SEMINARIO DE PROGRAMACIÓN COMPETITIVA

ESTRUCTURA DE DATOS

POR: LEONARDO SANTELLA

AGENDA

Introducción

Fenwick Tree (o Binary Indexed Tree)

- Idea general del algoritmo
- Repaso de operaciones bit a bit
- ¿Qué problema resuelve?
- Analisis de Complejidad en tiempo
- Observaciones
- Pseudocodigo del algoritmo
- Ejemplo: Inversion counting
- Notas adicionales
- Ejercicios recomendados
- Preguntas/Dudas

Segment Tree

- Idea general del algoritmo
- Que problema resuelve?
- Analisis de Complejidad en tiempo
- Observaciones
- Pseudocodigo del algoritmo
- Ejemplo
- Notas adicionales
- Ejercicios recomendados
- Preguntas/Dudas

Referencias

INTRODUCCIÓN

- Acerca de mi
  - Licenciado en Ciencias de la computación
  - Finalista Mundial ACM-ICPC 2018 (Beijing)
  - SDE I en Amazon Madrid
  - Todo el contenido de esta clase esta basada en mi opinión
  - Nada de lo que diga es en nombre, ni en representación de Amazon
  - No me considero un experto en Programación Competitiva
  - Me apasiona la computación
  - Estoy felizmente casado con mi esposa Karina Alejandra
  - ID en la mayoría de websites de PC: lstd/cftryharder

INTRODUCCIÓN

- Esquema de las clases:
  - Idea general del algoritmo
  - ¿Qué problema resuelve?
  - Análisis de Complejidad en tiempo
  - Observaciones
  - Pseudocódigo del algoritmo
  - Ejemplo
  - Notas adicionales
  - Ejercicios recomendados
  - Preguntas/Dudas

BINARY INDEXED TREE

BINARY INDEXED TREE

Idea general

- Problema introductorio:
  - Dado un arreglo A de N números enteros y Q consultas de la forma [l, r] donde 1 <= l,r <= N. Calcula para cada consulta la suma del intervalo [l, r]
- ¿Ideas?

Respuesta correcta: Suma de prefijos/sufijos

BINARY INDEXED TREE

A = [1, 10, 19, -5, 20, 45]

Suma de prefijos = psum = [0, 1, 11, 30, 25, 45, 90] (indexado desde 0)

- Consultas:
  - l = 3 r = 5 => Respuesta = 19 - 5 + 20 = 34 = psum[5] - psum[2]
  - l = 1 r = 3 => Respuesta = 1 + 10 + 19 = 30 = psum[3] - psum[0]
  - l = 2 r = 6 => Respuesta = 10 + 19 - 5 + 20 + 45 = 89 = psum[6] - psum[1]

BINARY INDEXED TREE

- Es posible solucionar el problema anterior en un orden de complejidad menor que O(N) - O(1) - O(N)
- Una de las aplicaciones del BIT resuelve el problema anterior con la siguiente orden de complejidad O(NlogN) - O(logN) - O(logN)
- BIT nos permite resolver problemas que requieran actualizar los valores del arreglo de una manera más eficiente que la suma de prefijos

BINARY INDEXED TREE

Técnica: Suma de prefijos

- Construir el arreglo con la suma de prefijos O(N)
- Consultar un rango en la suma de prefijos O(1)
- Que pasa si existiese 2 tipos de consulta:
  - i v: actualiza la posición i-esima del arreglo asignándole el valor v
  - l r: imprime el valor de la suma del intervalo [l, r]
- ¿Ideas?

Respuesta Correcta: cada vez que procesemos una actualización en el arreglo habrá que reconstruir la suma de prefijos.
Conclusión: actualizar un elemento en la suma de prefijos es O(N)

BINARY INDEXED TREE

Definición: Dada una función **reversible** f y un arreglo A de enteros de tamaño N. BIT es una estructura de datos que:

- Permite calcular el valor de f en un intervalo del arreglo [l,r] en O(logN).
- Actualiza el valor de un elemento de A en O(logN).
- Requiere O(N) memoria. Utiliza exactamente la misma cantidad de memoria que el arreglo A.
- Es fácil (y rápido) de codear.

