Lecture 9.
RSQ, RMQ.

Algorithms and Data Structures
Ivan Solomatin
MIPT

Outline

- Problem statement
- Static case
  - RSQ: prefix sum
  - RMQ: sparse table
- Dynamic case
  - RSQ/RMQ: sqrt-decomposition
  - RSQ: Fenwick tree
  - RSQ/RMQ: segment tree

Problem statement

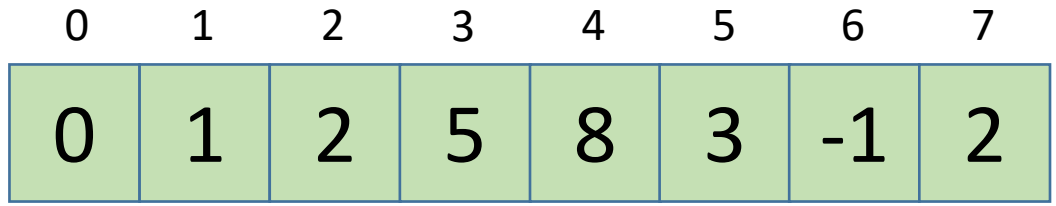
- **RSQ/RMQ**
- **Static/dynamic case**
- **Online/offline case**

Problem statement
RSQ/RMQ

- RSQ – range sum query
- RMQ – range min (max) query

$$RSQ(l,r) = \sum_{i=l}^{r-1} a[i], \qquad RMQ(l,r) = \min\{a[i] : i \in [l,r)\}$$

$$RSQ(l,r) = \sum a[l:r], \qquad RMQ(l,r) = \min(a[l:r])$$



$RSQ(1,5) = 16$ $RMQ(1,5) = 1$

$RSQ(3,7) = 15$ $RMQ(3,7) = -1$

$RSQ(4,5) = 8$ $RMQ(4,5) = 8$

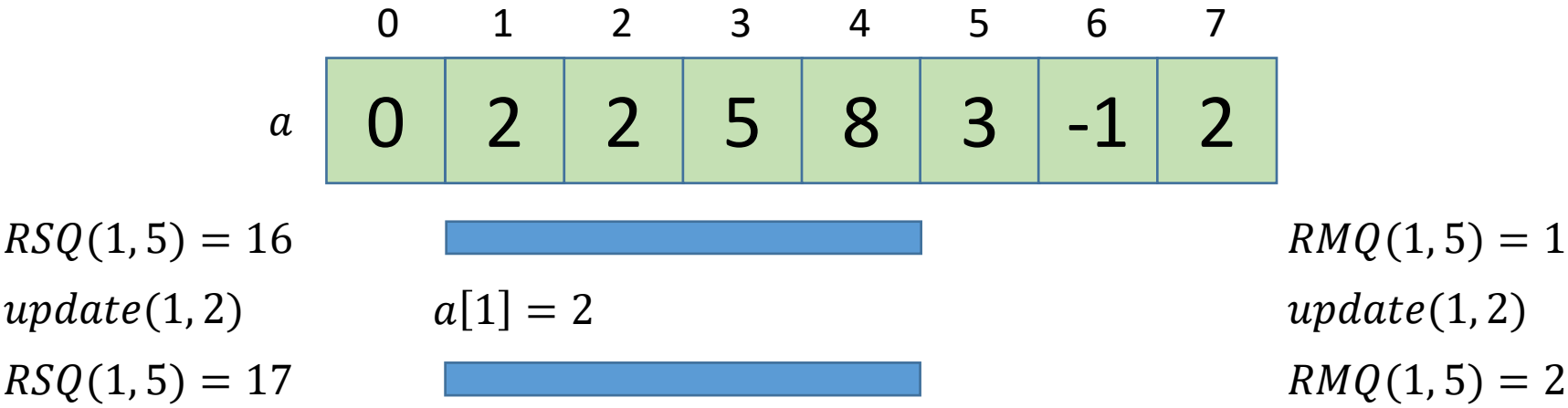
$RSQ(4,4) = 0$ $RMQ(4,4) = \infty$

Problem statement

Static/dynamic case

- Static case: a does not change
- Dynamic case: update queries are possible:

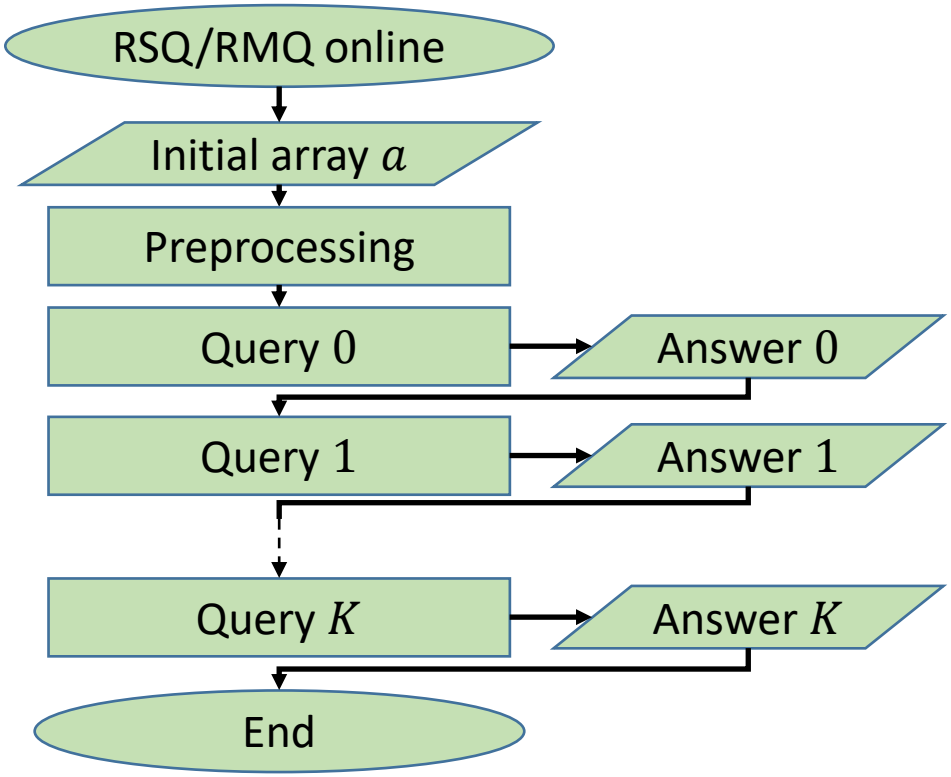
update(i, v): set $a[i]$ value to v: $a[i] = v$

