getFT(9);          → 100
getFT(10);         → 0
```

# Mejora copada 2

# Lazy Creation

# Fenwick trees

Supongamos que N es grande... $N \leq 10^{18}$.

No podemos hacer un array tan grande. Sin embargo, notemos que la mayoría de los vértices del FT valen 0, y en cada operación se modifican O(log N) vértices.

En lugar de declarar el Fenwick Tree como un array, ¡hacerlo como **unordered_map**!

```
unordered_map<long long, int> FT;
```

# Fenwick trees

Eso nos garantiza memoria O(M log N) donde M es la cantidad de llamadas a *setFT o getFT* (M = Q+U). El código de las funciones se mantiene idéntico.

Notemos que al usar un **unordered_map** (en contraposición a un simple map), estamos preservando la complejidad de las operaciones.

Sin embargo, se debe tener en cuenta que aumenta la constante computacional (tarda *un toque* más).

# Mejora copada 3

# Multidimensionalidad

# Fenwick trees

Supongamos que tenemos una matriz *implícita* de NxM y queremos poder sumarle valores a diferentes casillas y encontrar la suma de subrectángulos.

- ¡Los Fenwick Trees escalan fantásticamente a múltiples dimensiones!
- Las operaciones son $O((\log n)^k)$ donde k es la cant. de dimensiones

# Fenwick Trees: operaciones en 2 dimensiones

```
int FT[ MAXN ][ MAXM ];

// Devuelve la suma de A[1..a][1..b]

int getFT (int a, int b){
    int v = 0;
    for (int x = a; x; x -= x & -x)
        for (int y = b; y; y -= y & -y)
            v += FT[x][y];
    return v;
}
```

# Fenwick Trees: operaciones en 2 dimensiones

```
// Realiza A[a][b] += v


void setFT (int a, int b, int v){
    for (int x = a; x < MAXN; x += x & -x)
        for (int y = b; y < MAXM; y += y & -y)
            FT[x][y] += v;
}
```

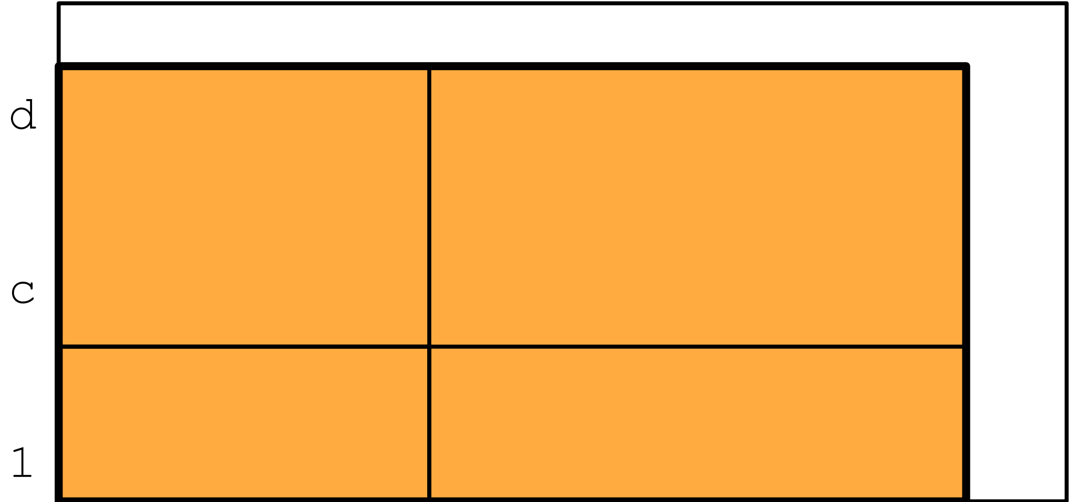
# Ejemplo

