Augmenting search trees

Sergey V Kozlukov

2016-03-21 01:23:15

Contact info

- rerumnovarum@openmailbox.org
- GPG: **B986D856**

- Sources and examples are available at `https://github.com/RerumNovarum/vsu.en`

Foreword

Speech is supposed to

- take at most 15 minutes

Foreword

Speech is supposed to

- take at most 15 minutes
- be simple

Foreword

Speech is supposed to

- take at most 15 minutes
- be simple
- be demonstrative

Motivation

Why this subject?

- It's fundamental

Motivation

Why this subject?

- It's fundamental
- It's applicable

Cumulative problems

- Range-queries
- Range-updates
- Order statistics
- &c

RSQ (range sum query)

Given array `a[1..n]` of n numbers for given $1 \leq l \leq r \leq n$ calculate the sum of all numbers in subarray `a[l..r]`

RSQ (range sum query)

Given array `a[1..n]` of n numbers for given $1 \leq l \leq r \leq n$ calculate the sum of all numbers in subarray `a[l..r]`
In offline-variation, simple preprocessing (calculating cumulative partial sums) in $O(n)$ time allows to answer any such query in $O(1)$ time

RSQ (range sum query)

Given array `a[1..n]` of n numbers for given $1 \leq l \leq r \leq n$ calculate the sum of all numbers in subarray `a[l..r]`
In offline-variation, simple preprocessing (calculating cumulative partial sums) in $O(n)$ time allows to answer any such query in $O(1)$ time

```
def rsq(l, r):
  return cumsum[r] - (cumsum[l-1] if l != 0 else 0)
```

RMQ (range minimum query)

Given array `a[1..n]` of n objects of well-ordered set for given $1 \leq l \leq r \leq n$ find the minimal element in subarray `a[l..r]`

RMQ (range minimum query)

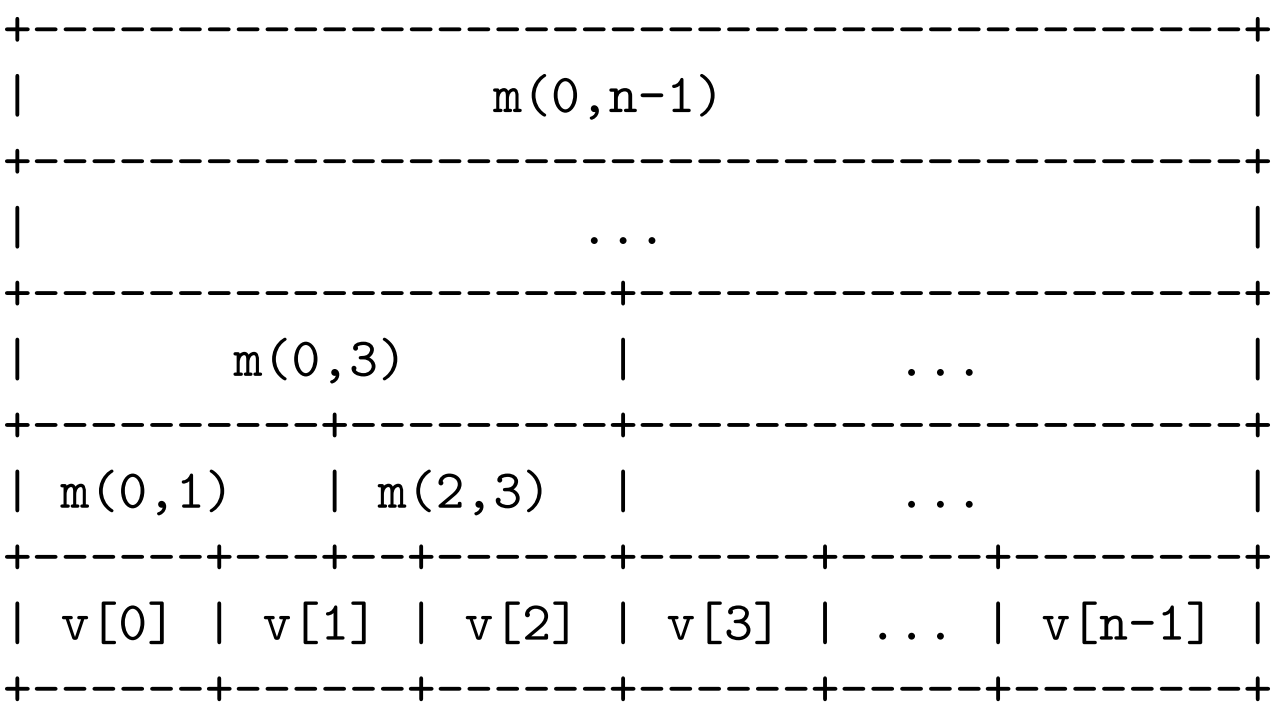
Given array `a[1..n]` of n objects of well-ordered set for given $1 \leq l \leq r \leq n$ find the minimal element in subarray `a[l..r]`
Old method won't work?

