

Segment Trees in Algorithmic Problems

Piotr Szczepaniak

June 6, 2025

Contents

1	Foundations of Segment Trees	1
1.1	Segment Tree with point update and range query	2
1.2	Segment Tree with range update and range query	4
2	A mathematical approach to segment trees	7
2.1	Monoids	8
2.2	Homomorphism	8
2.3	Endomorphism	9
2.4	Application of Monoids to Segment Trees	9
2.5	Application of Endomorphisms to Segment Trees	11
3	Binsearch on Segment Tree	14
4	Max prefix sum on Segment Tree	16
5	Sweeping Line	18

Abstract

This document provides an overview of segment trees. In the first place I will describe some algebraic topics which are necessary for better understanding how and why segment trees works. This knowledge will be useful for reading the rest of the paper where We will dive into different kinds of trees. For each structure, I will explain how it work and how to apply it to problems. Then, I will look at each structure's time complexity and space complexity.

1 Foundations of Segment Trees

A segment tree is a binary tree used for storing information about segments. To efficiently retrieve or update informations about elements stored in segment tree we can perform various operations most common of which are query/update on point or range. One of the examples can be maximum value of elements in given range or sum of elements in given range.

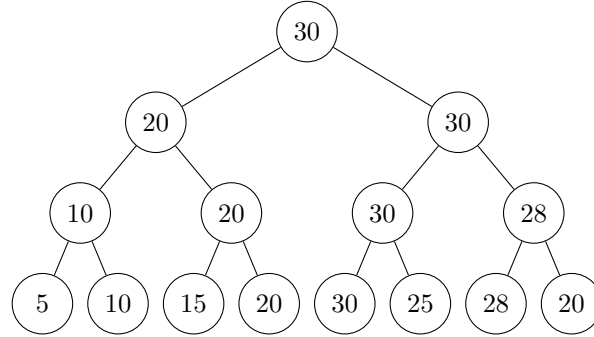


Figure 1: Example of a segment tree with maximum value of elements in given range.

1.1 Segment Tree with point update and range query

To illustrate use case of segment tree we will construct tree with max value on segment. Let's say we are given an array $A = [5, 10, 15, 20, 30, 25, 28, 20]$ of length $n = 8$. For now let's assume that the input array is of size $n = 2^k$ where k is integer (for different sizes of input we will fill input array with neutral elements (see section 2) to make it's length a power of 2). The height of tree is $h = \log_2 n$. Let's define $dep(i)$ as depth of node i in our tree. We can see that $dep(root) = 0$ and $dep(leaf) = h$. We want to be able to perform the following operations:

- **Build structure**

We will create a segment tree from an array. To build a segment tree, we need to create a binary tree where each node will store the maximum value of elements in its subtree.

Algorithm 1 Build Segment Tree for Maximum on Segment (Iterative)

```

procedure BUILDTREE(arr, seg)
    for  $i = 0$  to  $n - 1$  do                                ▷ Fill leaves of the segment tree
         $seg[n + i] \leftarrow A[i]$ 
    end for
    for  $i = n - 1$  downto  $1$  do                            ▷ Calculates nodes from bottom to top
         $seg[i] \leftarrow \max(seg[2 \times i], seg[2 \times i + 1])$ 
    end for
end procedure

```

- **Point update**

Now let's update single value in the array and update the tree. We will change the value of $A[1]$ from 10 to 35. To update the tree we need to

change the value of the leaf node and then update all the parent nodes up to the root.

Algorithm 2 Point Update on Segment Tree

```

procedure UPDATE(seg, index, value)
    index  $\leftarrow$  index + n                                 $\triangleright$  Shift index to leaf
    arr[index]  $\leftarrow$  value                                $\triangleright$  Update the value at the leaf
    while index > 1 do                                      $\triangleright$  Update the parent nodes
        index  $\leftarrow$   $\lfloor$ index/2 $\rfloor$ 
        arr[index]  $\leftarrow$  max(arr[2  $\times$  index], arr[2  $\times$  index + 1])
    end while
end procedure

```

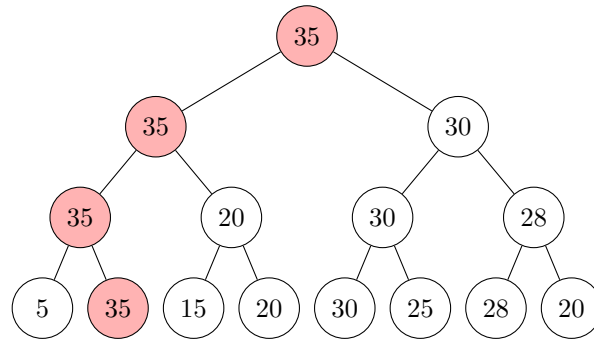


Figure 2: Example of point update.