```
setFT(2, 4, 100); → A[2][4] = 100
setFT(5, 3, 200); → A[5][3] = 200

getFT(3, 4);  → 100
getFT(3, 3);  → 0
getFT(5, 3);  → 200
getFT(5, 4);  → 300
```

# Ejemplo

Para obtener la suma de A[a..b][c..d]

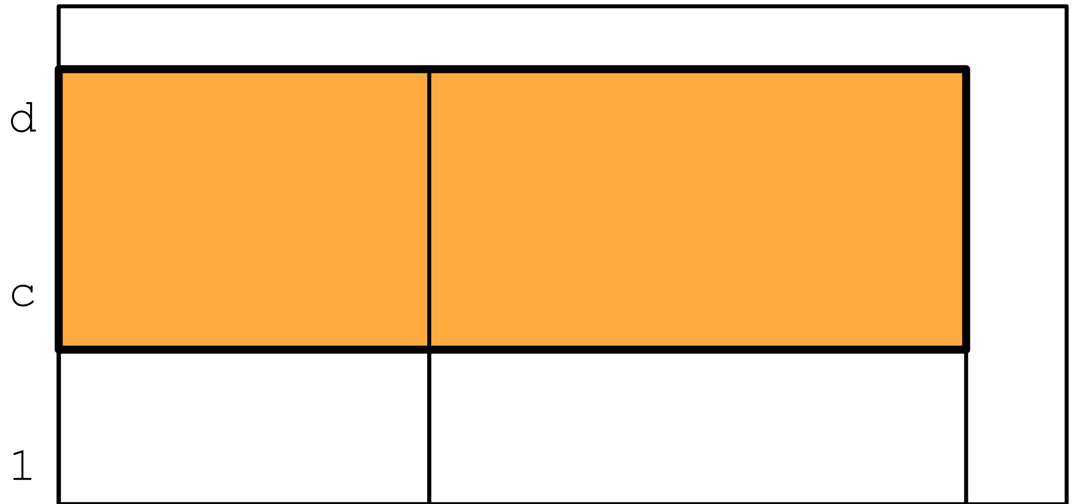
**get(b, d) -**
get(b, c-1) **-**
get(a-1, d) **+**
get(a-1, c-1)



# Ejemplo

Para obtener la suma de A[a..b][c..d]

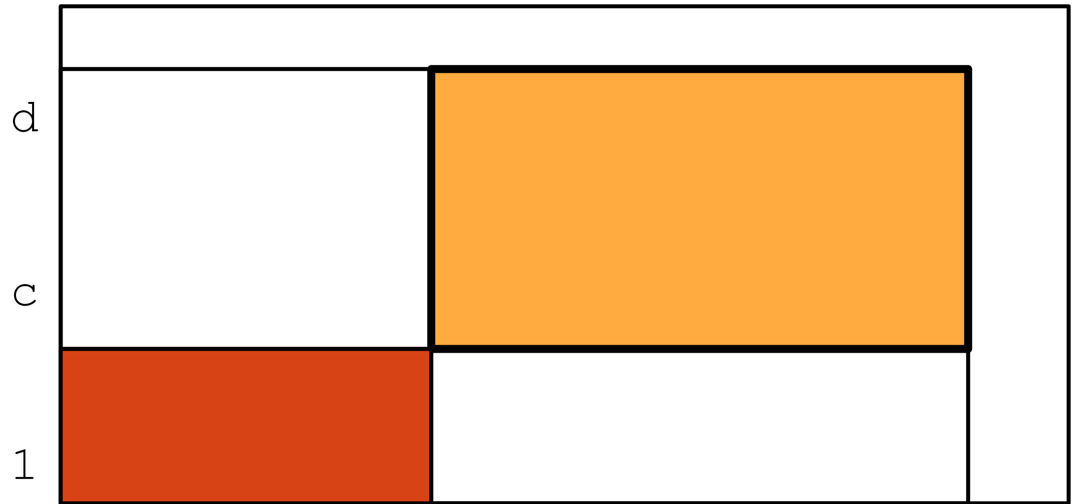
```
get(b, d) -
get(b, c-1) -
get(a-1, d) +
get(a-1, c-1)
```



# Ejemplo

Para obtener la suma de A[a..b][c..d]

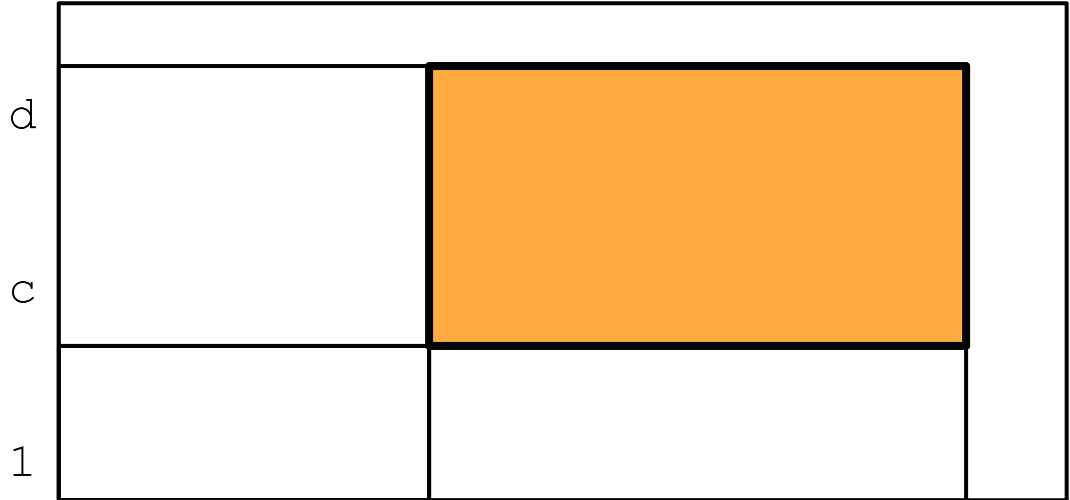
```
get(b, d) -
get(b, c-1) -
```
**get(a-1, d) +**
```
get(a-1, c-1)
```



# Ejemplo

Para obtener la suma de A[a..b][c..d]

get(b, d) **-**
get(b, c-1) **-**
get(a-1, d) **+**
**get(a-1, c-1)**



# Mejora copada 4

# Generalizaciones

# Fenwick trees

No hay necesidad de restringir la operación del Fenwick Tree a la suma, funciona con cualquier operación ★ que satisfaga:

- $a \star (b \star c) = (a \star b) \star c$ *(asociatividad)*
- existe $0_\star$ tal que $a \star 0_\star = a$ *(elemento neutro)*
- existe $\star^{-1}$ tal que $(a \star b) \star^{-1} a = b$ *(operación inversa)*

# Fenwick trees

Ejemplos:

- suma (+) con el cero (0) y la resta (-)
- producto (*) con el uno (1) y la división (/) en los reales sin 0
- xor (⊕) con el cero (0) y el xor otra vez (⊕)

Si p es un número primo,

- producto módulo p (* mod p) con el uno (1) y la división módulo p (/ mod p) en [1..p-1]

# Fenwick trees

Adicionalmente, si la operación ★ **no tiene inverso** (por ejemplo, "el mínimo" o "el divisor común mayor") se puede seguir usando con **updates monótonos**, y con queries en rangos que sean PREFIJOS (comenzando desde el índice 1).

*(por ejemplo, para el caso del mínimo, updates monótonos significa que el valor de cada update es más chico que el anterior; monótono significa que* $v_{viejo} \star v_{nuevo} = v_{nuevo}$*, en caso contrario el update no tendría efecto).*

# Ejemplo

Supongamos, $\star$ = ***mín***, *y* $0_{\star} = \infty$