RMQ (range minimum query)

Given array `a[1..n]` of n objects of well-ordered set for given $1 \leq l \leq r \leq n$ find the minimal element in subarray `a[l..r]`
Old method won't work?
Actually much-much larger class of similar problems is solvable with generic approach! (stay tuned)

Array-based segments tree

Simple solution is to remember each value `v[i]`, then minimal value `m(2*k, 2*k+1)` in each pair `(v[2*k], v[2*k+1])`, then minimal value `m(4*k, 4*k+3)` and so on
Then answer can be found in logarithmic time



There are n intervals of length 1, floor$(n/2)$ intervals of length 2, ...

There are n intervals of length 1, floor$(n/2)$ intervals of length 2, ...
Total number of such segments is bounded by

$$\sum_k N/2^k = N \sum_k 2^{-k} = N \frac{1}{1-1/2} = 2N$$

Which means that we only need to use space linear in n

There are n intervals of length 1, floor$(n/2)$ intervals of length 2, ...
Total number of such segments is bounded by

$$\sum_k N/2^k = N \sum_k 2^{-k} = N \frac{1}{1-1/2} = 2N$$

Which means that we only need to use space linear in n
Let `s(l, r)` be precomputed minimal value in subarray `a[l..r]` where l and r are powers of 2

There are n intervals of length 1, floor$(n/2)$ intervals of length 2, ...
Total number of such segments is bounded by

$$\sum_k N/2^k = N \sum_k 2^{-k} = N \frac{1}{1-1/2} = 2N$$

Which means that we only need to use space linear in n
Let `s(l, r)` be precomputed minimal value in subarray `a[l..r]` where l and r are powers of 2
Begining with $tl = 0, tr = n - 1$ we can solve task with simple recursive logic

```
def rmq(l, r, tl, tr):
  if not intersects(l, r, tl, tr) or tl>tr: return INFINITY;
  if contains(l, r, tl, tr): return s(l,r)
  m = (tl+tr)//2
  infimum = INFINITY
  if intersects(l, r, tl, m):
    infimum = min(infimum, rmq(l, r, tl, m))
  if intersects(l, r, m+1, tr):
     infimum = min(infimum, rmq(l, r, m+1, tr))
  return infimum
```

RMQ

Such cache can be easily represented with binary tree, where first node is assigned to segment $[0, n-1]$, it's children to segments $[0, m-1]$ and $[m, n-1]$ and so on.

Lazy propagation

Let's consider another type of queries:
Given $k_1, k_2 \in K, v \in V$ associate all the keys $k : k_1 \leq k \leq k_2$ with value v

Lazy propagation

We can modify existing solution to store in node `cache` along with value and propagate it lazily to childrens as you go down

Online problem

Given n — maximal number of elements, and q — number of queries, handle q queries of following types:

- `put(k, v)`: set k'th item to be equal to v
- `get(l, r)`: find sum/minimum/whatever in subarray

Arbitrary keys

Now let's make our task a bit trickier and say, that we want, for example, say that keys are points in time or whatever with total order defined on it, instead of indices of array
And like before we want to perform

- `put(k, v)`
- `get(l, r)`

reasonably fast

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!
Can we handle range queries anyhow but bruteforcing?

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!
Can we handle range queries anyhow but bruteforcing?
YES!

Monoid

What's monoid?

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!
Can we handle range queries anyhow but bruteforcing?
YES!

Monoid

What's monoid?
Monoid is a semigroup with an identity element

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!
Can we handle range queries anyhow but bruteforcing?
YES!

Monoid

What's monoid?
Monoid is a semigroup with an identity element
Semigroup is an associative magma

Arbitrary keys

Since key's aren't integers bounded by some small constant, we can't use key as index in array
What to do?
We need some ordered Symbol Table
Binary Search Tree!
Can we handle range queries anyhow but bruteforcing?
YES!

Monoid

What's monoid?
Monoid is a semigroup with an identity element
Semigroup is an associative magma
Magma is just a set with some binary operation on it, w/o any restrictions

Monoid

Monoid is a set with an associative binary operation on it and an identity element regarding this operation

Monoid

Monoid is a set with an associative binary operation on it and an identity element regarding this operation
It happens that monoid is fundamental structure for range-queries

Augmentation

The idea is following:
If you're considering node n, which represents range `[n.lk, n.rk]`, and you know the multiple `n.l.mul` of values with keys in range `n.l.lk, n.l.rk` and multiple of `n.r.mul` of values with keys in range `n.r.lk, n.r.lk`, then simply `n.mul = n.l.mul * n.v * n.r.mul`

Augmentation

More generally, if range $l..r$ is requested, and we're in node n, then we either

- return 0 if $[n.lk, n.rk] \cap [l, r] = \emptyset$
- return `n.mul` if $[n.lk, n.rk] \subset [l, r]$
- recursively go into childrens and combine answers and value in current node

Augmentation theorem

Let K be totally-ordered set and $(M, \circ, 1)$ be monoid and let $T \subset M$ note values in tree.
Then following operations can be performed in time logarithmic in input size just by storing additional data in the nodes of tree and maintaining this data during rotations:

- `put(k, v)` Associate key k with value v
- `mul(l, r)` Calculate $v_{k_1} \circ v_{k_2} \circ \cdots \circ v_{k_m}$, where $v_{k_j} \in M \cap T$ and $l, r, k_j \in K$ and $l \leq k_j \leq r$ and $k_i \leq k_j$ for all $i \leq j = \overline{1, m}$

Go deeper

We're not actually restricted to use values set' monoid structure.

