Augmenting search trees

Sergey V Kozlukov

2016-03-21 01:23:15

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]`
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 each subarray `a[2**k, 2**(k+1)]`.

Simple solution is to remember minimal values in each subarray `a[2**k, 2**(k+1)]`.

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

Simple solution is to remember minimal values in each subarray `a[2**k, 2**(k+1)]`.

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 each subarray `a[2**k, 2**(k+1)]`.

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 each subarray `a[2**k, 2**(k+1)]`.
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:

```
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,
```

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.

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

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

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 $i \leq j = \overline{1, m}$

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 $\text{floor}((v - 1)/2)$

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;
```