```
setFT(10, 5); → A[10] = 5
setFT(10, 3); → A[10] = 3
setFT(4, 7);  → A[4]  = 7
getFT(3);      → ∞
getFT(4);      → 7
getFT(9);      → 7
getFT(10);     → 3
```

# Mejora copada 5

# Range Updates
# Range Queries

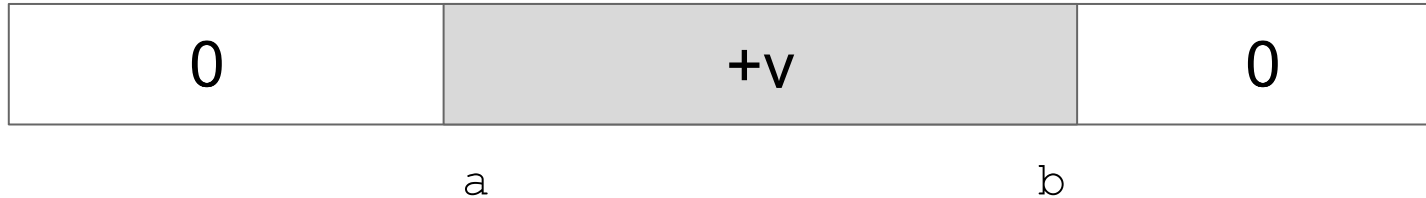
# Fenwick trees

Utilizando dos Fenwick Trees **en conjunto**, haciendo uso de las dos interpretaciones ya vistas, podemos desbloquear una tercera interpretación: **range updates - range queries**

Modificamos nuestras funciones para pasar el FT como argumento:

- `int getFT (int *FT, int x)`
- `void setFT (int *FT, int x, int v)`

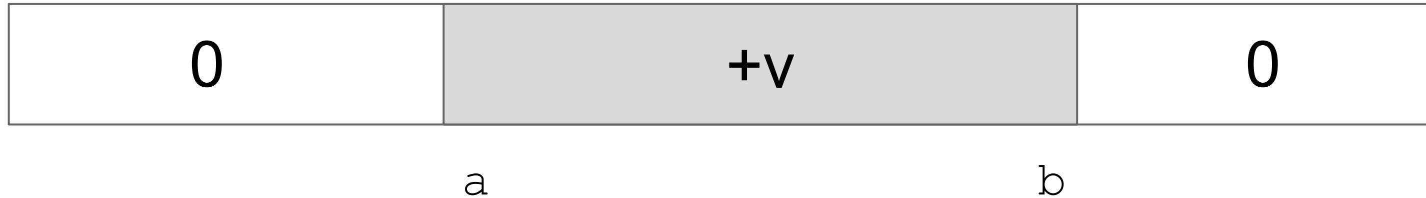
# Fenwick trees




Supongamos que sumamos un valor `v` a `A[a..b]`; luego, la suma de los elementos en `A[1..x]` debería ser:

- 0 : si $x < a$
- $v \cdot (x-a+1)$ : si $a \leq x \leq b$
- $v \cdot (b-a+1)$ : si $b < x$

# Fenwick trees




Es decir, puedo obtener cada valor restándole una constante a $A[x] \cdot x$, donde $A[x] = v$ para $x$ en $[a..b]$ y 0 en otro caso:

- $0 \quad = A[x] \cdot x - 0 \quad$ : si $x < a$
- $v \cdot (x-a+1) = A[x] \cdot x - v \cdot (a-1) \quad$ : si $a \leq x \leq b$
- $v \cdot (b-a+1) = A[x] \cdot x - v \cdot ((a-1)-b) \quad$ : si $b < x$

# Fenwick trees

Dado un x, para obtener el valor de A[x], utilizo un Fenwick Tree FT1 con la interpretación range update, point query.

Para saber cuánto le tengo que restar, utilizo otro Fenwick Tree, FT2, con la interpretación point update, range query.

# Fenwick trees