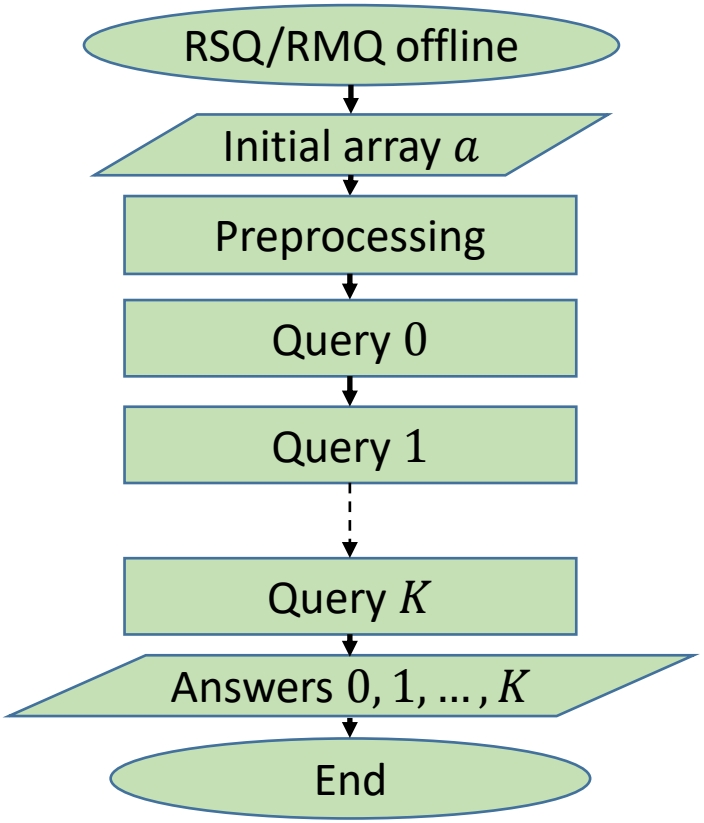


Problem statement

Online/offline case

- Online case: answers provided right after query
- Offline case: answers provided after obtaining all requests





Static case

- **RSQ: prefix sum**
- RMQ: sparse table

RSQ: prefix sum

Idea

Complexity:

- Preproc (build): $O(N)$
- Query: $O(1)$
- Update: $O(N)$

Let's pre-calculate array of prefix sums:

$$s[i] = \sum_{j=0}^{i-1} a[j] = \sum a[:i]$$

We can easily pre-calculate it during preprocessing in $O(N)$ operations:

```
s = []
for v in a:
    s.append(s[-1] + v)
```

And finally:

$$RSQ(l,r) = \sum_{i=l}^{r-1} a[i] = \sum_{i=0}^{r-1} a[i] - \sum_{i=0}^{l-1} a[i] = s[r] - s[l]$$

RSQ can be calculated in $O(1)$ operations.

But if array is updated, we need to recalculate s: $O(N)$.

This approach cannot be applied for RMQ, because minimum has no inverse function: having $\min(A \cup B)$ and $\min(A)$ we can't calculate $\min(B)$.

RSQ: prefix sum

Implementation

```
class RSQPrefixSum:
    def __init__(self, a):
        self.s = [0]
        for v in a:
            self.s.append(self.s[-1] + v)

    def rsq(self, l, r):
        return self.s[r] - self.s[l]
```

Complexity:

- Preproc (build): $O(N)$
- Query: $O(1)$
- Update: $O(N)$

Static case

- RSQ: prefix sum
- **RMQ: sparse table**

RMQ: sparse table

Idea

Having $\min(A \cup B)$ and $\min(A)$ we can't calculate $\min(B)$.

But having $\min(A)$ and $\min(B)$ we can calculate $\min(A \cup B)$:

$$\min(A \cup B) = \min(\min(A), \min(B))$$

So, we can pre-calculate minimums for some "basis" ranges and construct our request range $[l, r)$ of these basis ranges.

This leads us to idea of sparse table: Let's pre-calculate answers for the following ranges:

$$ST[k][i] = RMQ\left(i, i + 2^k\right) = \min\left(a[i : i + 2^k]\right)$$

Let's notice that range $(i, i + 2^k)$ consists exactly of 2 ranges:

$$[i, i + 2^k) = [i, i + 2^{k-1}) \cup [i + 2^{k-1}, i + 2^{k-1} + 2^{k-1}].$$

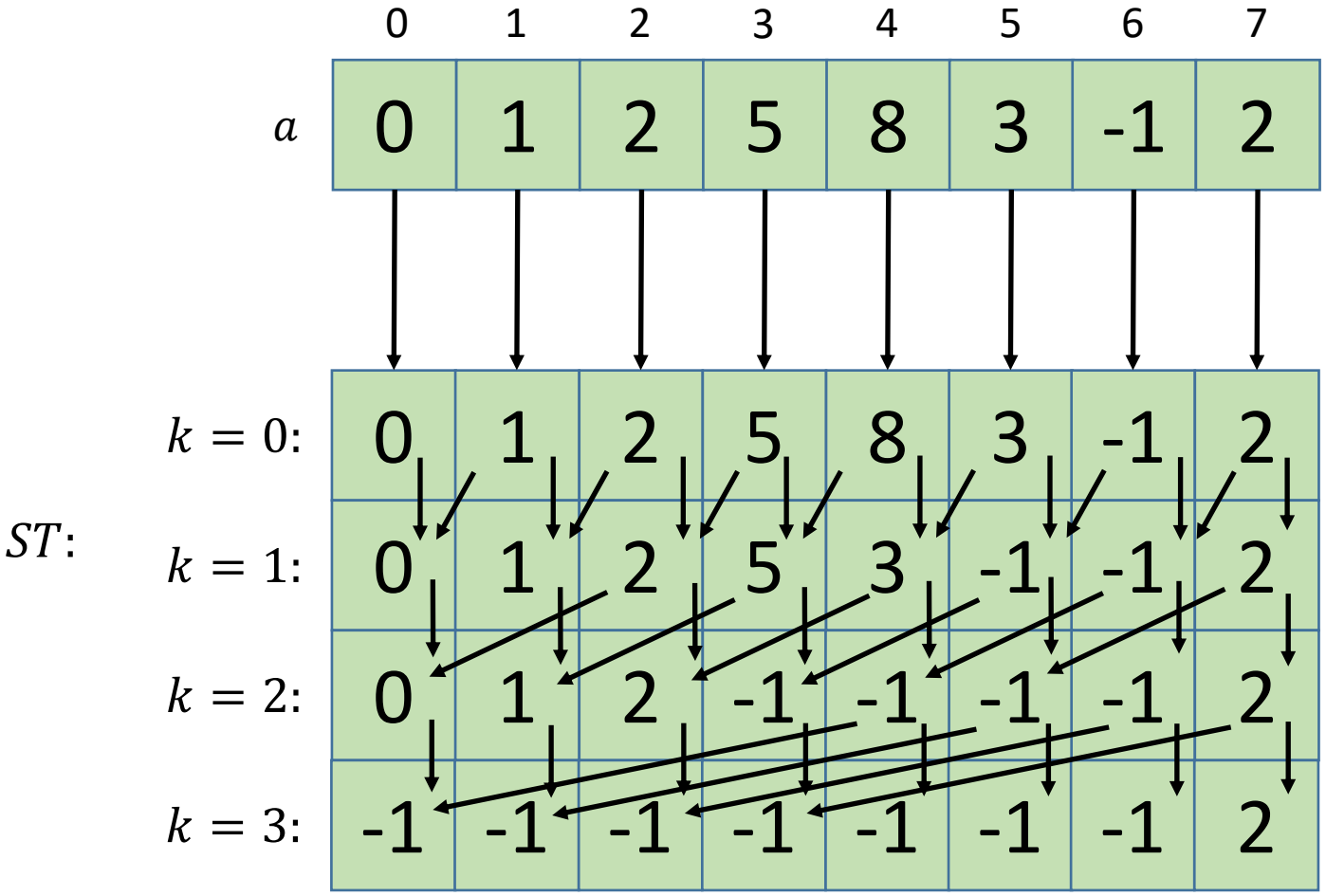
So: $ST[k][i] = \min(ST[k-1][i], ST[k-1][i + 2^{k-1}]$

Maximum k which makes sense to store is $k = \lceil \log_2 N \rceil$, because it will contain whole array.

So, we can pre-calculate ST in $O(N \log N)$ operations.

RMQ: sparse table

Idea

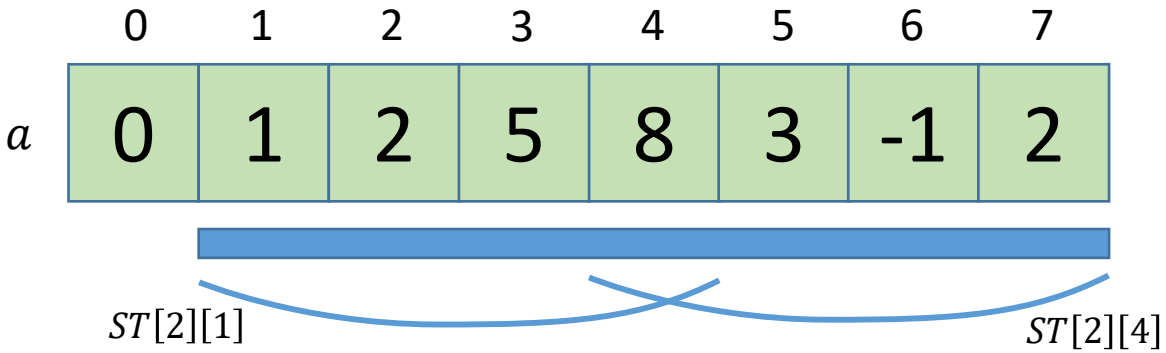


RMQ: sparse table

Idea

For each index we know minimum of the following 2^k items, for each k.

To calculate minimum on each possible range $[l, r)$, let's notice that each range can be represented as union of two pre-calculated ranges:



$$k = \lfloor \log_2(r - l) \rfloor$$

$$[l, r) = [l, l + 2^k) \cup [r - 2^k, r)$$

So, $RMQ(l, r) = \min\left(ST[k][l], ST[k][r - 2^k]\right)$.

RMQ: sparse table

Implementation

```
from math import log2, ceil, floor

class RMQSparseTable:
    def __init__(self, a):
        N = len(a)
        K = ceil(log2(N)) + 1
        self.ST = [[None] * N for k in range(K)]
        self.ST[0] = list(a)
        for k in range(1, K):
            for i in range(N):
                if i + 2 ** (k - 1) < N:
                    self.ST[k][i] = min(self.ST[k - 1][i],
                                        self.ST[k - 1][i + 2 ** (k - 1)])
                else:
                    self.ST[k][i] = self.ST[k - 1][i]

    def rmq(self, l, r):
        if l == r:
            return inf
        k = floor(log2(r - l))
        return min(self.ST[k][l],
                   self.ST[k][r - 2 ** k])
```

Complexity:

Preproc: $O(N \log N)$

Query: $O(1)$

Update: $O(N \log N)$

Dynamic case

- **RSQ/RMQ: sqrt-decomposition**
- RSQ: Fenwick tree
- RSQ/RMQ: segment tree

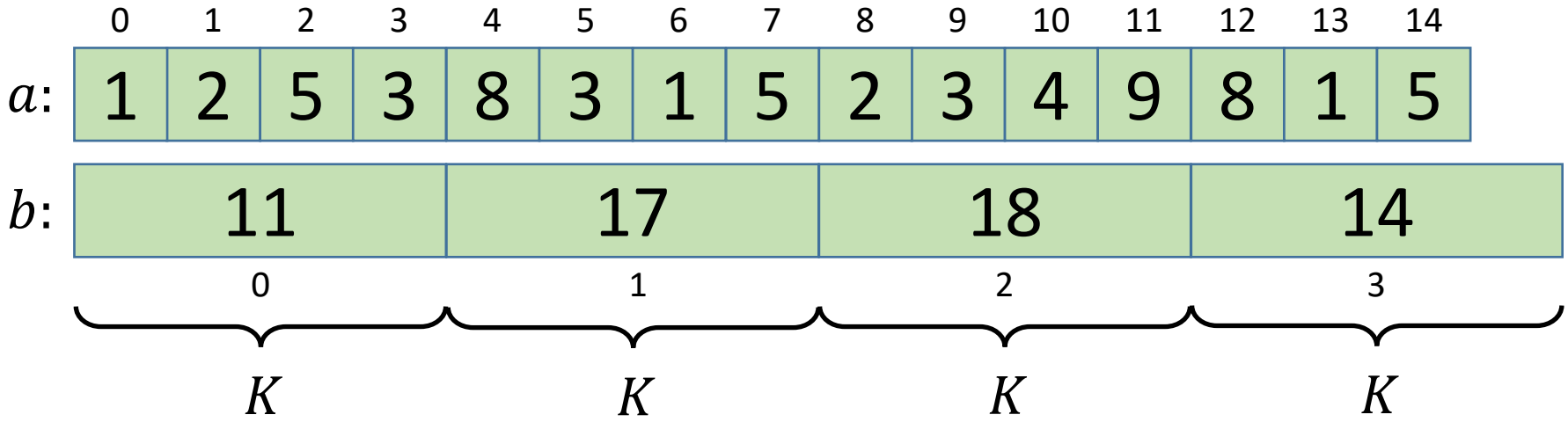
RSQ/RMQ: sqrt-decomposition

Idea

Let's continue idea of pre-calculating values for "basis" ranges and construct desired range using these "basis" ranges.

Prefix sum and sparse table are not suitable for dynamic case, because each value may influence lots of pre-calculated values. So, we need to re-initialize data structure if at least one item changes. Let's try to avoid that and localize possible changes.

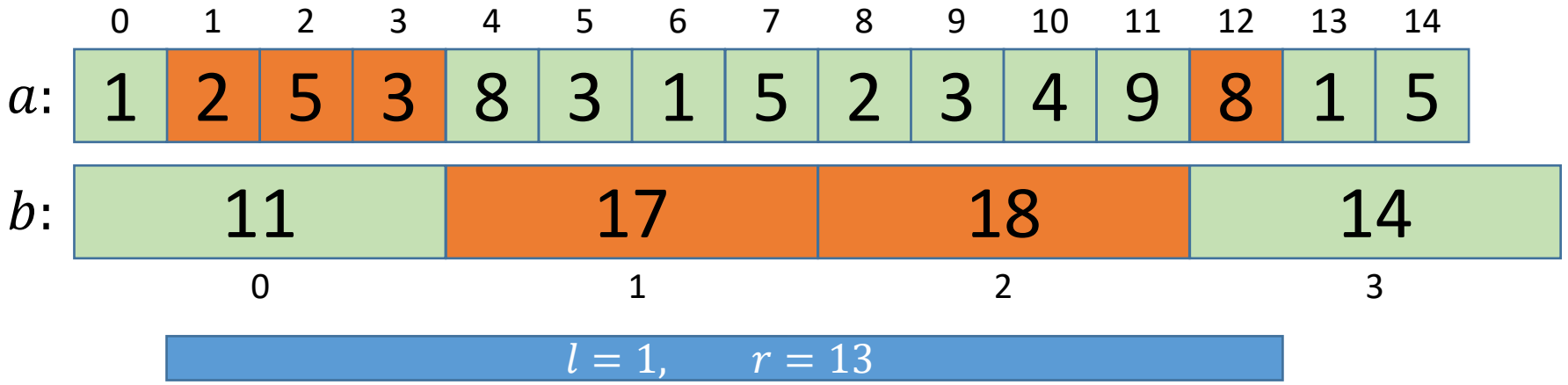
Let's divide array into independent blocks of K items. And pre-calculate queries (sums for RSQ and minimums for RMQ) inside these blocks.



RSQ/RMQ: sqrt-decomposition

Idea

We can use values in pre-calculated blocks which are completely inside desired range and manually calculated edges.



Query complexity:
K – manual edge calculation (left)
N/K – blocks calculation
K – manual edge calculation (right)

$$O\left(2K + \frac{N}{K}\right)$$

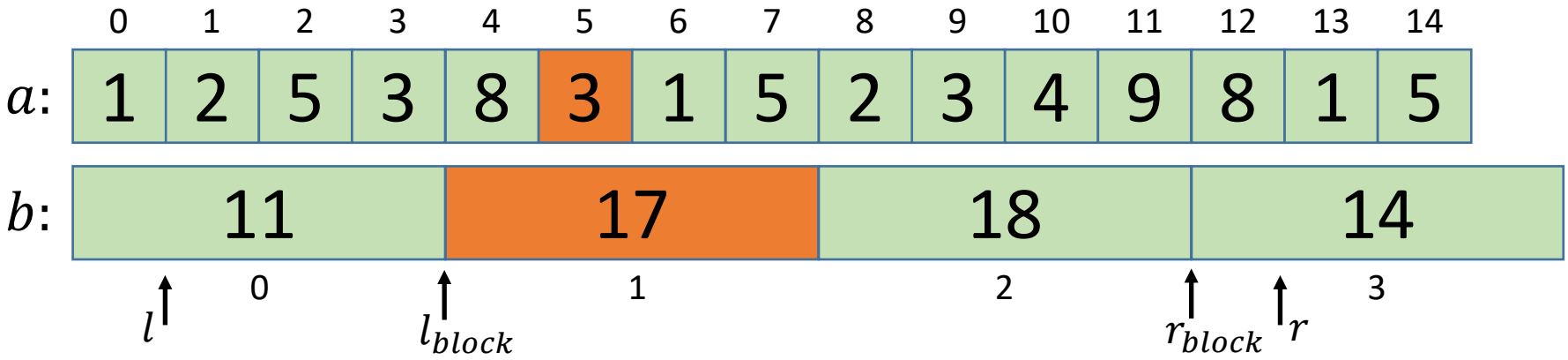
If $K = const$, it is $O(N)$.