- Range query

Now let's say we want to find the maximum value in the range $A[2 : 7]$. To do this we need to traverse the tree from the root to the leaves and find nodes that are in the range. Then we will get max the values of these nodes to get the final result. To get the result for range $A[2 : 7]$ we call *RangeQuery(seg, 1, 1, 8, 2, 7)*.

Algorithm 3 Range Maximum Query on Segment Tree (Recursive)

```
1: procedure RANGEQUERY(seg, index, l, r, a, b)    ▷ index: current node
   index in seg  ▷ [l, r]: segment represented by current node  ▷ [a, b]: query
   range
2:   if  $b < l$  or  $a > r$  then
3:     return  $-\infty$                                 ▷ No overlap
4:   else if  $a \leq l$  and  $r \leq b$  then
5:     return  $seg[index]$                             ▷ Total overlap
6:   else
7:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
8:      $left \leftarrow \text{RANGEQUERY}(seg, 2 \cdot index, l, mid, a, b)$ 
9:      $right \leftarrow \text{RANGEQUERY}(seg, 2 \cdot index + 1, mid + 1, r, a, b)$ 
10:    return  $\max(left, right)$ 
11:   end if
12: end procedure
```

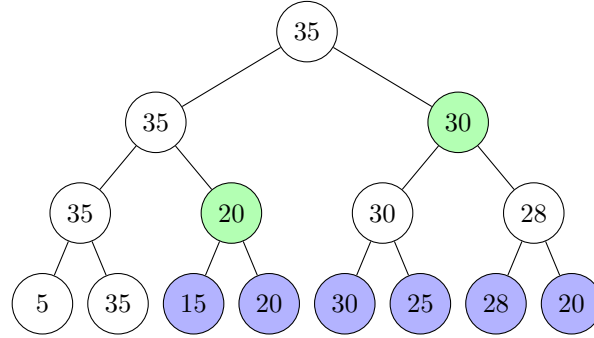


Figure 3: Example of range query. Blue leafs represents subarray $A[2 : 7]$. Green nodes represents nodes where ranges totally overlap and we can get max value. The result of the query is 30.

1.2 Segment Tree with range update and range query

A Little more complex version of segment tree is a segment tree with range update and range query which require **lazy propagation**. We will consider a segment tree with sum of elements in given range. Apart from values in tree we will also store lazy values in each node. The lazy value is used to delay the update of a node until it is needed.

- **Build structure** Only difference from previous example is initializing the lazy value for all nodes

Algorithm 4 Build Segment Tree for Sum on Segment (Iterative)

```
procedure BUILDTREE(arr, seg)
  for  $i = 0$  to  $n - 1$  do                                 $\triangleright$  Fill leaves of the segment tree
     $\{seg[n + i] \leftarrow A[i], 0\}$                          $\triangleright$  second value is for lazy propagation
  end for
  for  $i = n - 1$  downto  $1$  do                             $\triangleright$  Calculates nodes from bottom to top
     $seg[i] \leftarrow \{seg[2 \times i] + seg[2 \times i + 1], 0\}$ 
  end for
end procedure
```

- **Range update**

In this case lazy value is a value that we want to add to all elements in the range. Another way to think about this is that we want to add lazy value to all elements in array which are represented by all leaves in subtree of given node.

Algorithm 5 Range Update on Segment Tree

```
procedure QUERYUPDATE(seg, lazy index, l, r, a, b, value)  $\triangleright$  Value: value
  we wanted to add to each element in array
   $size = r - l + 1$ 
  if  $b < l$  or  $a > r$  then
    return
  else if  $a \leq l$  and  $r \leq b$  then
     $seg[index] \leftarrow seg[index] + value * size$ 
     $lazy[index] \leftarrow lazy[index] + value$ 
    return
  else
     $seg[index * 2] \leftarrow seg[index * 2] + value * size$   $\triangleright$  Update left child
     $lazy[index * 2] \leftarrow lazy[index * 2] + value$   $\triangleright$  Push lazy to left child
     $seg[index * 2 + 1] \leftarrow seg[index * 2 + 1] + value * size$   $\triangleright$  Same to right
     $lazy[index * 2 + 1] \leftarrow lazy[index * 2 + 1] + value$ 
     $lazy[index] \leftarrow 0$   $\triangleright$  Reset lazy value for current node
     $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
     $left \leftarrow \text{QUERYUPDATE}(seg, 2 \cdot index, l, mid, a, b, value)$ 
     $right \leftarrow \text{QUERYUPDATE}(seg, 2 \cdot index + 1, mid + 1, r, a, b, value)$ 
     $seg[index] \leftarrow seg[index * 2] + seg[index * 2 + 1] + lazy[index] * size$ 
   $\triangleright$  Update current node after lazy propagation
  end if
end procedure
```

How this algorithm works:

1. If the current node is fully in range update the sum of the node and set the lazy value. Sum - $value * size$ where $size$ is the number of

array elements (leafs) in the range of this node, Lazy is equal to *value*, which we will later push down to child nodes.

