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 5-10 minutes

Foreword

Speech is supposed to

- take 5-10 minutes
- be simple

Foreword

Speech is supposed to

- take 5-10 minutes
- be simple

Foreword

Speech is supposed to

- take 5-10 minutes
- be simple

But it's not completed yet so neither of these goals is achieved

Motivation

Why this subject?

- It's fundamental

Motivation

Why this subject?

- It's fundamental
- It's applicable

RSQ (range sum query)

Given array `a[1..n]` of n numbers for given $1 \le l \le r \le n$ 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]`
A bit more complicated?

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]`

A bit more complicated?

Actually much-much larger class of similar problems is solvable with generic approach! (stay tuned)

Simple solution is to remember minimal values in whole array `a[1..n]`, subarray `a[1..n/2]`, `a[n/2+1..n]`, ... (continuing splitting intervals, until $l = r$)

Simple solution is to remember minimal values in whole array `a[1..n]`, subarray `a[1..n/2], a[n/2+1..n]`, ... (continuing splitting intervals, until $l = r$)

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

Simple solution is to remember minimal values in whole array `a[1..n]`, subarray `a[1..n/2], a[n/2+1..n]`, ... (continuing splitting intervals, until $l = r$)

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 linear in n space

Simple solution is to remember minimal values in whole array `a[1..n]`, subarray `a[1..n/2], a[n/2+1..n]`, ... (continuing splitting intervals, until $l = r$)
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 linear in n space
Let `cache(l, r)` be precomputed minimal value in subarray `a[l..r]` where l and r are powers of 2

Simple solution is to remember minimal values in whole array `a[1..n]`, subarray `a[1..n/2], a[n/2+1..n]`, ... (continuing splitting intervals, until $l = r$)

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 linear in n space

Let `cache(l, r)` be precomputed minimal value in subarray `a[l..r]` where l and r are powers of 2

Begining with $tl = 1, tr = n$ we can solve task with simple recursive logic

```python
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 cache(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 firt node assigned to segment $1..n$, it's children to segments $1..m$ and $m + 1..n$ and so on.

RSQ using array-based tree

Let's represent binary tree with array indices arithmetics:

- Each node is assigned a number v, which is index in array
- Root's index is 0
- Left child of v has index $2v + 1$
- Right child of v has index $2v + 2$
- Parent of v has index floor$((v - 1)/2)$

RSQ using array-based tree (API)

```
template<typename T>
class RSQ {
  private:
    int n;
    int buffsize;
    T*  tree;
  public:
    RSQ(int n);
    ~RSQ();
    void put(int k, T v);
    T get(int k);
    T sum(int l, int r);
  private:
    // i
    put(int i, T v, int ti, int tl, int tr);
    T sum(int l, int r, int ti, int tl, int tr);
```

Constructor

```
RSQ(int n) {
  this->n    = n;
  this->buffsize=4*n;
  this->tree = new T[this->buffsize];
  for (int i = this->buffsize - 1; i >= 0; --i) this->tree[i] = 0;
}
~RSQ() {
  delete[] this->tree;
}
```

Retrieving

```
T get(int l, int r, int ti, int tl, int tr) {
  if (ti >= this->buffsize) return 0;
  if (tl > tr) return 0;
  if (l <= tl && tr <= r) return this->tree[ti];
  int m = (tl+tr)/2;
  T s = 0;
  if (intersects(l, r, tl, m))   s += get(l, r, left(ti),  tl,  m);
  if (intersects(l, r, m+1, tr)) s += get(l, r, right(ti), m+1, tr);
  return s;
}
```

Updating

```
void set(int i, T v, int ti, int l, int r) {
  if (ti >= this->buffsize) return;
  if (l>r)  return;
  if (l==r) {
    this->tree[ti] = v;
  }
  else {
    int m = (l+r)/2;
    if (i <= m) set(i, v, left(ti),  l,   m);
    else        set(i, v, right(ti), m+1, r);
    this->tree[ti] = this->tree[left(ti)] + this->tree[right(ti)];
  }
}
```

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

For array-based implementation solution is 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, it's straightforward

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, it's straightforward

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, it's straightforward

Can we handle range queries anyhow but bruteforcing?

YES!

Monoid

What's monoid?

Monoid

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

Monoid

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

Monoid

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

Monoid

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

Monoid

What's monoid?
Monoid is a semigroup with an identity element
What's semigroup?
Semigroup is an associative magma
And what the heck is 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
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 $i \leq j = \overline{1, m}$

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 local rotations, so we can maintain auxiliary data during them

Augmenting search trees

```
    def __put__(self, h, k, v):
        """__put__(h, k, v)
h:  root of subtree
k:  key
v:  value
recursive method to insert new kv-pair into tree
and maintain balance"""
        if h is None: return self.Node(k, v)
        if k < h.k:     h.l = self.__put__(h.l, k, v)
        elif h.k < k:   h.r = self.__put__(h.r, k, v)
        else:           h.v = v
        self.restore(h) # here we maintain aux
        h = self.balance(h) # and here too
        return h
```

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 (API)

```python
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)

```
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 maintaining aux data

```python
    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

```python
    def __mul__(self, h, l, r):
        """__mul__(h, l, r)
calculates cumulative in intersection of (h.lk, h.rk) and (l, r)"""
        s = self.id
        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.__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.__mul__(h.r, l, r))
        return s
```

References

- `http://e-maxx.ru/algo/segment_tree`
- CLRS: “Introduction to algorithms”, chapter “Augmenting binary trees”
- Sedgewick, Wayne: “Algorithms, part 4”, red-black trees
- “Моноиды и их приложения”, http://habrahabr.ru/post/112394/

Contact info

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

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