```
void update_range(int a, int b, int v){
    setFT (F1, a, v);
    setFT (F1, b+1, -v);

    setFT (F2, a, v*(a-1));
    setFT (F2, b+1, -b*v);
}
```

# Fenwick trees

```
int get_range(int a, int b){
    int p = b * getFT(F1, b) - getFT(F2, b);

    a = a-1;
    int q = a * getFT(F1, a) - getFT(F2, a);

    return p-q;
}
```

# Mejora copada 6

# Inverse Fenwick Tree

# Fenwick trees

Supongamos que hacemos updates con valores *positivos*, luego  decimos que tenemos un **Fenwick monótono**, ya que si `x` ≤ `y` entonces `getFT(x)` ≤ `getFT(y)`.

Los Fenwick Trees monótonos son **invertibles**, es decir, existe una función `invertFT(v)` que dado `v`, devuelve el menor `x` tal que `getFT(x)` ≥ `v`.

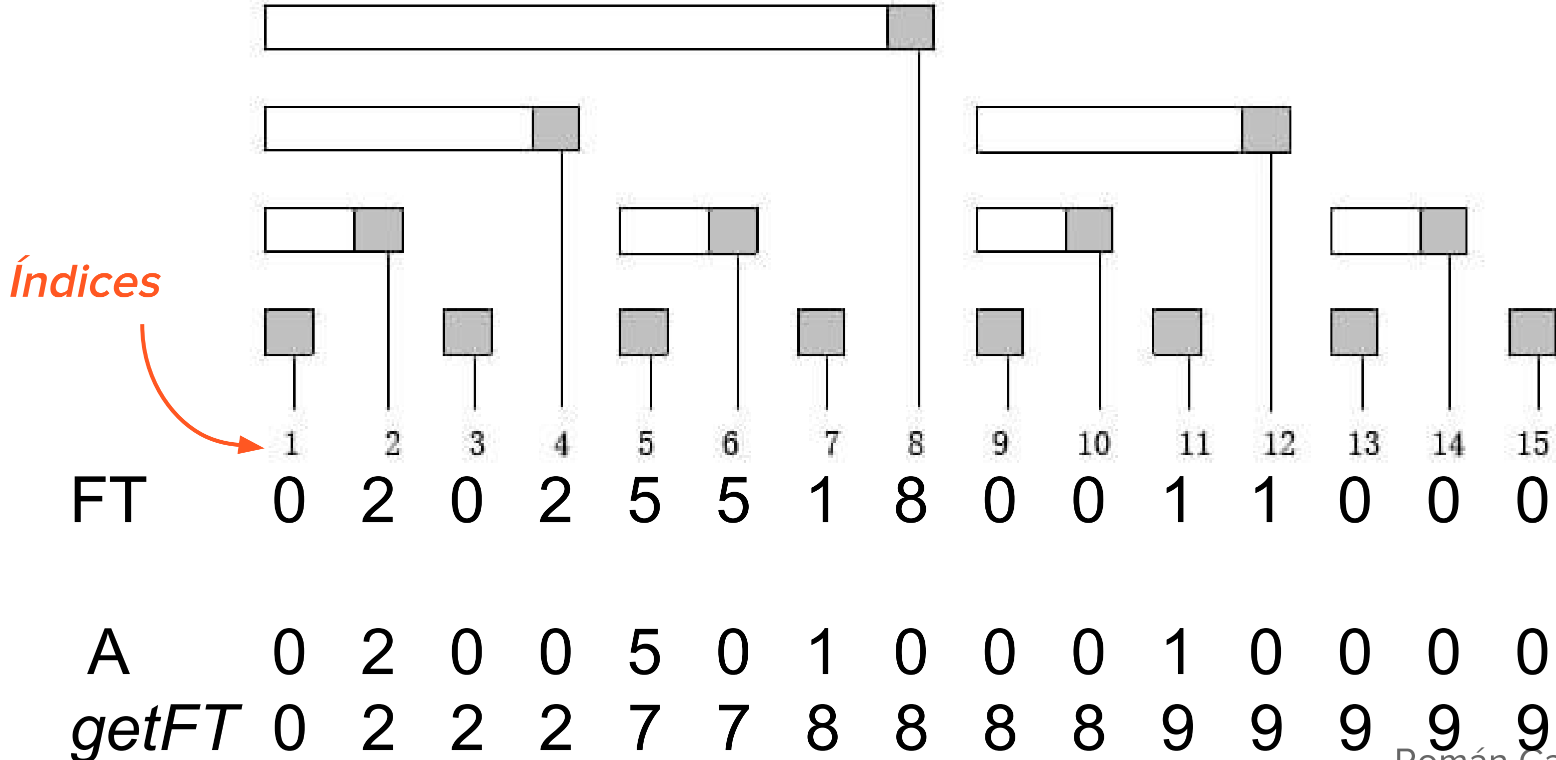
# Fenwick trees

Una forma de computar esta función es haciendo búsqueda binaria sobre getFT(x), pero esto nos llevaría a una solución $O((\log n)^2)$, que está más que bien.

Sin embargo, veamos que podemos llevarlo a $O(\log n)$, haciendo búsqueda binaria sobre los bits de los vértices del árbol. *(Advertencia: se debe entender bien la estructura del árbol de Fenwick)*

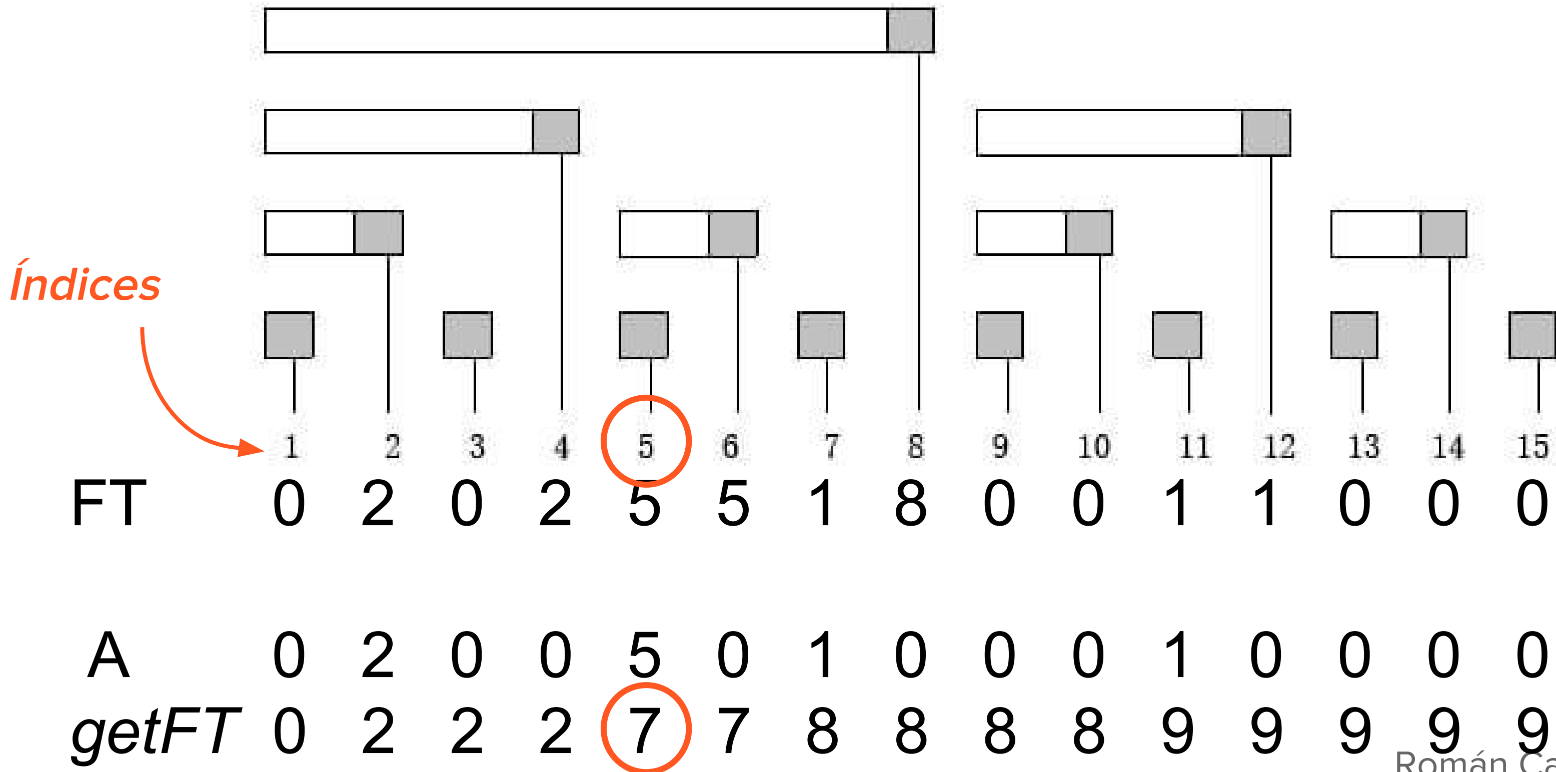
## InvertFT

¿Cuál es el menor índice x tal que `getFT(x)` ≥ 6 ?



# InvertFT

¿Cuál es el menor índice x tal que `getFT(x)` ≥ 6 ?



# Fenwick trees

