Estructuras de búsqueda II

Autor: Cristian Jeldes

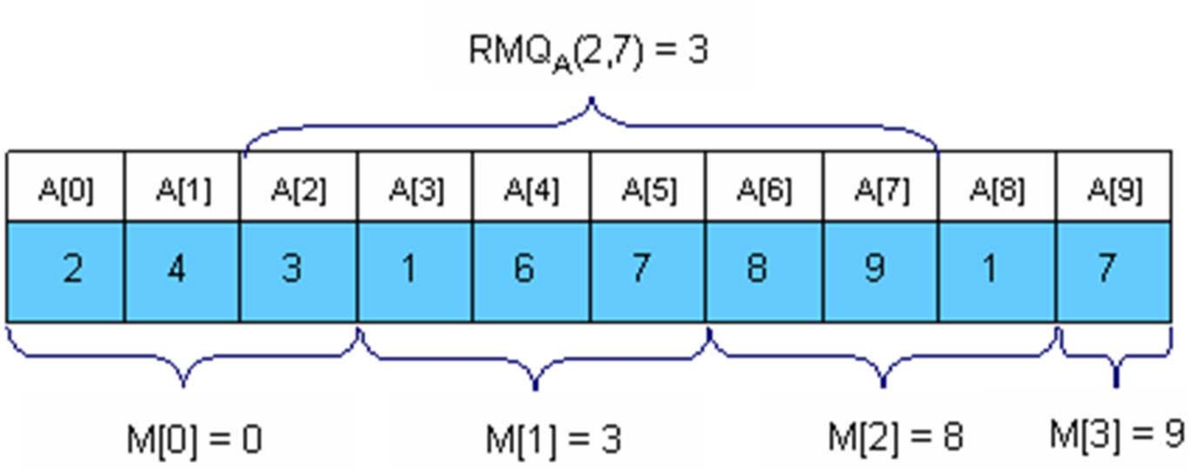
Range Minimum Query (RMQ)

El problema RMQ

$RMQ_A(2,7) = 3$

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	4	3	1	6	7	8	9	1	7

Dividir el espacio en SQRT



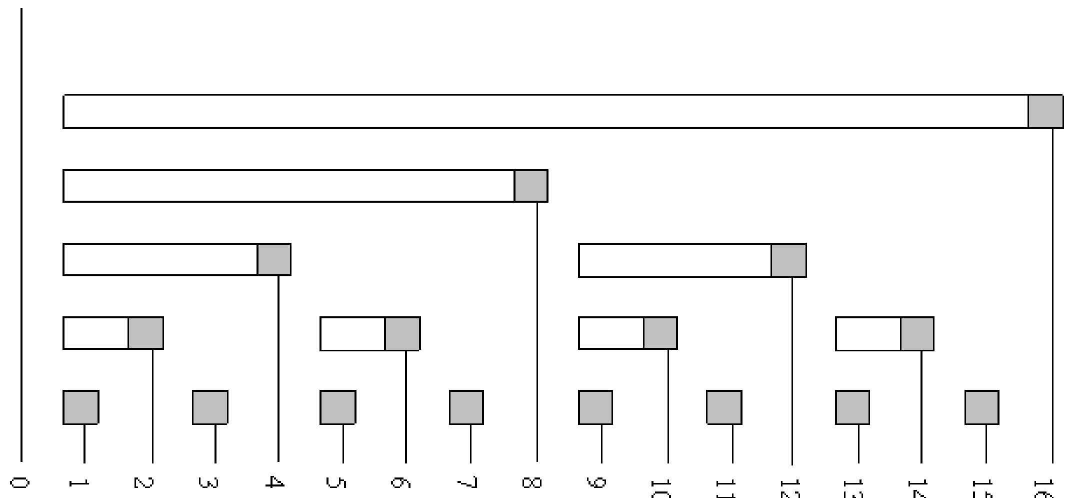
Estructuras para búsquedas

Fenwick Tree

Cada índice del arreglo puede ser representado por una suma de potencias de 2. En el mismo sentido , cada suma en un rango puede ser visto como la suma de dos sub-conjuntos.

