Data Structures (III)

Ian {ycwong}
2021-04-03

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Sub-Topics

1. Sparse Table

2. Segment Tree
   2.1. Lazy Propagation

3. 1-d/2-d binary indexed tree

Range Minimum Query

Given an integer array A with length N and M queries.
Each query is a range [L, R], and you have to return min(A[i] | L <= i <= R)

E.g. A = [5, 4, 1, 2, 3], query = [{1, 5}, {1, 2}, {4, 5}, {4, 4}].

Ans = [1, 4, 2, 2]

Range Minimum Query

Naive solution: O(N * Q) where Q is the number of query.

Just do what the problem said. Using a for loop searching the minimum inside the range.

Can we do better?

Range Minimum Query

Of course the answer is yes otherwise the session can end right now.

One of solutions is to use sparse table.

Sparse Table

Now just assume all the query have range of length 2^x.

So, if we can precompute a function f(i, x) = min(a[j] | i <= j < (i + 2^x)) efficiently, we can answer in O(1).

E.g. f(3, 2) = min(A[3], A[4], A[5], A[6])

Since x <= ceil($\log_2(N)$), the number of useful parameters is O(N * log(N)), which means we only need a 2-d array with size N * log(N) to store the values.

Sparse Table

In fact, we can compute the table in O(N * log(N)) which is exactly the same as the space complexity.

The main idea is that, once we compute f(1..N, x), we can then compute f(1..N, x + 1) based on only f(i, x).

f(3, 2) = min(A[3], A[4], A[5], A[6])

f(3, 2) = min(f(3, 1), f(5, 1))

Sparse Table

Consider A = [5, 4, 1, 2, 3], f(0, i) = A[i]

	1	2	3	4	5
2					
1					
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 1

	1	2	3	4	5
2					
1	4				
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 1

	1	2	3	4	5
2					
1	4	1			
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 1

	1	2	3	4	5
2					
1	4	1	1		
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 1

	1	2	3	4	5
2					
1	4	1	1	2	
0	5	4	1	2	3

Sparse Table

Notice that we don't have to compute f(1, 5) since 5 + 2^1 - 1 > N and we will never use this cell.

	1	2	3	4	5
2					
1	4	1	1	2	
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 2

	1	2	3	4	5
2	1				
1	4	1	1	2	
0	5	4	1	2	3

Sparse Table

Consider A = [5, 4, 1, 2, 3], now we compute layer 2

	1	2	3	4	5
2	1	1			
1	4	1	1	2	
0	5	4	1	2	3

Sparse Table

Now we can solve all query with range that is a power of 2. But how about query with arbitrary range?

Turns out it is kind of the same.

Let k be the maximum where (R - L + 1) >= 2^k. Now notice that 2^k must be at least half of the query range (otherwise k is not maximum).

Answer will be min(f(k, L), f(k, R - 2^k + 1)). (since we know 2 * 2^k >= (R - L + 1))

Sparse Table

A = [5, 4, 1, 2, 3]

Query = [2, 4]

[L, R] = f(k, L), f(k, R - 2^k + 1)

k = 1 2^1 = 2 <= 3

min(f(k, 2), f(k, 4-2^k+1))

f(k, 2) = min(A[2], A[3])

f(k, 4 - 2^k + 1) = f(k, 3) = min(A[3], A[4])

Sparse Table

So, now we can solve the rmq problem with precompute time O(N log N) and query time O(1).

Also, the information of f(i, x) is not limited to min/max. We can also use it to store the gcd/and/or of a range and answer query in O(1). But not sum/xor/product. (they can be solved in O(log N) query time with sparse table tho)

Sparse Table

Dist = 2^i + 2^j + 2^k + ... +

Pos = L

Sum = 0

For i in 0..log(N)

    If i-th of Dist is 1

        Sum += f(i, pos)

        Pos += 2^i

Sparse Table For LCA

We can define f(i, v) as the 2^i parent of node v, e.g. f(0, v) = parent(v).

If we want to find the lca of v and u.

Without loss of generality, assume depth(v) >= depth(u). We lift v up such that depth(v) = depth(u). If v = u then u is the LCA.