2. If not fully in range push the lazy value down to the child nodes and update their sum and also reset the lazy value for the current node.
3. Call the function recursively for the left and right child nodes.
4. Finally update the sum of the current node with the sum of its children.

On the following example we will update the range $A[0 : 5]$ by adding 10 to all elements in this range where initial array is $A = [5, 10, 15, 20, 30, 25, 20, 20]$.

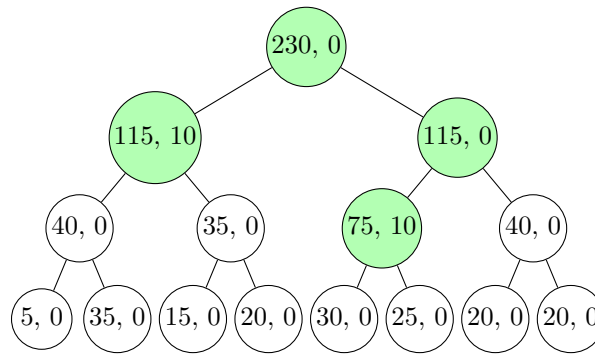


Figure 4: This is state of the tree after range update. Green nodes were changed during the update.

- **Range query** We will try to retrieve the sum of some given range. The algorithm for this will be similar to the one for range update. First we will check if the current node is fully in range. If it is we add the sum of this node, propagating lazy values along the way.

Algorithm 6 Range Query on Segment Tree

```
procedure RANGEQUERY(seg, lazy, index, l, r, a, b, value)
  if  $b < l$  or  $a > r$  then
    return 0
  else if  $a \leq l$  and  $r \leq b$  then
    return seg[index]
  else
    size = r - l + 1
    seg[index * 2]  $\leftarrow$  seg[index * 2] + value * size  $\triangleright$  Update left child
    lazy[index * 2]  $\leftarrow$  lazy[index * 2] + value  $\triangleright$  Push lazy to left child
    seg[index * 2 + 1]  $\leftarrow$  seg[index * 2 + 1] + value * size  $\triangleright$  Same to right
    lazy[index * 2 + 1]  $\leftarrow$  lazy[index * 2 + 1] + value
    lazy[index]  $\leftarrow$  0  $\triangleright$  Reset lazy value for current node
    result = 0
    mid  $\leftarrow$   $\lfloor \frac{l+r}{2} \rfloor$ 
    result += RANGEQUERY(seg, 2 * index, l, mid, a, b, value)
    result += RANGEQUERY(seg, 2 * index + 1, mid + 1, r, a, b, value)
    seg[index]  $\leftarrow$  seg[index * 2] + seg[index * 2 + 1] + lazy[index] * size
    return result
  end if
end procedure
```

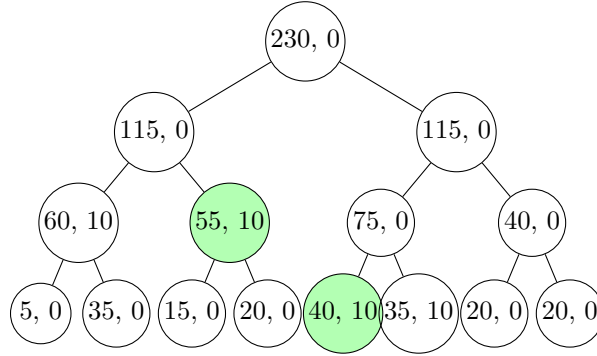


Figure 5: This is state of the tree after range query on range [2: 4]. Green nodes were we took values form to the sum. Result of query is 95.

2 A mathematical approach to segment trees

In this section we will try to understand how segment trees work and why they are so efficient. In order to do this we will need to understand some basic algebra that is used in segment trees.

2.1 Monoids

A monoid $(S, *, e)$ is a set equipped with an associative binary operation $S \times S \rightarrow S$ and an identity element e .

- **Associativity**

For all $a, b, c \in S$, $(a * b) * c = a * (b * c)$.

- **Identity element**

There exists an element $e \in S$ such that for all $a \in S$, $a * e = e * a = a$.

- **Commutativity**

Monoid is called **commutative** if for all $a, b \in S$, $a * b = b * a$.