Responsabilidades:

- Array[I] = Suma del indice **I** – 2^**r** + 1 hasta **I**
- I es el índice
- r es el LSB del índice
- LSB(I) = (I & (-I)) = r
- ^ es XOR no potencia



Fenwick Tree

Leer la suma acumulada a un punto

```
int read(int idx){
    int sum = 0;
    while (idx > 0){
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}
```

Complejidad
Log(n)

Fenwick Tree

Actualización individual y escalamiento de todo

```
void update(int idx ,int val){
    while (idx <= MaxVal){
        tree[idx] += val;
        idx += (idx & -idx);
    }
}
void scale(int c){
    for (int i = 1 ; i <= MaxVal ; i++)
        tree[i] = tree[i] / c;
}
```

Complejidad
Log(n)

Complejidad
n

Fenwick Tree – 2D

Consulta en 2D

```
int read(int x,int y){
    int sum= 0;
    while( x){
        int y1 = y;
        while(y1){
            sum += tree[x][y1];
            y1 -= y1 & -y1;
        }
        x -= x & -x;
    }
    return sum;
}
```

Complejidad
Log(n)*Log(n
)

Fenwick Tree – 2D

Actualización en 2D

```
void update(int x , int y , int val){
    int y1;
    while (x <= max_x){
        y1 = y;
        while (y1 <= max_y){
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
```

Complejidad
Log(n)*Log(n
)

Fenwick Tree With RMQ

Consulta por rango de mínimo

http://ioinformatics.org/oi/pdf/v9_2015_39_44.pdf

Efficient Range Minimum Queries using Binary Indexed Trees

Mircea DIMA[1], Rodica CETERCHI[2]

[1] *Hickery, Martir Closca st., 600206 Bacau, Romania*
[2] *University of Bucharest, Faculty of Mathematics and Computer Science 14 Academiei st., 010014 Bucharest, Romania*
e-mail: mircea@hickery.net, rceterchi@gmail.com

Abstract. We present new results on Binary Indexed Trees in order to efficiently solve Range Minimum Queries. We introduce a way of using the Binary Indexed Trees so that we can answer different types of queries, e.g. the range minimum query, in $O(\log N)$ time complexity per operation, outperforming in speed similar data structures like Segment/Range Trees or the Sparse Table Algorithm.

Keywords: binary indexed tree (BIT), least significant non-zero bit (LSB), range minimum query (RMQ).

Complejidad
Log(n)

Segment Tree

Segment Tree

[5, 2, 4, 6, 11, 8, 3, 2]

Segment Tree

[5, 2, 4, 6, 11, 8, 3, 2]

[2 |2]

Segment Tree

[5, 2, 4, 6, 11, 8, 3, 2]

[2 |2]

[2 |4][8 |2]