BINARY INDEXED TREE – REPASO DE OPERACIONES BIT A BIT

Operador AND (&):

- 3 & 4 = 011 & 100 = 0 = 000
- 7 & 4 = 111 & 100 = 4 = 100

Operador OR(|):

- 3 & 4 = 011 & 100 = 7 = 111
- 7 & 4 = 111 & 100 = 7 = 111

Operador XOR(^):

- 3 ^ 4 = 011 ^ 100 = 7 = 111
- 7 ^ 4 = 111 ^ 100 = 3 = 011

Complemento a1(~):

- ~3 = ~(011) = 100 = 4
- ~4 = ~(100) = 011 = 3

Complemento a2(-):

- -3 = -(011) = ca(3) + 1 = 101 = 5
- -4 = -(100) = ca(4) + 1 = 100 = 4

Bit shift hacia la izquierda (<<):

- (1 << 2) = 001 << 2 = 00100 = 4
- (2 << 2) = 010 << 2 = 01000 = 8

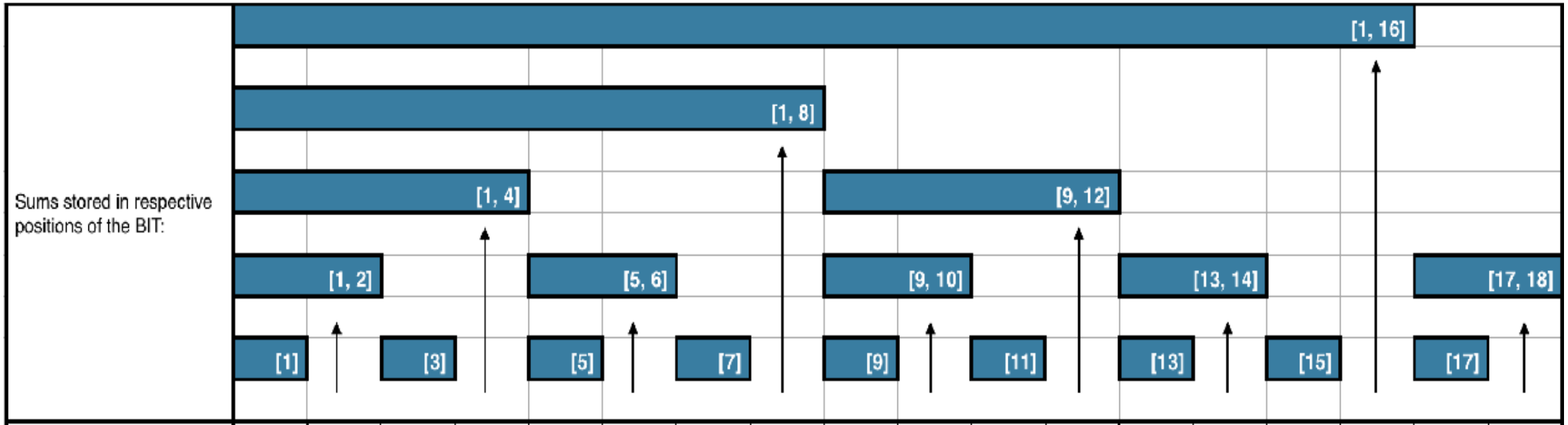
Bit shift hacia la derecha (>>):