Best complexity may be obtained for $K = \sqrt{N}$:

$$O\left(\sqrt{N} + \frac{N}{\sqrt{N}}\right) = O(\sqrt{N})$$

RSQ/RMQ: sqrt-decomposition

Idea



Block which contains item i: $\lfloor i/K \rfloor$

Left border of first complete block in query (l, r): $l_{block} = K\lceil l/K \rceil$

Right border of last complete block in query (l, r): $r_{block} = K\lfloor r/K \rfloor$

If we update a value, we should only update the block which contains this element which will take $O(\sqrt{N})$ operations.

For RSQ we can just add difference in $O(1)$.

RSQ/RMQ: sqrt-decomposition

Implementation

```python
from math import log2, floor, ceil
class RSQSqrtDecomposition:
    def __init__(self, a):
        N = len(a)
        self.k = int(ceil(log2(N)))
        self.a = a
        self.b = [0] * (self.k + 1)
        for i, v in enumerate(a):
            self.b[i // self.k] += v

    def rsq(self, l, r):
        l_block = self.k * int(ceil(l / self.k))
        r_block = self.k * int(floor(r / self.k))
        if l_block > r_block:
            return sum(self.a[l : r])
        return (sum(self.a[l : l_block]) +
                sum(self.b[l_block // self.k : r_block // self.k]) +
                sum(self.a[r_block : (self.a[l : ])))

    def update(self, i, v):
        delta = v - self.a[i]
        self.a[i] = v
        self.b[i // self.k] += delta
```

Complexity:

Preproc: $O(N)$

Query: $O(\sqrt{N})$

Update (RSQ): $O(1)$

Update (RMQ): $O(\sqrt{N})$

Dynamic case

- RSQ/RMQ: sqrt-decomposition
- **RSQ: Fenwick tree**
- RSQ/RMQ: segment tree

RSQ: Fenwick tree

Idea

Idea of Fenwick tree is rather difficult to understand, because it's not natural at all. But it's extremely easy to implement!

Fenwick tree uses idea of inverse function, so it's not suitable for RMQ, but suitable for RSQ (actually, exists modification which allows to solve RMQ using Fenwick tree, but it requires additional modifications).

Idea: let's use 1-d array f to store partial sums. Let $f[i]$ store sum in a range of length l which ends in i: $f[i] = sum(a[i - l + 1 : i + 1])$

The trick is in function $l(i)$: $l(i) = 2^{k(i)}$, where $k(i)$ is number of 1s in the end of the binary representation of i.

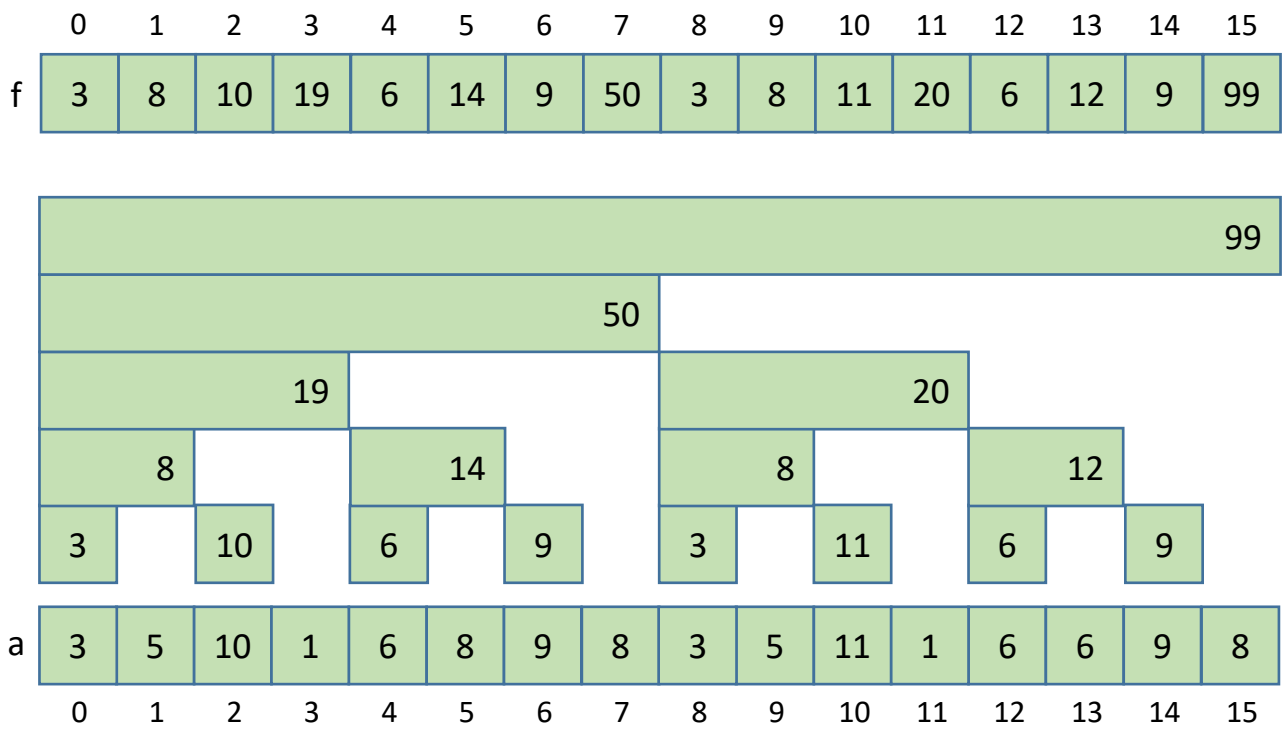
$$i = 4815 = 10010110\overbrace{01111}^{k(i)=4}_2$$

$$l(4815) = 2^4 = 16$$

RSQ: Fenwick tree

Idea

i	$bin(i)$	$k(i)$	$l(i)$
0	0000	0	1
1	0001	1	2
2	0010	0	1
3	0011	2	4
4	0100	0	1
5	0101	1	2
6	0110	0	1
7	0111	3	8
8	1000	0	1
9	1001	1	2
10	1010	0	1
11	1011	2	4
12	1100	0	1