Segment Tree

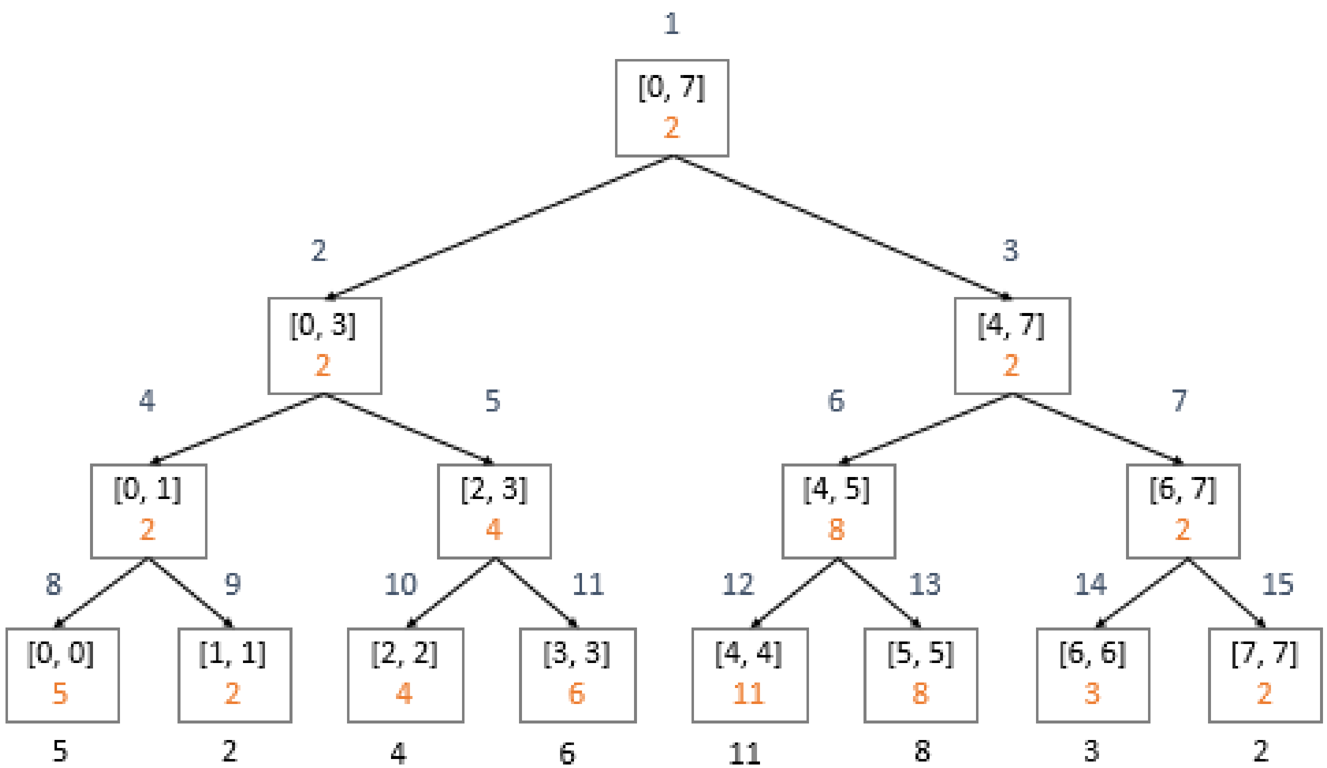
[2]

[2 |2]

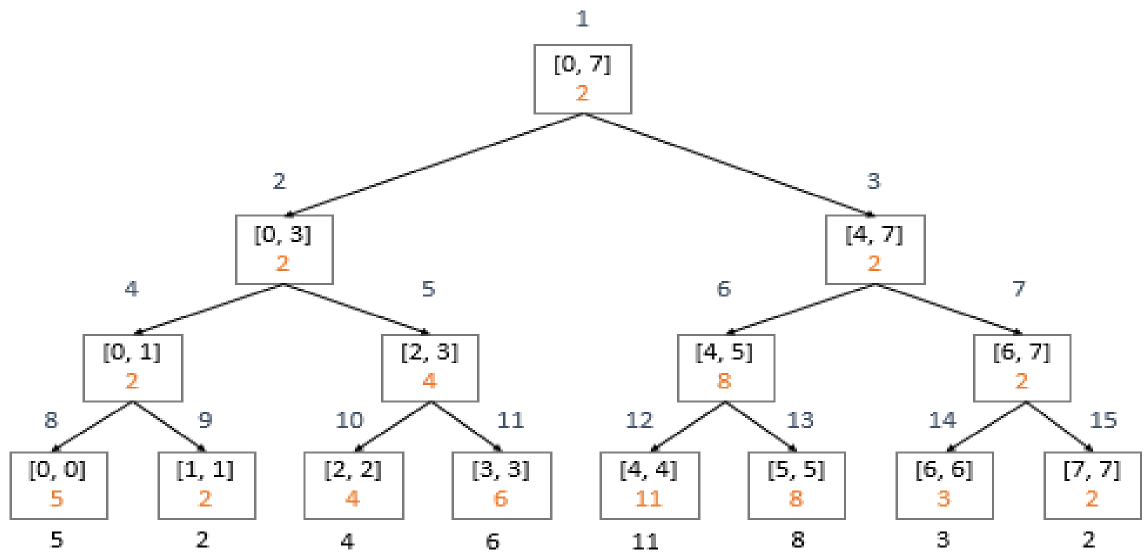
[2 |4][8 |2]