- (1 >> 2) = 001 >> 2 = 00000
- (2 >> 1) = 010 >> 1 = 001

BINARY INDEXED TREE

- Chequear si el i-esimo bit esta encendido:
  - bool f(x, i): return x & (1 << i-1)
  - f(3, 3): return (011) & (100) -> false
  - f(3, 2): return (011) & (010) -> true
- Encender el i-esimo bit:
  - f(x, i): return x | (1 << i-1)
  - f(3, 3): return (011) | (1 << 2) = 111 = 7
  - f(3, 2): return (011) | (1 << 1) = 011 = 3
- Alternar el i-esimo bit:
  - f(x, i): return x ^ (1 << i-1)
  - f(3,3): return (011) ^ (1 << 2) = 111 = 7
  - f(3,2): return (011) ^ (1 << 1) = 001 = 1
- Obtener el bit menos significativo:
  - f(x): return (x)&(-1)
  - f(13): return (01101) &(10011) = 1 = 1
  - f(12): return (01100) & (10100) = 100 = 4
- Multiplicar por 2:
  - f(x) = return x << 1
  - f(5) = return (101) << 1 = 1010 = 10
- Dividir entre 2:
  - f(x) = return x >> 1
  - f(5) = return (101) >> 1 = 010 = 2
- Chequear si un numero es potencia de 2:
  - bool f(x) = return x & (x-1) == 0
  - f(5) = return (101) & (100) == 0 = false
  - f(4) = return (100) & (011) == 0 = true

BINARY INDEXED TREE – REPRESENTACION GRAFICA



Position in BIT:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Binary Representation:	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010

BINARY INDEXED TREE

Pseudocódigo:

```
int a[N], bit[N]
update(int idx, int val)
query(int l, int r)
query(int x)
lsb(int x)
```

BINARY INDEXED TREE

Pseudocódigo:

```
int a[N], bit[N]
int lsb(int x):
 return x&(-x)

update(int idx, int val):
  while(idx < N) do:
   bit[idx] = bit[idx] + val
   idx = idx + lsb(idx)
```

```
int query(l, r):
 return query(r) - query(l)
int query(x):
 int ans = 0
 while(x > 0) do:
   ans = ans + bit[x]
   x = x - lsb(x)
 return ans
```

BINARY INDEXED TREE

Análisis de complejidad:

- Actualizar una posición:
  - Al actualizar una posición, debemos actualizar todas posiciones que inicialmente eran 0 siguientes al lsb(x). El peor caso siempre será actualizar la primera posición. Ejemplo:
    - Si tenemos un arreglo de 10^3 elementos y actualizamos la primera posición:
    - Rule of thumb: 2^10 ~ 10^3...2^20 ~ 10^6...2^30 ~ 10^9…
    - 0000000001 seria el índice del primer elemento, por lo tanto tendríamos que actualizar 10 elementos del bit. log(10^3) = 3 (si es base 10) log_2(10^3) = 9,96...

BINARY INDEXED TREE

Análisis de complejidad:

- Consulta(x):
  - Consultar el rango 1-x toma log(N) en el peor caso. Al ir decrementando por el bit menos significativo, al igual que con la operación de actualización, estamos limitados por log(N)
  - Ejemplo:
    - Consulta(7) = Consulta(111). En este caso habrá que agregar el valor de bit[1] + bit[2] + bit[4].
    - Si tenemos un arreglo de 7 posiciones, es claro que log_2(7) = 2.807…
- Construcción del BIT:
  - Esto es equivalente a hacer N operaciones de actualización. Por lo tanto, la complejidad en tiempo de construir el BIT es O(NlogN)

BINARY INDEXED TREE

Problema de ejemplo: Conteo de inversiones en un arreglo

- Dado un arreglo A de N elementos, implementar un algoritmo que cuente cuantas **inversiones** hay en el arreglo.
- Dado un un índice i y un índice (1 <= i, j <= N) donde i < j, se dice que hay una **inversión** si A[i] > A[j].

A = [4, 3, 1, 2, 5, 6] => inversiones: (4, 3); (4,1); (4,2); (3, 1); (3, 2)

BINARY INDEXED TREE

- Como resolver este problema. ¿Ideas?
  - Bubble sort: Por cada swap la respuesta aumentaría +1
  - Se puede hacer de manera más eficiente?
    - Si. Mergesort es una opción. Es un algoritmo clásico para resolver el problema de las inversiones
    - La desventaja de esta solución es que codear un merge sort en una competencia puede ser arriesgado ya es es error prone.
  - Solución alternativa: BIT