Examples of monoids:

- **Natural numbers under addition, $(\mathbb{N}, +, 0)$**

In this simple example we can see that addition is associative and commutative. Adding 0 (neutral element) to any of the numbers will not change the result of the operation.

- **Strings over an alphabet, $(\Sigma^*, \cdot, \epsilon)$**

Σ^* - set of all strings over an alphabet Σ

\cdot - concatenation of strings

ϵ - empty string

This is a example of non-commutative monoid. Concatenation of strings is associative but not commutative as we can see in the following example.

$$abc \cdot def = abcdef \neq defabc = def \cdot abc \quad (1)$$

2.2 Homomorphism

A homomorphism is a structure-preserving map between two algebraic structures. In this section we will look at homomorphisms between monoids. Let $(S, *, e)$ and (T, \cdot, θ) be two monoids. A function $f : S \rightarrow T$ is a homomorphism if:

- **Preserves the operation**

For all $a, b \in S$, $f(a * b) = f(a) \cdot f(b)$.

- **Preserves the identity element**

$f(e) = \theta$.

Let's consider a simple example of homomorphism between two monoids. Let $S = (\mathbb{N}, +, 0)$ and $T = (\mathbb{N}, \cdot, 1)$. Let $f : S \rightarrow T$ be defined as $f(x) = 2^x$. We can see that:

- $f(a + b) = 2^{a+b} = 2^a \cdot 2^b = f(a) \cdot f(b)$
- $f(0) = 2^0 = 1$

2.3 Endomorphism

A special case of homomorphism is an endomorphism. This is a homomorphism from a monoid to itself. Lets consider a simple example of endomorphism. Let $S = (\mathbb{N}, +, 0)$ and $f : S \rightarrow S$ be defined as $f(x) = x + 1$. We can see that:

- $f(a + b) = a + b + 1 = (a + 1) + (b + 1) = f(a) + f(b)$
- $f(0) = 0 + 1 = 1$

Interesting property of endomorphisms is that a Set of all endomorphisms of a monoid is also a monoid. Let's consider a set of all endomorphisms of a monoid S . Let $S = (\mathbb{N}, +, 0)$ and let E be a set of all endomorphisms of S . We can define a binary operation on E as follows:

$$f * g = h \text{ where } h(x) = f(g(x)) \quad (2)$$

We can see that this operation is associative and has an identity element $e(x) = x$.

2.4 Application of Monoids to Segment Trees

How does this all relate to segment trees? First of all we can see that segment trees requires the properties of monoids to work. The associativity of the operation is required to be able to combine the results of the operations on the segments. The identity element is required when we ask for the result of the operation on an empty segment or when we want extend our initial array to a power of 2. Since all segment trees must preserve monoid properties, we can think more generic approach to the implementation. We will try to create a generic segment tree that can be used for any monoid for range queries and point updates. Let $*$ be a binary operation on a monoid S and let e be the identity element of the monoid.

Algorithm 7 Build Segment Tree over a Monoid $(S, *, e)$

```

1: procedure BUILDTREE( $A, seg, e$ )
2:    $n \leftarrow$  next power of two greater than or equal to  $|A|$ 
3:   for  $i = 0$  to  $|A| - 1$  do
4:      $seg[n + i] \leftarrow A[i]$  ▷ Insert original values
5:   end for
6:   for  $i = |A|$  to  $n - 1$  do
7:      $seg[n + i] \leftarrow e$  ▷ Pad remaining leaves with identity
8:   end for
9:   for  $i = n - 1$  downto  $1$  do
10:     $seg[i] \leftarrow seg[2 \times i] * seg[2 \times i + 1]$ 
11:   end for
12: end procedure

```

Algorithm 8 Generic Point Update in a Segment Tree over a Monoid $(S, *, e)$

```
1: procedure UPDATE( $arr, index, value$ )
2:    $index \leftarrow index + n$  ▷ Move to the leaf node
3:    $arr[index] \leftarrow value$  ▷ Set the new value at the leaf
4:   while  $index > 1$  do
5:      $index \leftarrow \lfloor index/2 \rfloor$  ▷ Move to parent
6:      $arr[index] \leftarrow arr[2 \times index] * arr[2 \times index + 1]$ 
7:   end while
8: end procedure
```

Algorithm 9 Range Query on Segment Tree over Monoid $(S, *, e)$

```
1: procedure RANGEQUERY( $seg, index, l, r, a, b, e$ ) ▷  $index$ : current node
   in segment tree ▷  $[l, r]$ : range covered by node ▷  $[a, b]$ : query range
2:   if  $b < l$  or  $a > r$  then
3:     return  $e$  ▷ No overlap
4:   else if  $a \leq l$  and  $r \leq b$  then
5:     return  $seg[index]$  ▷ Total overlap
6:   else
7:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
8:      $left \leftarrow$  RANGEQUERY( $seg, 2 \cdot index, l, mid, a, b, e$ )
9:      $right \leftarrow$  RANGEQUERY( $seg, 2 \cdot index + 1, mid + 1, r, a, b, e$ )
10:    return  $left * right$ 
11:   end if
12: end procedure
```

Now we can replace this generic monoid with some specific one. For example we can use a segment tree for sum of elements in given range. In this case we will use $*$ as addition and e as 0. Another example can be a segment tree for maximum value in given range. In this case we will use $*$ as maximum and e as $-\infty$. We can see that using this generic approach we can create a segment tree for any monoid.