[5, 2, 4, 6, 11, 8, 3, 2]

Segment Tree



Segment Tree

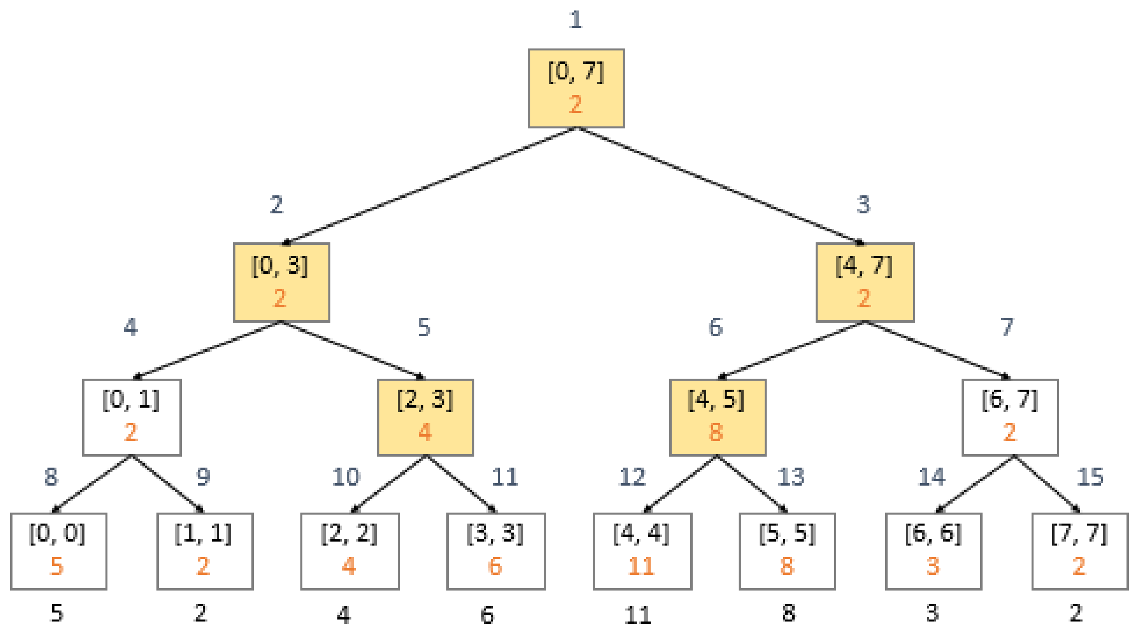


Rango padre: [l, r]

Rango hijo izquierdo: [l, (l + r) / 2]

Rango hijo derecho: [(l + r) / 2 +1, r]

Segment Tree - Operación QUERY[2,5]