BINARY INDEXED TREE

```
A = [4, 3, 1, 2, 5, 6]
bit = [0, 0, 0, 0, 0 ,0]
int ans = 0
for x in A:
 ans = ans + query(6-x)
 update(x, 1)
return ans
```

BINARY INDEXED TREE

Observaciones:

- Para utilizar BIT correctamente es necesario que la función/operación f sea **reversible**.
- Una función/operación es reversible si $a \circ b = c$, donde $\circ$ es nuestra operación, no solo podemos obtener el resultado, sino que también podemos "deshacerlo": por ejemplo, si sabemos que $7 \circ c = 18$, si $\circ$ es reversible, el valor de c seria único. La suma, por ejemplo, es reversible.
- Que operaciones no son reversibles? max(a,b), min(a, b), md5(x), a^b, a*b
- Sin embargo, hay maneras de obtener el RMQ utilizando 2 BITs. (Esto no forma parte de esta clase)

BINARY INDEXED TREE

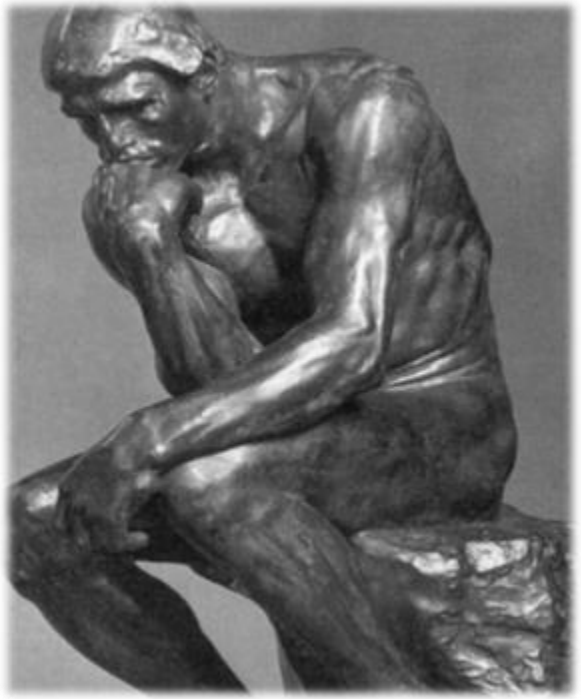
Notas adicionales:

- Esta estructura de datos fue creada por Peter M. Fenwick
- Este algoritmo suele utilizarse para generar tablas de frecuencias
- Fenwick Tree puede ser aplicado en el algoritmo de codificación aritmética
- Se le llama árbol porque puede ser representado gráficamente de la siguiente manera: https://visualgo.net/en/fenwicktree

BINARY INDEXED TREE

Problemas recomendados:

- https://www.spoj.com/problems/YODANESS/
- https://www.spoj.com/problems/NKTEAM/
- https://www.spoj.com/problems/DQUERY/
- https://www.spoj.com/problems/CTRICK/
- https://www.spoj.com/problems/TRIPINV/
- https://www.spoj.com/problems/SUMSUM/
- Más: https://cp-algorithms.com/data_structures/fenwick.html



¿PREGUNTAS?

SEGMENT TREE

SEGMENT TREE

Los segment trees son estructuras de datos que almacenan información sobre subarrays