```
int invertFT(int v){
    int x = 0;
    for(int d = 1<<(LOGMAXN-1); d; d>>=1)
        if( FT[x|d] < v ){
            x |= d;
            v -= FT[x];
        }
    return x+1;
}
```

# Ejemplo

Refiriéndonos al gráfico anterior, sería

```
setFT(11, 1); → A[11] = 1
setFT(5, 5);  → A[5]  = 5
setFT(7, 1);  → A[7]  = 1
setFT(2, 2);  → A[2]  = 2

invertFT(6);   → 5
getFT(5);       → 7
getFT(4);       → 2
```

# ¿Puedo combinar todas estas copadísimas mejoras como yo quiera?

sí.

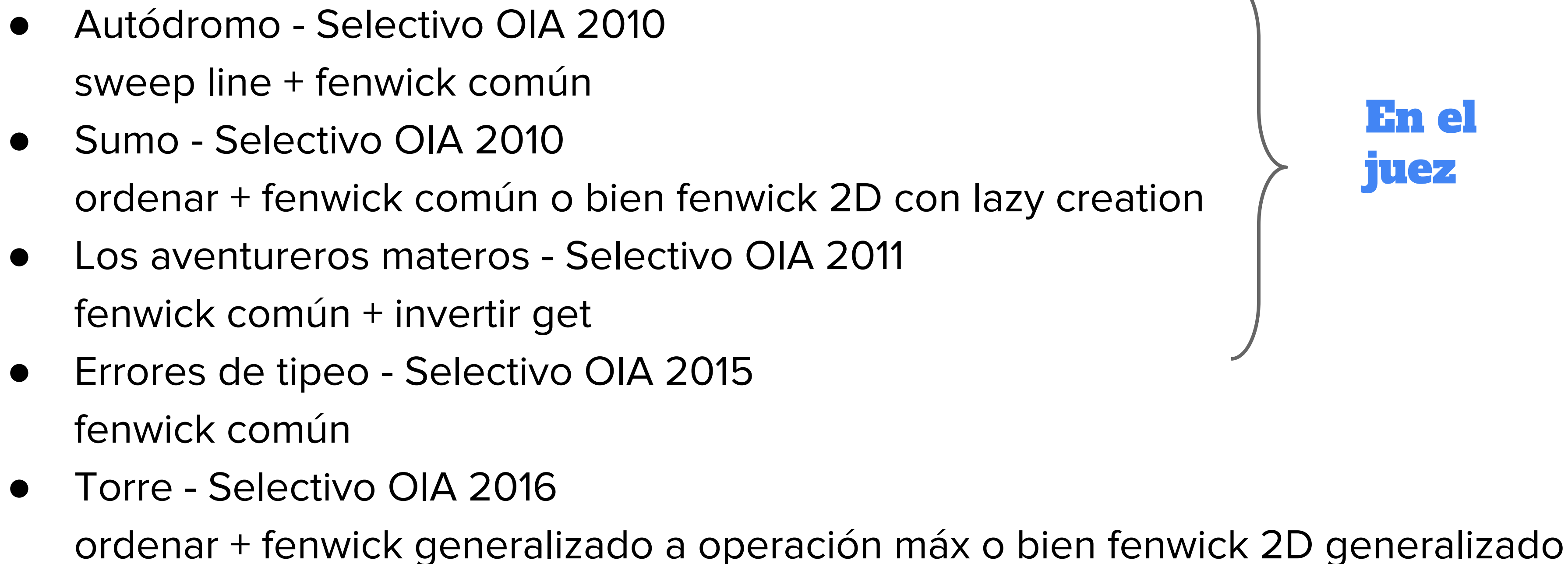
# ST vs FT

# Segment Trees vs Fenwick Trees

| | Segment Tree | Fenwick Tree |
|---|---|---|
| Memoria | 2N | N |
| Rapidez | O(log n) | O(log n) *(pero más rápido)* |
| Longitud de código | Más largo | Más corto |
| Operación natural | asociativa :) | asociativa e invertible :( |
| Lazy creation | fácil | trivial |
| Multidimensionalidad | complicado | trivial |
| Range u., range q. | muy complicado | bastante sencillo |
| Invertir el get | fácil | fácil |

# Problemas

# Problemas (OIA)



# Problemas (Externos)

- UVa 12532 - Interval Product
  fenwick común o bien dos fenwicks comunes o bien fenwick generalizado
- MATSUM - Spoj
  fenwick 2D
- HORRIBLE - Spoj
  Range update, range query
- KQUERY - Spoj
  Odenar + fenwick común
- Mishka and interesting sum - Codeforces round 365
  Dificil, Fenwick generalizado a xor

# ty (gracias)