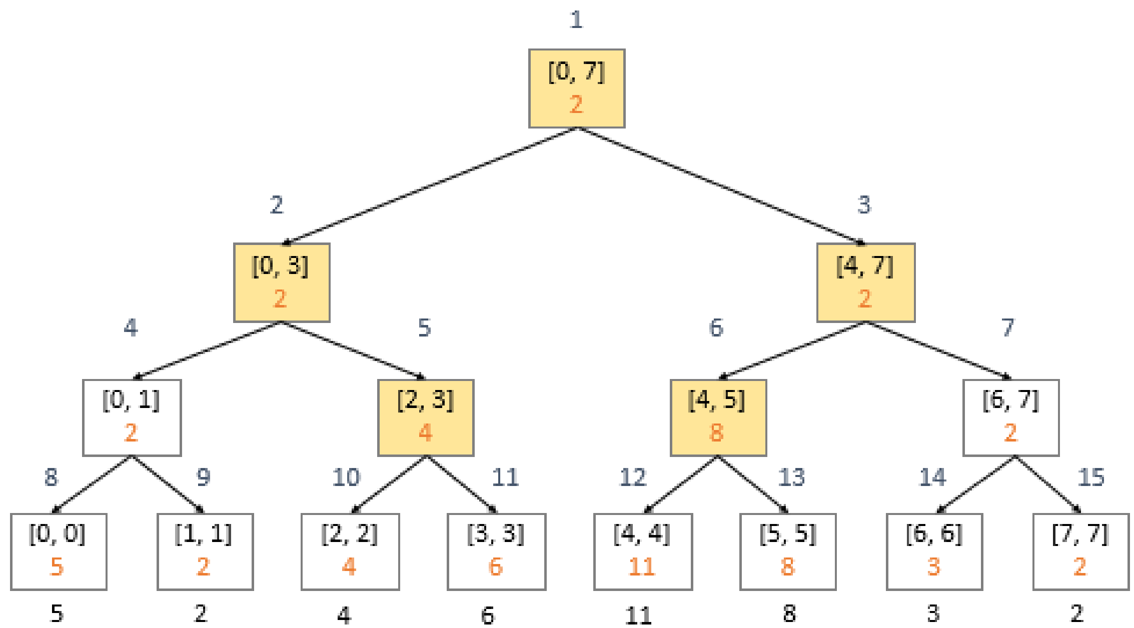


Rango padre: [l, r]

Rango hijo izquierdo: [l, (l + r) / 2]

Rango hijo derecho: [(l + r) / 2 +1, r]

Segment Tree - Operación QUERY[2,5]

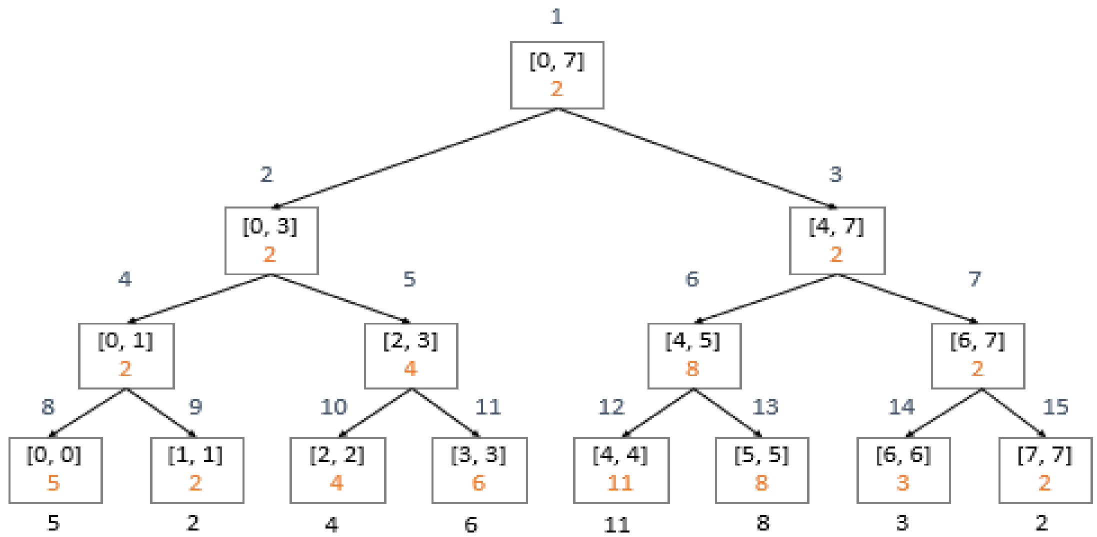


Rango padre: [l, r]

Rango hijo izquierdo: [l, (l + r) / 2]

Rango hijo derecho: [(l + r) / 2 +1, r]

Segment Tree – Construir uno



[Nulo,2,2,2,2,4,8,2,5,2,4,6,11,8,3,2]

Segment Tree – Construir uno

[5, 2, 4, 6, 11, 8, 3,2,
2,4,8,2,
2,2,
2]

Array[1] => Rango[1,1]
Array[2] => Rango[2,2]
Array[3] => Rango[3,3]
Array[4] => Rango[4,4]
Array[5] => Rango[5,5]
Array[6] => Rango[6,6]
Array[7] => Rango[7,7]
Array[8] => Rango[8,8]

Array[9] => Rango[1,2]
Array[10] => Rango[3,4]
Array[11] => Rango[5,6]
Array[12] => Rango[7,8]

Array[13] => Rango[1,4]
Array[14] => Rango[5,8]