- Let $(K, \odot, I)$ be monoid too
- Their cartesian product $K \times M$ with product $\otimes = (\odot, \circ)$ and identity $J = (I, 1)$ is a monoid as well
- We can maintain $\odot$-multiple of keys in range in the same manner as multiple of values or, equivalently, $\otimes$-multiple of key-value pairs

Augmenting search trees

To insert into BST:

- Choose subtree to go
- Call 'insert' for it recursively
- Update aux data based on children's aux

Augmenting search trees

Balancing? Is based on rotations, these are local operations, so we can maintain auxiliary data during them

Augmenting search trees

Say red-black tree is implemented in recursive manner, with e.g. insertion looking like this:

```
    def subtree_put(self, h, k, v):
        """subtree_put(h, k, v)
h:  root of subtree
k:  key
v:  value"""
        if h is None: return self.Node(k, v)
        if k < h.k:      h.l = self.subtree_put(h.l, k, v)
        elif h.k < k:    h.r = self.subtree_put(h.r, k, v)
        else:            h.v = v
        self.restore(h)
        h = self.balance(h)
        return h
```

We can upgrade it to serve for range-queries by augmenting `Node` and overriding `restore()`

SegmentTree (API)

```
class SegmentTree(rbbst.RBBST):
    """SegmentTree(mul, id)
mul:    multiplication operation (callable, binary)
id:     identity element
*   put(k, v) where $k\\in K, v\\in M$
        associates key 'k' with value 'v'
*   get(k)  where $k\\in K$
        retrieve value associated with key 'k' if any
*   mul(l, r) where $l,r\\in K$ and $l\\leq r$"""
```

SegmentTree (Sample client)

```
# SegmentTree for Monoid of numbers
# with regular addition operation
# and 0 as identity element
def add(x, y): return x+y
t = SegmentTree(add, 0)
t.put('a', 1) # associate value 1 with key 'a'
t.put('b', 3)
t.put('c', 22)
t.mul('a', 'c') # -> 26
t.mul('a', 'a') # -> 1
```

SegmentTree (API)

```
# SegmentTree for Monoid of strings
# with associative operation of concatenation
# and empty string as identity element
def add(x, y): return x+y
t = SegmentTree(add, '')
t.put(1, 'some ')
t.put(10**32, 'strings') # we can use some large 'indices'
t.put(-10**9, 'concat ')
t.mul(-10**64, 10**64)   # yields 'concat some strings'
```

SegmentTree (Augmentation)

We'll augment each node `h` to store multiple 'h.mul' of elements in range from 'h.lk' through 'h.rk'

```
class Node(rbbst.Node):
    def __init__(self, k, v):
        super(Node, self).__init__(k, v)
        # we insert every new `Node` as a leaf
        # so it represents segment [k,k]
        self.lk  = self.rk = k
        # and multiple in this segment is simply `v`
        self.mul = v
```

SegmentTree (Maintaining aux data)

```
    def restore(self, h):
        """restore(h)
overrided `restore` will update cumulative
after insertions and balancing"""
        assert not h is None
        h.lk, h.rk = h.k, h.k
        m = h.v
        if h.l:
            m = self.mulbin(h.l.mul, m)
            h.lk = h.l.lk
        if h.r:
            m = self.mulbin(m, h.r.mul)
            h.rk = h.r.rk
        h.mul = m
```

SegmentTree query

```
    def subtree_mul(self, h, l, r):
        """subtree_mul(h, l, r)
calculates cumulative in intersection of (h.lk, h.rk) and (l, r)"""
        s = self.id
        if h is None: return s
        if l <= h.lk <= h.rk <= r: return h.mul
        if h.l and intersects(h.l.lk, h.l.rk, l, r):
            s = self.mulbin(self.subtree_mul(h.l, l, r), s)
        if l <= h.k <= r:
            s = self.mulbin(s, h.v)
        if h.r and intersects(h.r.lk, h.r.rk, l, r):
            s = self.mulbin(s, self.subtree_mul(h.r, l, r))
        return s
```

Examples

Range-query

Range-minimum, range-maximum, range-sum and range-whatever queries

Solution: Store "multiple" in node

Order statistics

1. *For given key* `k` *find it's position in ordered sequence of all the keys in the tree*
2. *For given position* `j` *find key* `k_j` *at this position in ordered sequence of all the keys in the tree*

Solution: Store size of subtree in each node

References

- `http://e-maxx.ru/algo/segment_tree`
- CLRS: "Introduction to algorithms", chapter "Augmenting binary trees"
- Sedgewick, Wayne: "Algorithms, part 4", red-black trees
- "Мочкошка и их применения", http://habrahabr.ru/post/112304/