2.5 Application of Endomorphisms to Segment Trees

In this section we will try to implement a generic segment tree for range update and range query. In order to do this we will need to use endomorphisms and their properties. Let's consider what happens when we update a range of elements in the segment tree. What we actually do is application of endomorphism on some range of elements. This operation doesn't break the structure of the segment tree, because endomorphism is a function that doesn't change the structure of the monoid. Making multiple updates (that is applying multiple endomorphisms) on the same range of elements also doesn't break the structure of the segment tree. This statement follows from the fact which we showed earlier that the set of all endomorphisms of a monoid is also a monoid. Multiple updates are just composition of endomorphisms. Let's consider a simple example:

We have a segment tree that can add some value to a range of elements and query the sum of elements in a range. Let's say we updated some range of elements by adding 1 to them, we can look at this as applying an endomorphism $f_1(x) = x + 1$ to the range of elements. We also updated the same range of elements by adding 5 to them, (again endomorphism $f_5(x) = x + 5$). In result we apply composition of both endomorphisms to the range of elements. The result of this operation is the same as applying endomorphism $f_6(x) = x + 6$ to the range of elements. We can see that composition of endomorphisms is also an endomorphism (endomorphisms are monoid). Now we will try to implement a generic segment tree for range update and range query. Let $*$ be a binary operation on a monoid S and let e be the identity element of the monoid. Let $e(x) = x$ be a identity element for monoid of endomorphisms

Algorithm 10 Build Segment Tree for Range Update and Range Query over Monoid $(S, *, e)$

```

1: procedure BUILDTREE( $A, seg, lazy, n, e$ )
2:   for  $i = 0$  to  $2n - 1$  do
3:      $seg[i] \leftarrow e$  ▷ Initialize segment tree with identity element
4:      $lazy[i] \leftarrow e$  ▷ Initialize lazy array with identity element
5:   end for
6:   for  $i = 0$  to  $n - 1$  do
7:      $seg[n + i] \leftarrow A[i]$  ▷ Copy array values to leaves of the segment tree
8:   end for
9:   for  $i = n - 1$  downto  $1$  do ▷ Build the tree from bottom to top
10:     $seg[i] \leftarrow *(seg[2 \times i], seg[2 \times i + 1])$  ▷ Combine child nodes
11:   end for
12: end procedure

```

Algorithm 11 Range Update on Segment Tree over Monoid $(S, *, e)$

```

1: procedure RANGEUPDATE( $seg, lazy, index, l, r, a, b, value, e$ )
2:   Call: PUSHDOWN( $index, l, r, lazy, e$ )
3:   if  $b < l$  or  $a > r$  then
4:     return ▷ No overlap, do nothing
5:   else if  $a \leq l$  and  $r \leq b$  then
6:      $seg[index] \leftarrow *(seg[index], (r - l + 1) \times value)$ 
7:      $lazy[index] \leftarrow *(lazy[index], value)$ 
8:   else
9:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
10:    RANGEUPDATE( $seg, lazy, 2 \times index, l, mid, a, b, value, e$ )
11:    RANGEUPDATE( $seg, lazy, 2 \times index + 1, mid + 1, r, a, b, value, e$ )
12:     $seg[index] \leftarrow *(seg[2 \times index], seg[2 \times index + 1])$ 
13:   end if
14: end procedure
15: procedure PUSHDOWN( $index, l, r, lazy, e$ )
16:   if  $lazy[index] \neq e$  then
17:      $seg[index] \leftarrow *(seg[index], lazy[index])$ 
18:     if  $l \neq r$  then
19:        $lazy[2 \times index] \leftarrow *(lazy[2 \times index], lazy[index])$ 
20:        $lazy[2 \times index + 1] \leftarrow *(lazy[2 \times index + 1], lazy[index])$ 
21:     end if
22:      $lazy[index] \leftarrow e$  ▷ Clear lazy value after pushing it down
23:   end if
24: end procedure