Array[15] => Rango[1,8]

Segment Tree – Consulta

```
Query(nodo, inicio, fin, L, R): (L y R son rango de la consulta)
Si(R < inicio o fin < L):
        return neutro
Si(L <= inicio y fin <=R):
        return Array[nodo]
medio = (inicio + fin)/2
resultado1 = Query(2*nodo, inicio, medio, L, R)
resultado2 = Query(2*nodo+1, mid+1, fin, L, R)
return operacion(resultado1,resultado2)
```

Array[8] => Rango[1,1]
Array[9] => Rango[2,2]
Array[10] => Rango[3,3]
Array[11] => Rango[4,4]
Array[12] => Rango[5,5]
Array[13] => Rango[6,6]
Array[14] => Rango[7,7]
Array[15] => Rango[8,8]

Array[4] => Rango[1,2]
Array[5] => Rango[3,4]
Array[6] => Rango[5,6]
Array[7] => Rango[7,8]

Array[2] => Rango[1,4]
Array[3] => Rango[5,8]

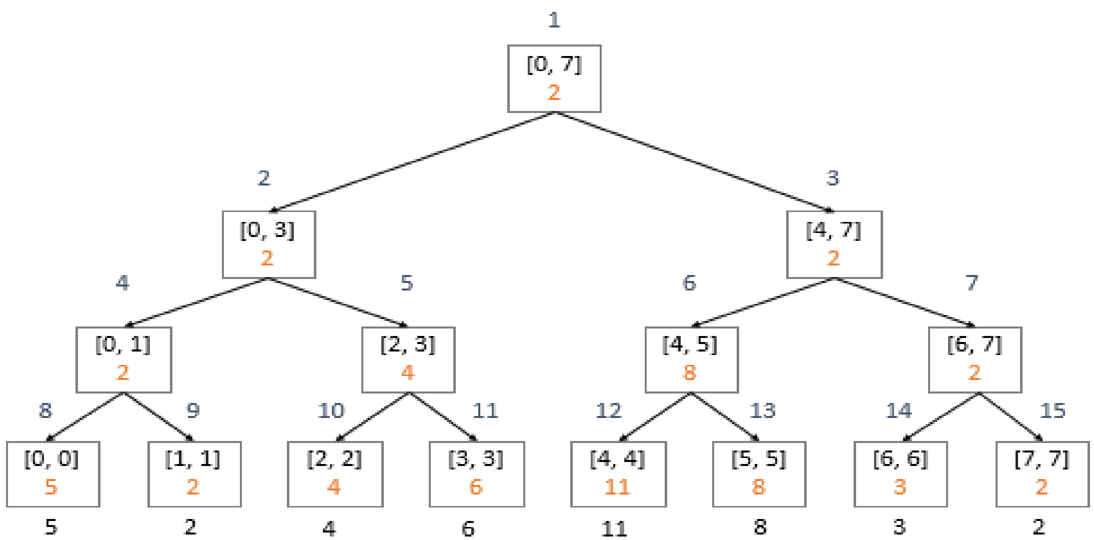
Array[1] => Rango[1,8]

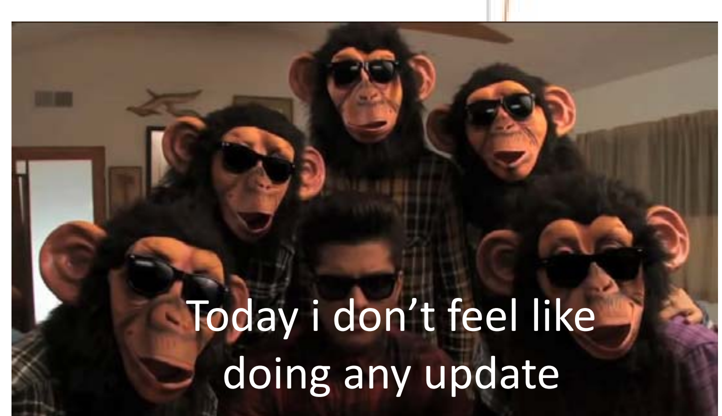
Segment Tree – Actualización

```
Query(nodo, inicio, fin, L, R,valor): (L y R son rango de la actualización)
Si(inicio>fin o inicio < R o fin < L):
        return
Si(inicio == fin):
        Array[nodo] = operacion(Array[nodo],valor)
        return
medio = (inicio + fin)/2
actualizacion(2*nodo, inicio, medio, L, R)
actualizacion(2*nodo+1, mid+1, fin, L, R)
Array[nodo] = operación(Array[2*nodo] , Array[2*nodo+1] )
```