Es el patrón de recursión del mergesort llevado a una estructura de datos

Es una de las estructuras de datos más útiles en la programación competitiva

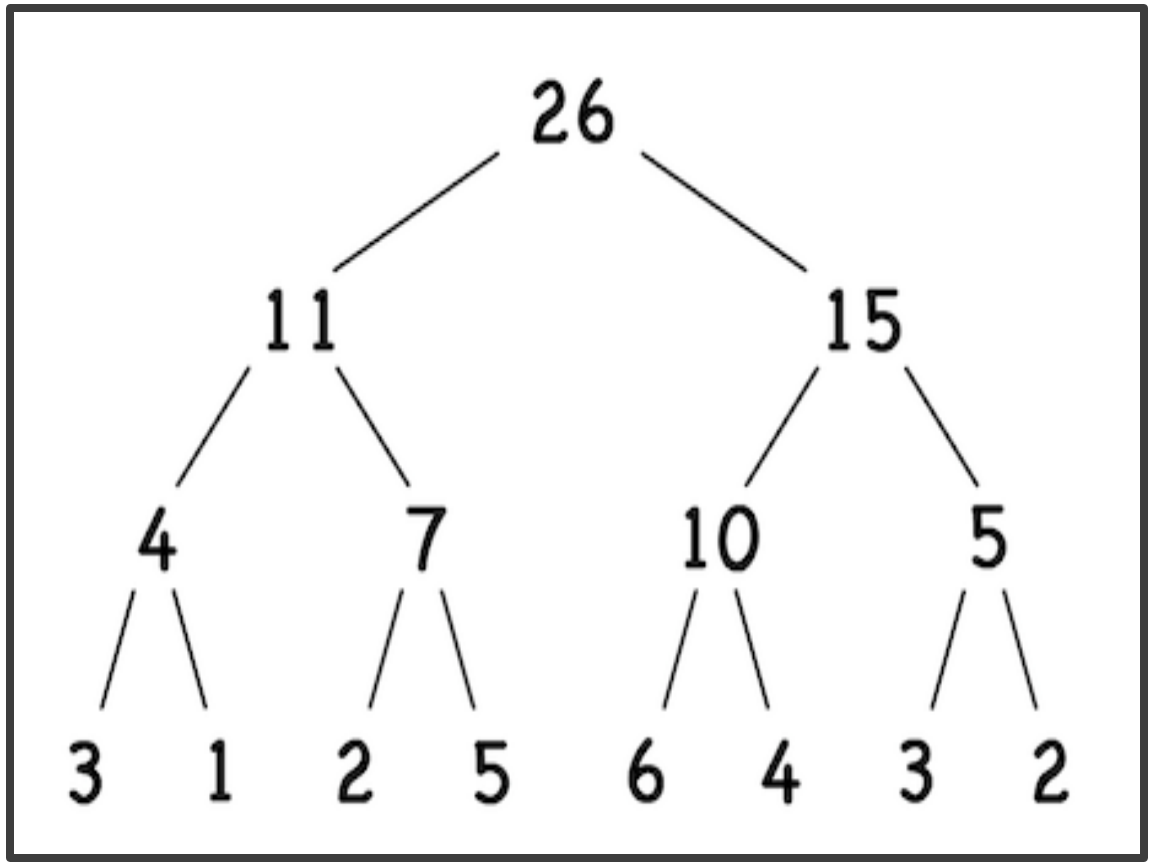
SEGMENT TREE

Problema introductorio

Dado un arreglo A de N enteros y se quieren realizar 2 tipos de operaciones sobre el:

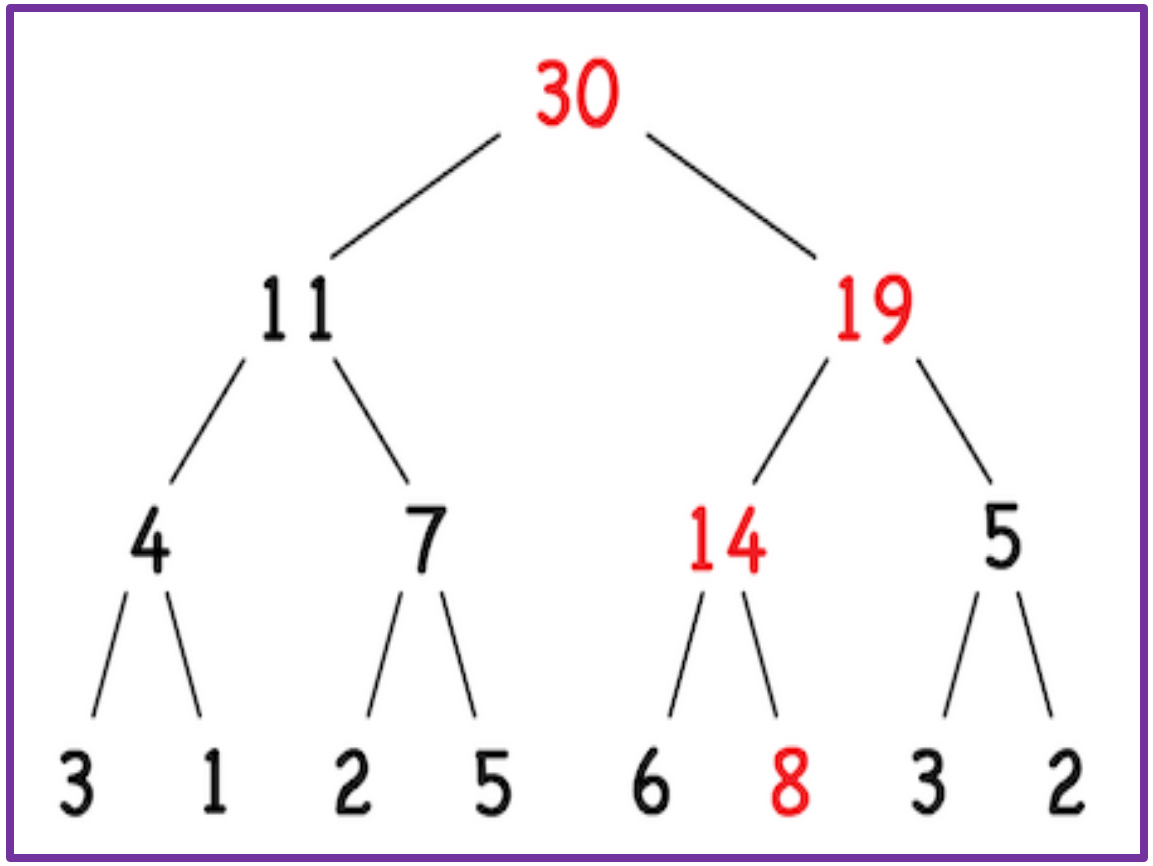
- set(i, v): Asignar un valor v a la posición i-esima del arreglo
- sum(l, r): Encontrar la suma en el segmento l a r-1

SEGMENT TREE – EJEMPLO:



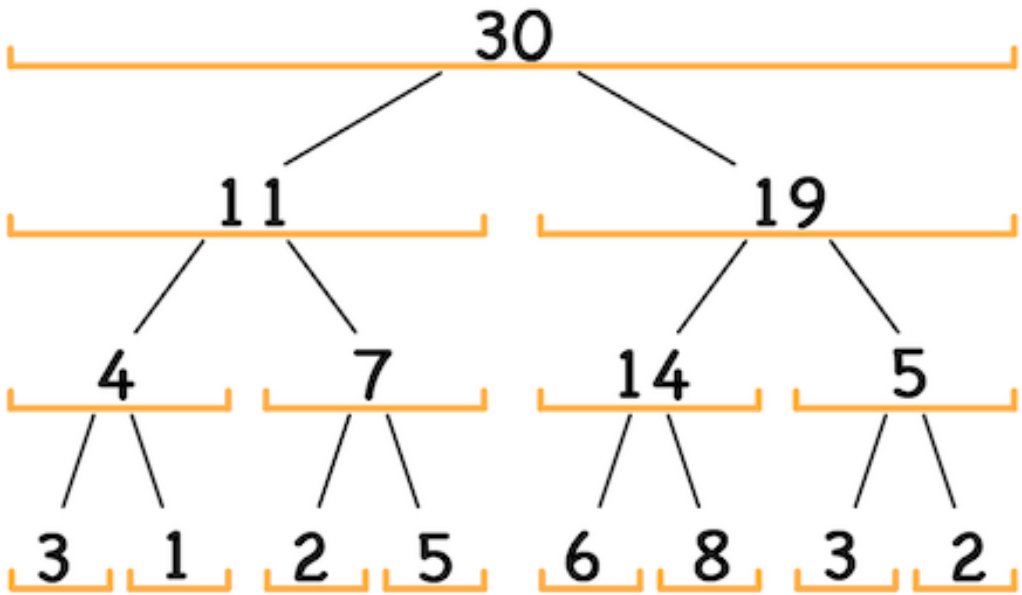
- A = [3, 1, 2, 5, 6, 4, 3, 2]