```

Algorithm 12 Range Query on Segment Tree over Monoid $(S, *, e)$

```
1: procedure RANGEQUERY(seg, lazy, index, l, r, a, b, e)
2:   Call: PUSHDOWN(index, l, r, lazy, e)
3:   if  $b < l$  or  $a > r$  then
4:     return e ▷ No overlap, return identity element
5:   else if  $a \leq l$  and  $r \leq b$  then
6:     return seg[index] ▷ Total overlap, return the current node's value
7:   else
8:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
9:     left  $\leftarrow$  RANGEQUERY(seg, lazy,  $2 \times index$ , l, mid, a, b, e)
10:    right  $\leftarrow$  RANGEQUERY(seg, lazy,  $2 \times index + 1$ , mid + 1, r, a, b, e)
11:    return  $*(left, right)$  ▷ Combine results from left and right child
12:   using *
13:   end if
14: end procedure
15: procedure PUSHDOWN(index, l, r, lazy, e)
16:   if lazy[index]  $\neq e$  then
17:     seg[index]  $\leftarrow$   $*(seg[index], lazy[index])$ 
18:     if  $l \neq r$  then
19:       lazy[ $2 \times index$ ]  $\leftarrow$   $*(lazy[2 \times index], lazy[index])$ 
20:       lazy[ $2 \times index + 1$ ]  $\leftarrow$   $*(lazy[2 \times index + 1], lazy[index])$ 
21:     end if
22:     lazy[index]  $\leftarrow e$  ▷ Clear lazy value after pushing it down
23:   end if
24: end procedure
```

Using this generic approach we can simply replace the monoid with some specific one and set endomorphism as some function that we want to apply to the range of elements.

3 Binsearch on Segment Tree

In this section we will see that we can use segment trees not only for queries and updates but also for binary search. Let's consider the following problem:

Given an array $a[1 \dots n]$, answer a query $Q(i, j, x)$: What is the smallest index k so that $a[k] \geq x$ in subarray $a[i \dots j]$?

We want to be able to answer this query in $O(\log n)$ time. We can create a segment tree with max value in given range but this is not enough. We need to upgrade our segment tree with binsearch operation, which goes as follows:

- Traverse recursively the tree to find the nodes that are in the range of the query (just like in range query). Always go to the left child first.
- For every nodes perform a binary search on the subtree.
- Binsearch
 - if right child's value is less than x , return -1 (no such index)
 - if left child's value is greater than x , go to left child
 - else go to right child
- Because we traverse the tree from left to right, we can be sure that the first binsearch that returns index different than -1 is the answer to the query.
- if no such index was found, return -1.

Algorithm 13 Find First Index With Value Greater Than x in Range $[a, b]$

```

1: procedure QUERY(segTree, index, l, r, a, b, x)      ▷ index: current node
   index ▷ [l, r]: segment represented by current node  ▷ [a, b]: query range
2:   if  $b < l$  or  $a > r$  then
3:     return -1                                          ▷ No overlap
4:   else if  $a \leq l$  and  $r \leq b$  then
5:     return BINARYSEARCH(segTree, index, x)
6:   else
7:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
8:      $left \leftarrow$  QUERY(segTree,  $2 \cdot index$ ,  $l$ ,  $mid$ ,  $a$ ,  $b$ ,  $x$ )
9:     if  $left \neq -1$  then
10:      return  $left$ 
11:     end if
12:     return QUERY(segTree,  $2 \cdot index + 1$ ,  $mid + 1$ ,  $r$ ,  $a$ ,  $b$ ,  $x$ )
13:   end if
14: end procedure
15:
16: procedure BINARYSEARCH(tree, index, x)
17:   while  $index * 2 < tree.size$  do
18:      $left \leftarrow tree[index * 2]$ 
19:     if  $left > x$  then
20:        $index \leftarrow index * 2$ 
21:     else
22:        $index \leftarrow index * 2 + 1$ 
23:     end if
24:   end while
25:   if  $tree[index] > x$  then
26:     return  $index$ 
27:   else
28:     return -1                                          ▷ No index found
29:   end if
30: end procedure

```

Let's consider the following example:
 We have an array $A = [5, 8, 2, 7, 1, 11, 13, 12, 19, 14, 15, 0, 15, 10, 15, 4]$ and we want to find the smallest index k such that $A[k] \geq 14$ in the range $A[5 : 12]$.