RSQ: Fenwick tree

$l(i)$ calculation

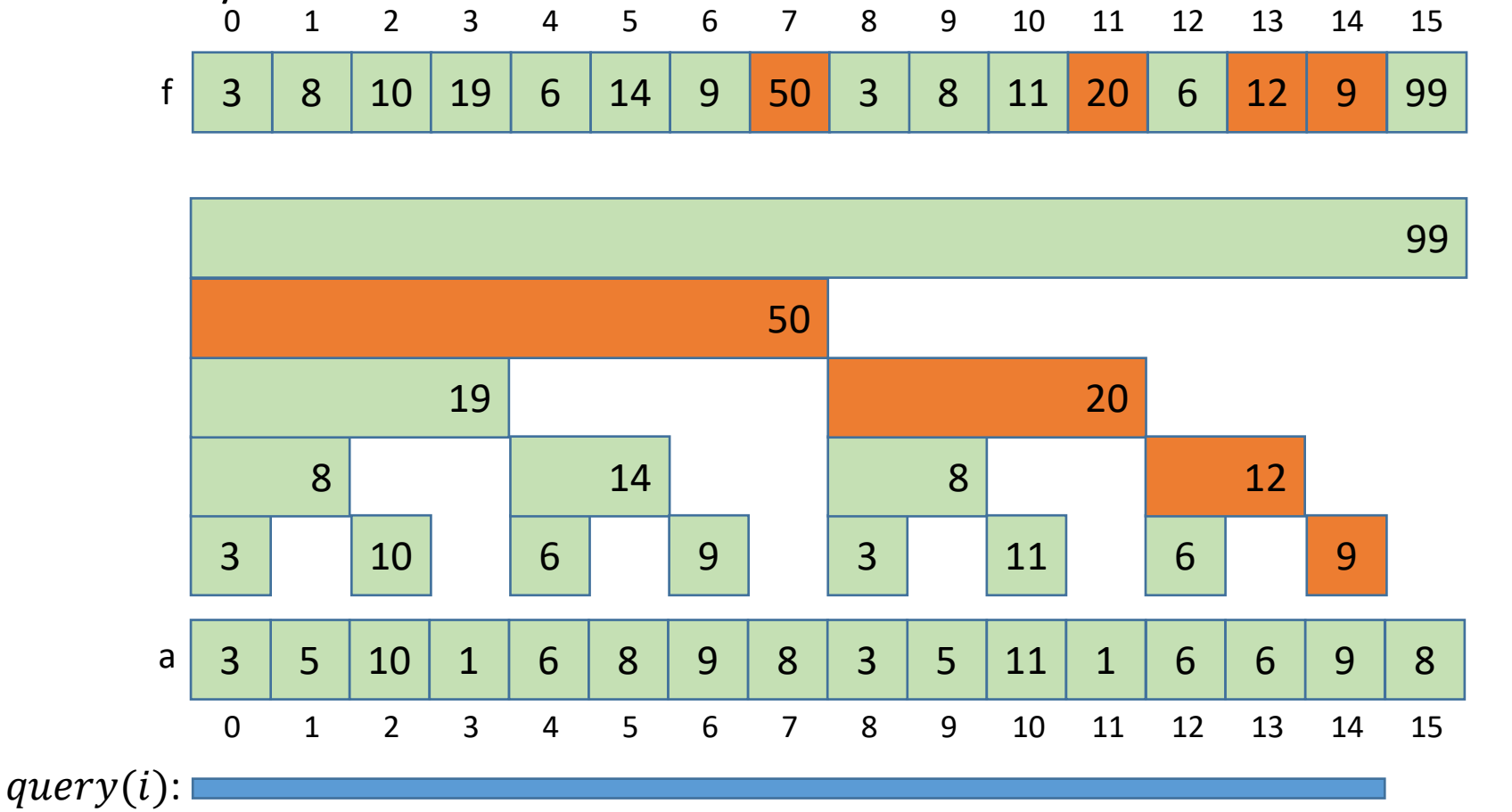
$$
\begin{aligned}
i &= \; ???????01111_2 \\
i+1 &= \; ???????10000_2 \\
\sim i &= \; \overline{?}\overline{?}\overline{?}\overline{?}\overline{?}\overline{?}\overline{?}\overline{?}10000_2 \\
\sim i \;\&\; (i+1) &= \; 00000000010000_2 = l(i)
\end{aligned}
$$

$$l(i) = \sim i \;\&\; (i+1)$$

RSQ: Fenwick tree

$query(i) = RSQ(0, i+1)$

Query



Previous block:

$i \rightarrow i - l(i)$

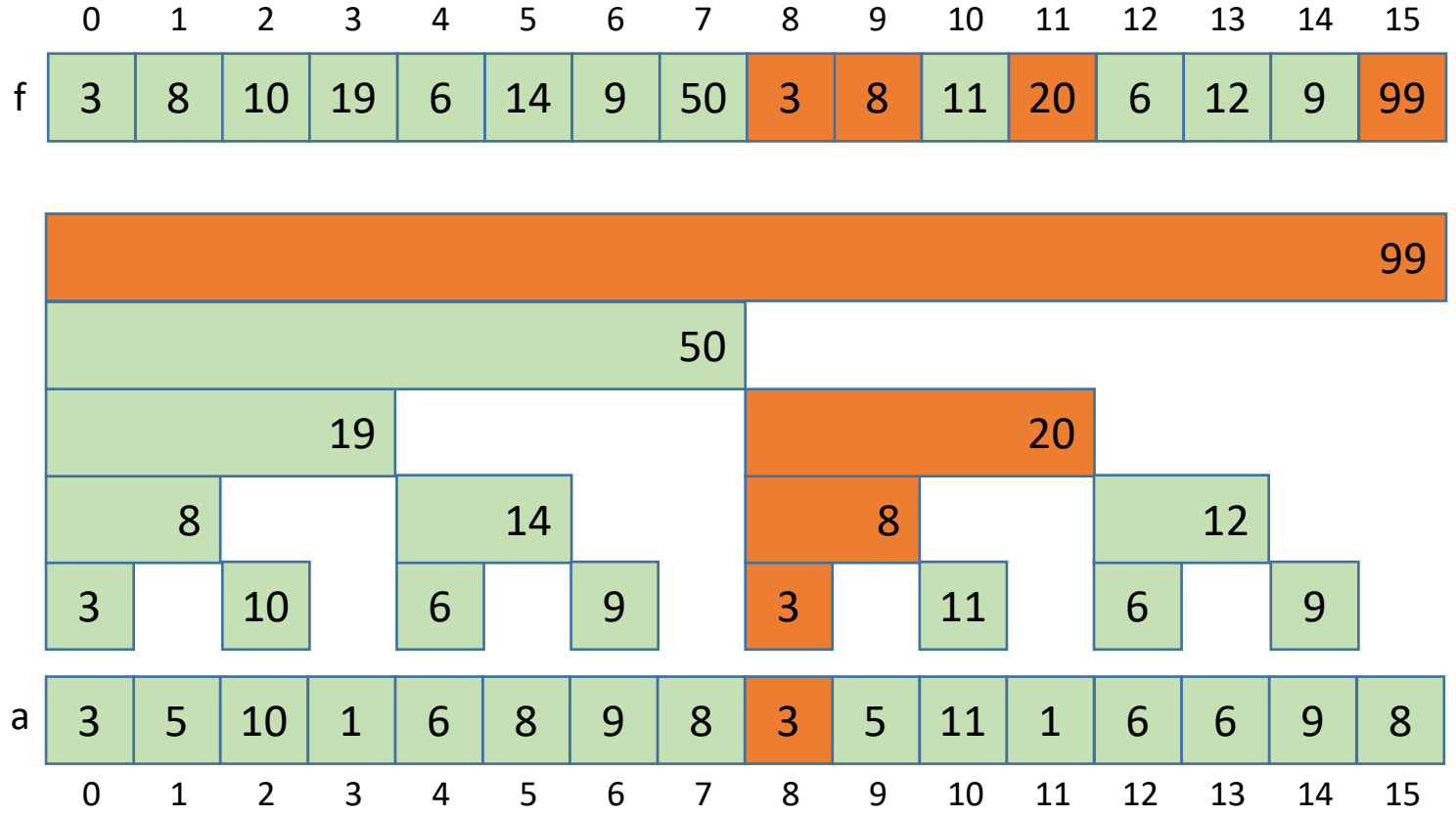
$$\begin{array}{r} ?????10001111_2 \\ - \quad 00000000010000_2 \\ \hline ?????\mathbf{0111}1111_2 \end{array}$$

Replaces right most group of 0s with 1s:

$O(\log N)$

RSQ: Fenwick tree

Update



$update(8, delta)$:

Next block:

$i \rightarrow i + l(i)$

$$\begin{array}{r} ????????01111_2 \\ + \quad 000000010000_2 \\ \hline ????????\mathbf{1}1111_2 \end{array}$$

Replaces right most 0 with 1:

$O(\log N)$

RSQ: Fenwick tree

Implementation

```
class RSQFenwick:
    def __init__(self, N):
        self.N = N
        self.f = [0] * self.N

    def query(self, i):
        res = 0
        while i >= 0:
            res += self.f[i]
            i -= ~i & (i + 1)
        return res

    def update(self, i, delta):
        while i < self.N:
            self.f[i] += delta
            i += ~i & (i + 1)

    def rsq(self, left, right):
        return self.query(right - 1) - self.query(left - 1)
```

Complexity:
Query: $O(\log N)$
Update: $O(\log N)$
Preproc (build): $O(N \log N)$

Dynamic case

- RSQ/RMQ: sqrt-decomposition
- RSQ: Fenwick tree
- **RSQ/RMQ: segment tree**

RSQ/RMQ: Segment tree

Idea

Let's store binary tree (as in heap) with items as leafs. Let each node contain sum (min) of all leafs in his subtree.

$$s_i = RSQ(l_i, r_i); \qquad [l_i, r_i) - \text{ responsibility range of vertex } i$$

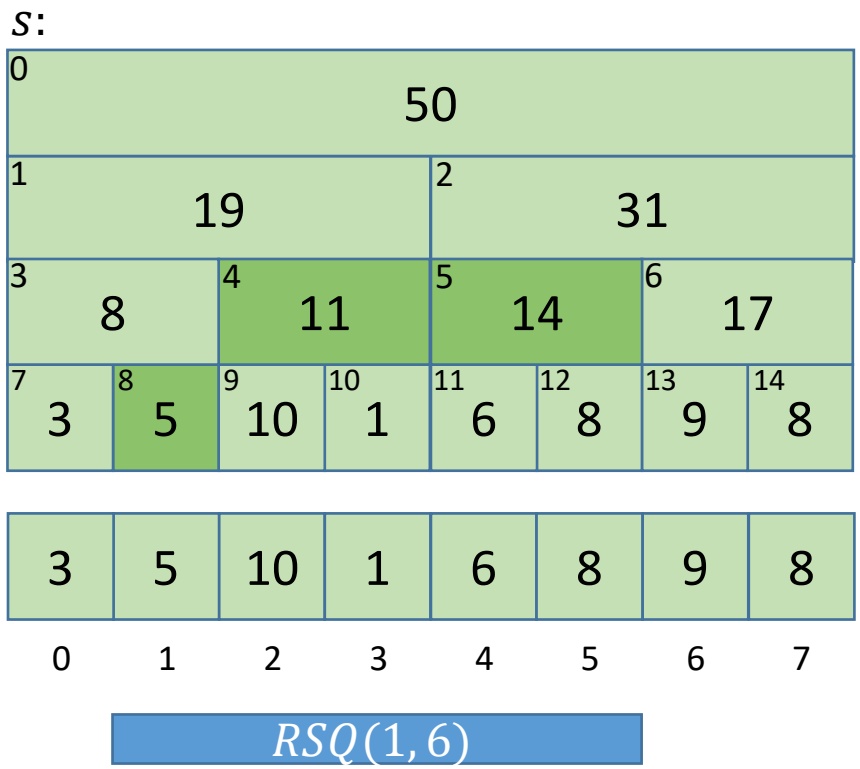
$$children(i) = \begin{bmatrix} 2i+1 \\ 2i+2 \end{bmatrix}$$

