Segment Tree and Lazy Propagation

beOI Training

March 11, 2016

Table of Contents

Motivating problem

You are given an integer array A of size n ($n < 10^6$).
Given two integers a and b, can you give the minimum value of A between indices a and b ?

$$\min_{i=a...b} A[i]$$

Well that's easy, just iterate over the interval and return the minimum!

Motivating problem

You are given an integer array A of size n ($n < 10^6$).
Given two integers a and b, can you give give the minimum value of A between indices a and b ?

$$\min_{i=a\ldots b} A[i]$$

100000 times?

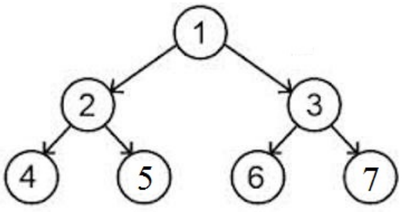
This is called the **range minimum query** (RMQ) problem.

Naive solution

For each query, iterate over the corresponding range and return the minimum.
If k is the number of queries, time complexity is $\mathcal{O}(nk)$.
TLE

Array representation of a binary tree

- 1-based array, index $1 = $ root
- For each node of index p,
  - left child has index $2p$
  - right child has index $2p + 1$



Segment Tree

Each node is responsible of one segment
Root represents the whole array $[0 \cdots n-1]$
Given a node representing segment $[l \cdots r]$

- left child represents the segment's first half $\left[l \cdots \frac{l+r}{2}\right]$
- right child represents the segment's second half $\left[\frac{l+r}{2}+1 \cdots r\right]$

The value of a node will be the **index of the minimum element in segment** $[l \cdots r]$.

Querying

When we query the minimum value in an interval, we look for **big segments that are contained within the query range**, and among those segments, take the minimum value. Recursively,

- segment is within query range $\Rightarrow$ return value of the node;
- segment and query range are disjoint $\Rightarrow$ do nothing;
- otherwise, return minimum among both children.

Querying implementation

```
// p is array index of current node,
// [L..R] is current segment,
// [i..j] is search interval
int query(int p, int L, int R, int i, int j) {
    // inside  query range
    if(L >= i && R <= j) return st[p];
    // outside query range
    if(i >  R || j <  L) return -1;

    // compute the min position in the left
    // and right part of the interval
    int p1 = query(2*p, L, (L+R)/2, i, j);
    int p2 = query(2*p+1, (L+R)/2+1, R, i, j);

    // if we try to access segment outside query
    if(p1 == -1) return p2;
    if(p2 == -1) return p1;
    return (A[p1] <= A[p2]) ? p1 : p2;
}
```

Querying complexity

At each level, at most 4 nodes are visited (see coach for proof).
There are exactly $\lceil \log_2 n \rceil$ levels.

$$\mathcal{O}(4 \times \lceil \log_2 n \rceil) = \mathcal{O}(\log n)$$

Overall complexity $\mathcal{O}(k \log n)$ is now reasonable!
AC

Building

Building the Segment Tree is also done recursively.
For each node,

- if no child, store current index;
- otherwise,
  - build left child;
  - build right child;
  - store minimum child.

Building implementation

```
void build(int p, int L, int R) {
    if(L == R)
        // leaf node
        st[p] = L;
    else {
        // build children
        build(2*p, L, (L+2)/2);
        build(2*p+1, (L+R)/2+1, R);
        // take minimum among them
        int p1 = st[2*p], p2 = st[2*p+1];
        st[p] = (A[p1] <= A[p2]) ? p1 : p2;
    }
}
```

Building complexity

We visit every node once.
In general, the number of nodes is
$N + \frac{N}{2} + \frac{N}{4} + \cdots + 2 + 1 \approx 2N$, so time complexity is

$$\mathcal{O}(2 \times N) = \mathcal{O}(N)$$

This also proves memory is $\mathcal{O}(N)$ (in practice one always takes an array of $4 \times N$ for safety).

Segment Trees are extremely powerful!

We saw how to solve the range **minimum** query problem.
But we can do much more than that!

- Range maximum query
- Range sum query
- Range *insert any function here* query

One last operation

Suppose that, between queries, the array is being **updated**.
Naive solution: re-build the Segment Tree in $\mathcal{O}(N)$.
TLE
Segment Trees allow efficient **updating**!

Updating

To update p, we only need to update the segments that contain p.
Update the leaf to root path in $\mathcal{O}(\log N)$!

Updating implementation

```
// i is the node that is to be updated
void update(int p, int L, int R, int i) {
    // if leaf node
    if(L == R) return;
    // if i is in segment
    if(i >= L && i <= R) {
        // if new value is smaller, update
        if(A[i] <= st[p])
            st[p] = i;
        update(2*p, L, (L+R)/2, i);
        update(2*p+1, (L+R)/2+1, R, i);
    }
}
```

Table of Contents

Motivating problem

In the Range Sum Query (RSQ) problem, we add one operation: range update.
We want to update a range (e.g. increment every value in range by k) efficiently.

Naive solution

At each range update query, re-build tree in $\mathcal{O}(N)$.
TLE

Let's be lazy!

Key idea behind lazy Segment Tree: don't update everything at once; put a flag on segments that need to be updated, and leave it for another traversal.

Propagation

Keep an array lazy that stores for each segment by how much each value needs to be incremented.
Every time we visit a node p (in query or update) where lazy[p] != 0,

- increment current segment by lazy[p] times size of segment;
- if node is not leaf,
  - increment lazy[2*p] by lazy[p]
  - increment lazy[2*p+1] by lazy[p]
- reset lazy[p].

That is called **propagation**.
Obviously, complexity is $\mathcal{O}(1)$.

Propagation implementation

```
void propagate(int p, int L, int R) {
    if(lazy[p] != 0) {
        st[p] += (L-R+1)*lazy[p];

        if(L != R) {
            lazy[2*p] += lazy[p];
            lazy[2*p+1] += lazy[p];
        }

        lazy[p] = 0;
    }
}
```

Querying

We do exactly the same, but we propagate at each node!
Complexity $\mathcal{O}(\log N)$.

Querying implementation

```
int query(int p, int L, int R, int i, int j) {
    propagate(p, L, R);

    if(i > R || j < L) return 0;
    if(L >= i && R <= j) return st[p];

    return query(2*p, L, (L+R)/2, i, j)
        + query(2*p+1, (L+R)/2+1, i, j);
}
```

Updating

For each node,

- propagate
- if outside of range, return
- if inside of range, set the lazy flag, and return
- otherwise
  - update left child
  - update right child
- merge both children (add them up)

Complexity $\mathcal{O}(\log N)$.

Updating implementation

```
void update(int p, int L, int R, int i, int j, int k) {
    propagate(p, L, R);

    if(i > R || j < L) return;
    if(L >= i && R <= j) {
        lazy[p] = k;
        return;
    }

    update(2*p, L, (L+R)/2, i, j, k);
    update(2*p+1, (L+R)/2+1, R, i, j, k);

    st[p] = st[2*p] + st[2*p+1];
}
```