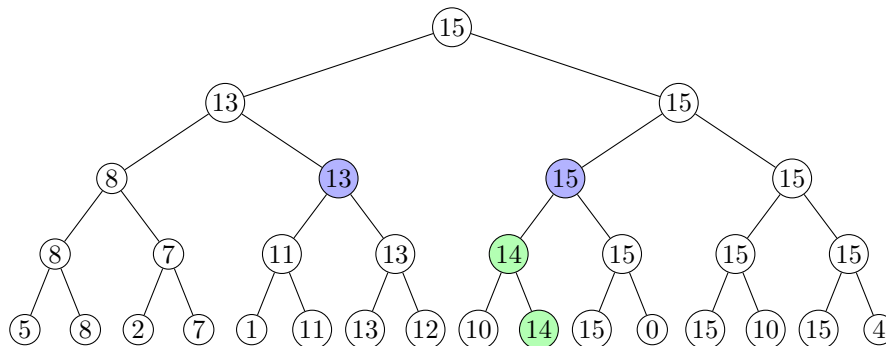


Figure 6: The blue nodes are the ones that we performed binsearch on. The green nodes are the ones that binsearch traversed. The result of the query is index 10. As we can see, the first binsearch (on node with value 13 returned -1)

4 Max prefix sum on Segment Tree

In this section we will take look at a little bit more complex query which will require a little bit more complex segment tree. Let's consider the following problem:

Given an array $a[1 \dots n]$, we want to be able to efficiently answer a query $Q(i, j)$: What is the maximum prefix sum in subarray $a[i \dots j]$? While still being able to perform point updates on array.

What structure we can use to solve this problem? ...You guessed it, segment tree! This time nodes of our segment tree will store more information than just a single value. We will store a pair of values in each node:

$$(prefix, sum) \quad (3)$$

Where *prefix* is the maximum prefix sum in the subtree of this node and *sum* is the sum of all elements in the subtree of this node. Before we construct the segment tree we need to define merge operation, that will be used to calculate the values in the parent node from the values in the child nodes.

Algorithm 14 Merge Two Segments in Segment Tree

```
1: procedure MERGE(seg, index)
2:    $seg[index].sum \leftarrow seg[2 * index].sum + seg[2 * index + 1].sum$ 
3:    $seg[index].prefix \leftarrow \max(seg[2 * index].prefix, seg[2 * index].sum + seg[2 * index + 1].prefix)$ 
4: end procedure
```



Figure 7: In this example the red blocks represent the subarray in left child and the blue blocks represent the subarray in right child. Max prefix sum is either the max prefix sum in left child or the sum of left child plus max prefix sum in right child. In this example the max prefix (equal to 7) of parent node is sum of left child plus max prefix of right child as it is greater than max prefix of left child (equal to 6).

Now we can construct the segment tree and perform point updates (left as exercise) exactly like in previous examples only this time we use the merge operation defined above. Keeping in mind that when creating or updating the leaf node we need to set the prefix to the value of the element in the array.

Query operation on the other hand needs a little more modification. We traverse the tree and find the nodes in the range just like in range query. The result of the query is max prefix of some node plus sum of the nodes to the left we found during the traversal. In particular the result might be max prefix of leftmost node in range or sum of all nodes we found.

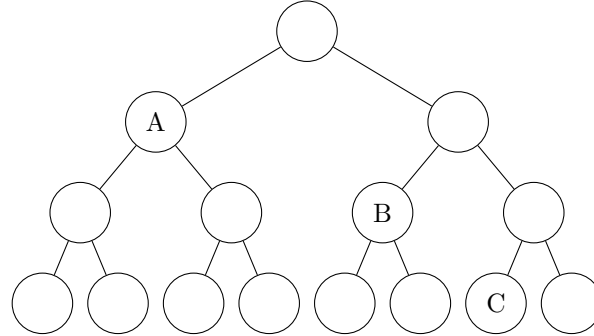


Figure 8: Let A, B, C be the nodes in the range of query. The result of this query would be $\max(prefix(A), sum(A) + prefix(B), sum(A) + sum(B) + prefix(C))$.

Our range query, very similar to standard one, needs new variable sum , and a little modification when it finds a node in range:

Algorithm 15 Range Max Prefix Query

```
1: procedure RANGEQUERY(seg, index, l, r, a, b, sum)    ▷ sum: sum of all
   nodes in range to the left of current node
2:   if  $b < l$  or  $a > r$  then
3:     return 0
4:   else if  $a \leq l$  and  $r \leq b$  then
5:      $tmp\_sum \leftarrow sum$ 
6:      $sum += seg[index].sum$ 
7:     return  $seg[index].prefix + tmp\_sum$ 
8:   else
9:      $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
10:     $left \leftarrow$  RANGEQUERY(seg,  $2 \cdot index$ ,  $l$ ,  $mid$ ,  $a$ ,  $b$ ,  $sum$ )
11:     $right \leftarrow$  RANGEQUERY(seg,  $2 \cdot index + 1$ ,  $mid + 1$ ,  $r$ ,  $a$ ,  $b$ ,  $sum$ )
12:    return  $\max(left, right)$ 
13:   end if
14: end procedure
```

5 Sweeping Line

In this section we will use segment trees to implement a sweeping line algorithm. Consider the following problem: We are given a set of vertical and horizontal lines in a plane and we want to find the number of intersections between these lines. The format of the input is as follows:

- n - number of lines
- $0 \leq x, y \leq 10^6$ - coordinates of the points that represent the lines.
- We can assume that the length of the line is greater than 0, we do not consider intersection at the endpoints of the lines and 2 parallel can't intersect.
- We are given number n and n tuples (x_1, y_1, x_2, y_2) where (x_1, y_1) and (x_2, y_2) are the coordinates of the endpoints of the line.