Segment Tree Lazy

- Seamos LAZYS!
- Hagamos una actualización estrictamente cuando es necesario





Segment Tree Lazy

```
[           41          ]
[17        |24          ]
[7   |10  ][19  |5  ]
[5,2,4,6,11,8,3,2]
```

Actualizar rango [3,4] sumando 3

```
[          41*         ]
[17*       |24         ]
[7   |10*  ][19 |5     ]
[5,2,4*,6*,11,8,3,2]
```

Actualizar rango [3,4] sumando 3

Tree	Lazy
[41]	[0]
[17 |24]	[3 |0]
[7 |10][19 |5]	[0 |3][0 |0]
[5, 2, 4, 6, 11, 8, 3, 2]	[0, 0, 0, 3, 3, 0, 0, 0]

Segment Tree Lazy

```
updateRange(nodo, inicio, fin, l, r, valor)
  if(lazy[nodo] != neutro)
    arbol[nodo] += (fin - inicio + 1) * lazy[nodo]
    if(inicio != fin)
      lazy[nodo*2] += lazy[nodo]
      lazy[nodo*2+1] += lazy[nodo]
    lazy[nodo] = neutro
  if(inicio > fin o inicio > r o fin < l)
    return
  if(inicio >= l y fin <= r)
    arbol[nodo] += (fin - inicio + 1) * valor
    if(inicio != fin)
      lazy[nodo*2] += valor
      lazy[nodo*2+1] += valor
    return
  mid = (inicio + fin) / 2
  updateRange(nodo*2, inicio, mid, l, r, valor)
  updateRange(nodo*2 + 1, mid + 1, fin, l, r, valor)
  arbol[nodo] = arbol[nodo*2] + arbol[nodo*2+1]
```

Segment Tree Lazy

```
queryRange(nodo, inicio, fin, l, r)
  if(inicio > fin o inicio > r o fin < l)
    return 0
  if(lazy[nodo] != 0)
    arbol[nodo] += (fin - inicio + 1) * lazy[nodo]
    if(inicio != fin)
      lazy[nodo*2] += lazy[nodo]
      lazy[nodo*2+1] += lazy[nodo]
    lazy[nodo] = 0
  if(inicio >= l y fin <= r)
    return arbol[nodo]
  mid = (inicio + fin) / 2
  resultado1 = queryRange(nodo*2, inicio, mid, l, r)
  resultado2 = queryRange(nodo*2 + 1, mid + 1, fin, l, r)
  return resultado1+resultado2
```

Segment Tree
Generalización

Mínimo/Máximo elemento (Típico)

Mínimo/Máximo y cantidad de veces que aparece.

En este caso tenemos que agregar en cada nodo una variable para almacenar la cantidad de veces que aparece el mínimo/ máximo valor.

Segment Tree
Generalización

Máximo Común Divisor (GCD) / Mínimo Común Múltiplo (LCM)

Esta es una generalización y se obtiene de la misma forma vista hasta ahora, solo hay que llevar en cada nodo el GCD o LCM de los números en el intervalo del arreglo que corresponde.

Segment Tree
Generalización

Suma y/o producto

Cnodo almacena la suma de los valores en sus hijos, el producto puede ser implementado de forma similar.

Segment Tree

Ejercicios:

http://www.spoj.com/problems/BRCKTS/

http://www.spoj.com/problems/DQUERY/

http://www.spoj.com/problems/FREQUENT/

https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3977

Segment Tree

Bibliografía:

https://www.hackerearth.com/practice/notes/segment-tree-and-lazy-propagation/

Halim S., Halim F. - *Competitive Programming 3*