SEGMENT TREE



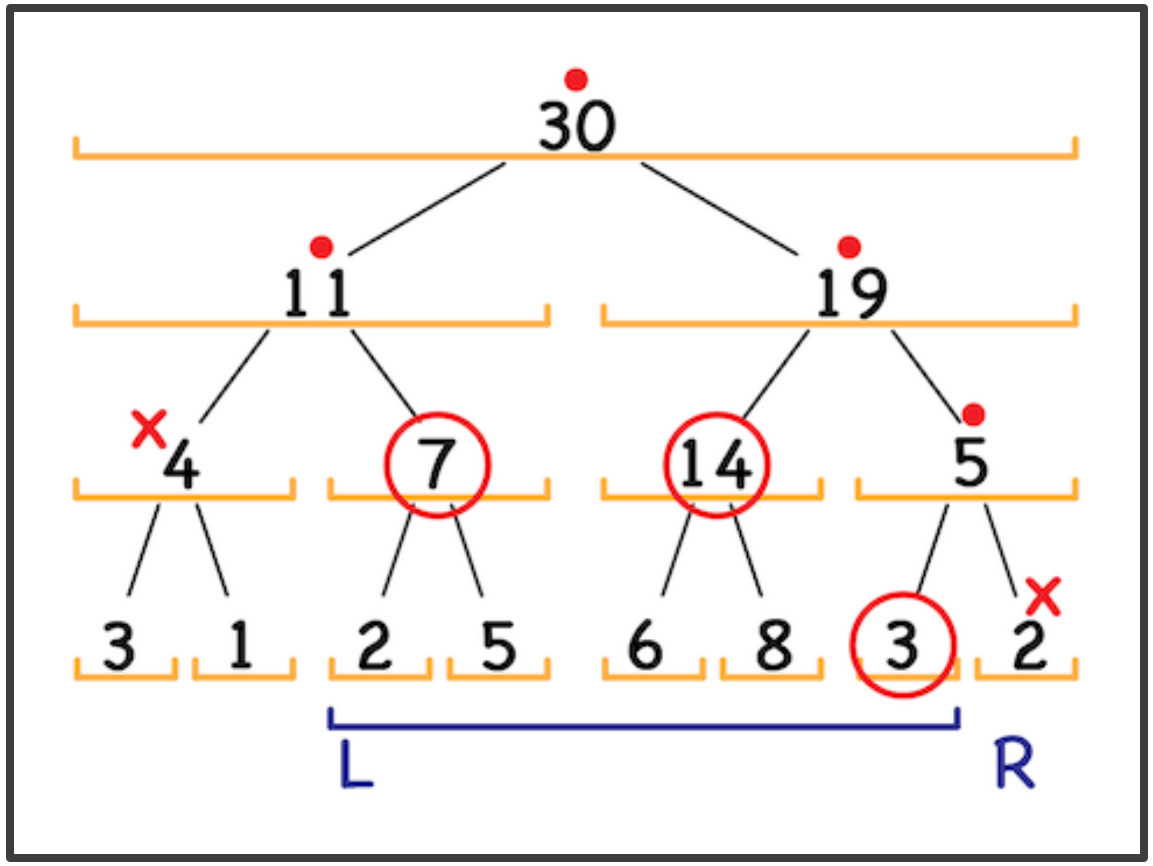
- Operación set(5, 8):

SEGMENT TREE



- Operación sum(l, r):

SEGMENT TREE



- Operación sum(3, 50):

SEGMENT TREE

- Análisis de la complejidad:
  - Al actualizar un elemento se sigue un camino desde el elemento actualizado hasta la raíz del árbol. Como el árbol es un árbol binario completo, el numero de operaciones realizadas por el computador será O(logN)
  - Para calcular la complejidad de la operación suma tenemos que darnos cuenta de que en cada nivel del árbol no puede haber más de un segmento que contenga 1 o los 2 limites. En base a esto, podemos deducir que máximo se van a recorrer 2 caminos desde la raíz hasta el nivel de las hojas. Esto es 2logN y esto termina siendo O(logN)