$$parent(i) = (i-1)//2$$

$RSQ_I(l, r, i)$ = RSQ on intersection: $[l, r) \cap [l_i, r_i)$,

$$RSQ_I(l, r, i) = \begin{cases} s_i, & if\ [l\_i, r\_i) \subseteq [l, r) \\ 0, & if [l\_i, r\_i) \cap [l, r) = \emptyset \\ RSQ_I(l, r, 2i+1) + RSQ_I(l, r, 2i+2), & otherwise \end{cases}$$

$$RSQ(l, r) = RSQ_I(l, r, 0)$$



RSQ/RMQ: Segment tree

Update

`update(i, val):`

When $a[i]$ is updated, we need to update values in nodes which contain i in their responsibility range:

$$\{v: i \in [l_v, r_v)\}$$

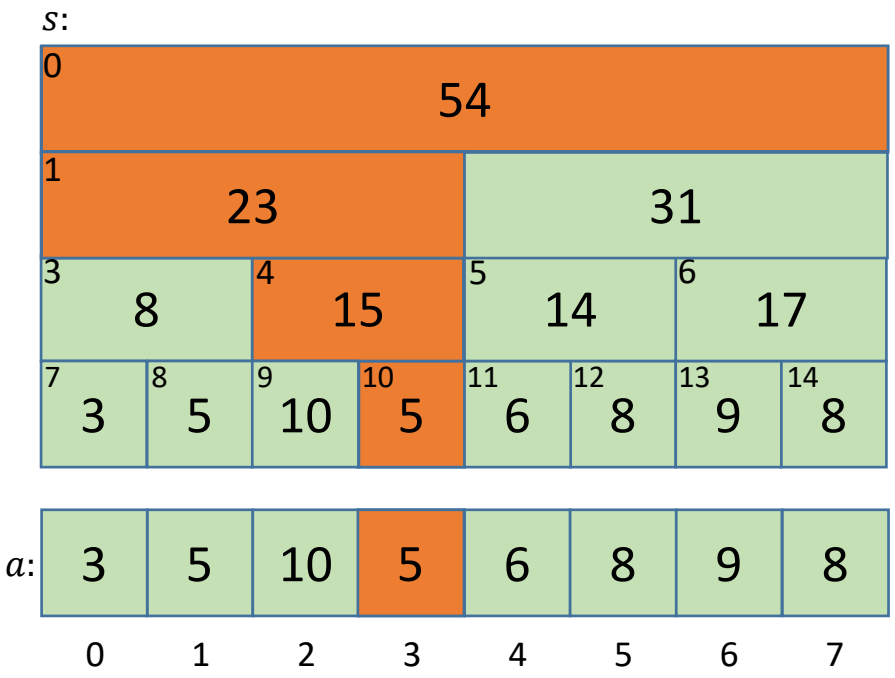
First we assign new value to the leaf corresponding to $a[i]$:

$$s[N-1+i] = val$$

And then iterate over parents until we reach root.

Each visited node value is refreshed:

$$s[i] = s[2 * i + 1] + s[2 * i + 2]$$



RSQ/RMQ: Segment tree

Implementation

```
class RSQSegmentTree:
    neutral_value = 0
    def __init__(self, a):
        self.N = 2 ** int(ceil(log2(len(a))))
        self.s = [None] * (self.N - 1) + list(a) + ([self.neutral_value] * (self.N - len(a)))
        for i in range(self.N - 2, -1, -1):
            self.refresh_s(i)

    def refresh_s(self, i):
        self.s[i] = self.s[2 * i + 1] + self.s[2 * i + 2]

    def rsq_i(self, l, r, i, li, ri):
        if (r <= li) or (ri <= l):
            return self.neutral_value
        if (l <= li) and (ri <= r):
            return self.s[i]
        middle = li + (ri - li) // 2
        return (self.rsq_i(l, r, i * 2 + 1, li, middle) +
                self.rsq_i(l, r, i * 2 + 2, middle, ri))

    def update(self, i, v):
        i += self.N - 1
        self.s[i] = v
        while i > 0:
            i = (i - 1) // 2
            self.refresh_s(i)

    def rsq(self, l, r):
        return self.rsq_i(l, r, 0, 0, self.N)
```

Complexity:
Preproc (build): $O(N)$
Query: $O(\log N)$
Update: $O(\log N)$

Conclusion

Structure	Preproc (build)	Query	Update
Prefix sum (RSQ)	$O(N)$	$O(1)$	$O(N)$
Sparse table (RMQ)	$O(N \log N)$	$O(1)$	$O(N \log N)$
Sqrt-decomposition (RSQ)	$O(N)$	$O(\sqrt{N})$	$O(1)$
Sqrt-decomposition (RMQ)	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$
Fenwick tree (RSQ)	$O(N \log N)$	$O(\log N)$	$O(\log N)$
Segment tree (RSQ/RMQ)	$O(N)$	$O(\log N)$	$O(\log N)$

Thank you for watching!