As we can see the problem is not online (we don't have to answer any queries) and yet the easiest way to solve it is to use a segment tree. We perform something called a sweeping line algorithm. We will create from a segment tree a kind of brush that will sweep through the plane and count the intersections. Lets say we swipe from left to right (we can do this from right to left or top to bottom as well). If our brush (we can also look at this as infinite vertical straight line) is at position x we know that it stores all the lines that intersect this vertical line. Knowing that we can ask how many lines intersect this vertical line at some range (y_1, y_2) , in another words for every vertical line at x we know how many lines it intersects. How do we know that? First of all we need to sort our input lines by their lower x coordinate prioritizing the vertical lines. Now we iterate over x axis if we encounter the horizontal line we add 1 to the segment

tree at position y (the y coordinate of the horizontal line). If we encounter the end of the horizontal line we remove 1 from the segment tree at position y . You can see that the segment tree is used to store the number of horizontal lines and the leaves of the tree represent the y coordinates of the horizontal lines. Now if we encounter a vertical line we can ask the segment tree how many horizontal lines intersect this vertical line at the range $(y1, y2)$. We add this number to the final result and continue iterating over the x axis.

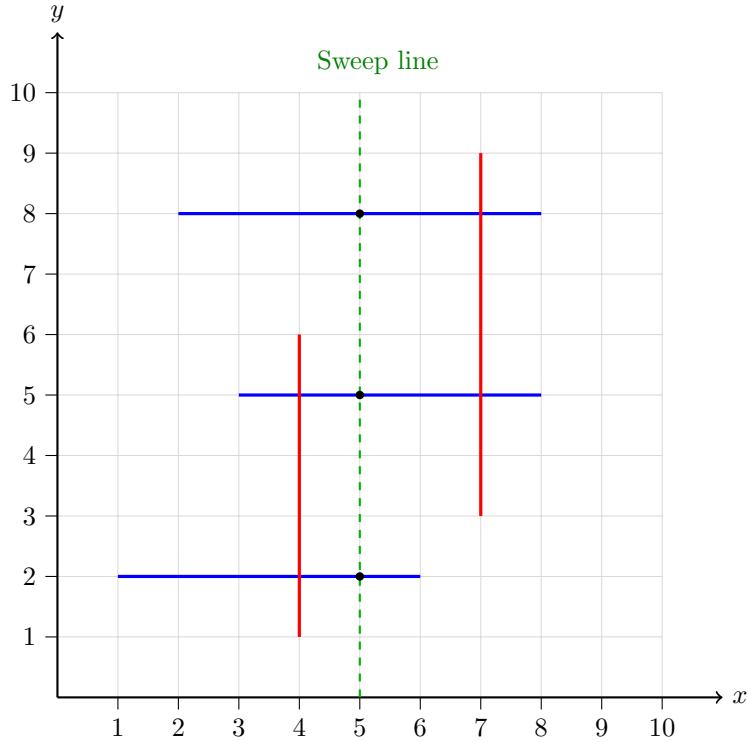


Figure 9: This picture illustrates algorithm in proces. The result so far is 2, intersections of the red line at $x = 4$. Current state of segment trres leaves is $[0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]$.

How does the operations on segment tree looks like then? This is acctually very simple segment tree with point update (add 1) and range query (sum on given range), which i described in the first section. The build operation is even simpler as we start with 0 in all leaves.

For easier implementation we store every line as node:

Structure Line

- $x1$: integer
- $y1$: integer

- `x2` : integer
- `y2` : integer
- `vertical` : boolean

The whole algorithm can be implemented as follows:

- sort lines by `x1` coordinate, if two lines have the same `x1` choose the vertical line first
- initialize segment tree with size of 10^6
- iterate over lines:
 - if line is horizontal, add 1 to segment tree at position `y1`
 - if line is vertical, query segment tree for range `(y1, y2)` and add result to final result
 - if line is horizontal and we reached the end of the line, remove 1 from segment tree at position `y2`
- return final result

Link to problem: <https://cses.fi/problemset/task/1740>