SEGMENT TREE

Observaciones:

- La función utilizada en el problema introductorio fue suma. Sin embargo, las operaciones que se pueden utilizar de manera similar son todas aquellas que sean **conmutativas y asociativas.**
- Ejemplo de funciones conmutativas y asociativas: min(x,y), max(x,y), lcm(x, y), gcd(x,y), a x b...etc
- Existen muchas variantes de Segment tree. Solo cubriremos una de ellas

SEGMENT TREE

Implementación C++:

```
struct segtree{
   int size;
   vector<long long> sums;
   void init(int n){...}
   void set(int i, long long v, int x, int lx, int rx) {...}
   void set(int i, long long v) {...}
   long long sum(int l, int r, int x, int lx, int rx) {...}
   long long sum(int l, int r) {...}
};
```

SEGMENT TREE

```
void init(int n) {
        size = 1;
        while(size < n)
            size *= 2;
        sums.assign(2*size, 0LL);
    }
```

```
void set(int i, long long v, int x, int lx, int
rx){
        if (rx - lx == 1) {
            sums[x] = v;
            return;
        }

        int mid = (lx + rx) / 2;
        if (i < mid) {
            set(i, v, 2*x+1, lx, mid);
        } else {
            set(i, v, 2*x+2, mid, rx);
        }
        sums[x] = sums[2*x+1] + sums[2*x+2];
    }

    void set(int i, long long v) {
        set(i, v, 0, 0, size);
    }
```

SEGMENT TREE

```
long long sum(int l, int r, int x, int lx, int rx) {
      if (lx >= r || l >= rx) return 0;
      if (lx >= l && rx <= r) return sums[x];
      int mid = (lx + rx) / 2;
      long long s1 = sum(l, r, 2*x+1, lx, mid);
      long long s2 = sum(l, r, 2*x+2, mid, rx);

      return s1 + s2;
  }

  long long sum(int l, int r) {
      return sum(l, r, 0, 0, size);
  }
```

SEGMENT TREE

Ejemplo: Problema C del calentamiento del SWERC - Rounds

1. Por cada elemento:
   a) Hacemos un incremento masivo de los rangos incR(0, i, -S), incR(i+1, N, -S) (recordar que es [i, j))
   b) Incrementamos el i-esimo elemento set(i, (N-1)*S)
   c) aux = RMQ(0, N)
   d) ans = max(ans, aux)

SEGMENT TREE

Notas Adicionales:

- Esta estructura de datos es distinta a esta: https://en.wikipedia.org/wiki/Segment_tree . Son similares, pero no del todo iguales.
- Existe una variante del ST permiten persistir los estados entre cada update (persistent ST): https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/
- Existe una variante que es "perezosa" en las operaciones de actualización (lazy propagation ST): https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/
- Aplicaciones: RMQ, RSQ, Intersección de segmentos, conteo de inversiones,

SEGMENT TREE

Ejercicios recomendados:

- https://codeforces.com/edu/course/2/lesson/4/1/ Todos los ejercicios, de los steps
- https://codeforces.com/edu/course/2/lesson/5 Todos los ejercicios, de los steps

REFERENCIAS

- https://cp-algorithms.com/data_structures/fenwick.html
- https://cp-algorithms.com/data_structures/segment_tree.html
- https://www.hackerearth.com/practice/notes/binary-indexed-tree-or-fenwick-tree/
- https://www.topcoder.com/thrive/articles/Binary%20Indexed%20Trees
- https://codeforces.com/edu/course/2/lesson/4
- https://codeforces.com/edu/course/2/lesson/5
- https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/
- https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/
- https://www.youtube.com/watch?v=kPaJfAUwViY&t=3102s
- https://www.youtube.com/watch?v=Tr-xEGoByFQ&t=2169s

AGRADECIMIENTOS

¡GRACIAS!

HAPPY CODING A TODOS