Else, for each 2^x (from high to low), if f(x, v) <> f(x, u), we lift them together, i.e. v = f(x, v) and u = f(x, u).

Finally, f(0, v) will be the answer.

Sparse Table For LCA

Now the problem is the first part, how do we lift node v to same depth of node u. Let distance = depth(v) - depth(u), seeing the binary representation of distance will let us know how to lift.

E.g. distance = 3. Lift v to its {2^0}-th parent. Then lift it again to its {2^1}-th parent.

Sparse Table

LG(i) = 2^k | 2^k <= i, k maximum

LG(i) = LG(i / 2) + 1

LG(1) = 0

Practice Problem

https://judge.hkoi.org/task/M0921

https://judge.hkoi.org/task/T181

Extra Readings

https://cp-algorithms.com/data_structures/sparse-table.html

Segment Tree

Now we add another type of query in our rmq problem.

Some query will give two integers x, val and update a[x] = val.

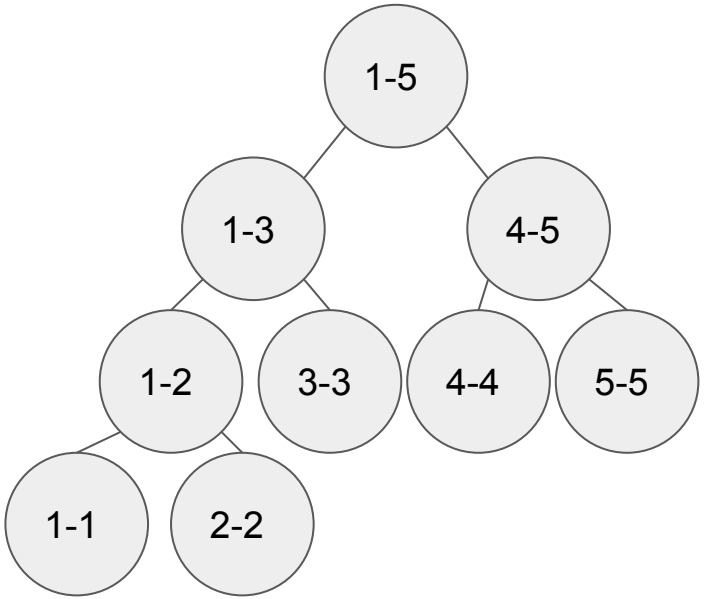
We now can’t solve it with sparse table as it doesn’t support update. (as we have to compute the whole table again)

Segment Tree

Imagine we have a binary tree. Each node stores the minimum of an interval [L, R]. And its left child stores information of [L, mid] and right child stores information of [mid + 1, R].

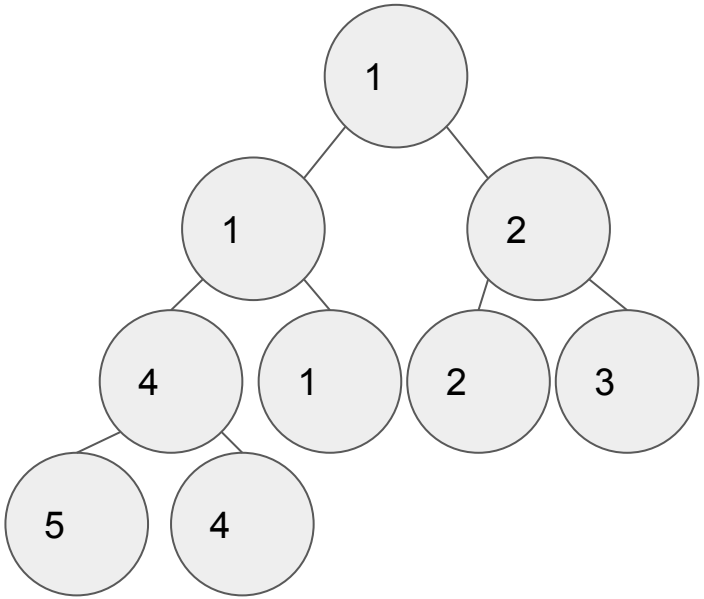
Segment Tree

A = [5, 4, 1, 2, 3]



Segment Tree

A = [5, 4, 1, 2, 3]



Segment Tree

First, let's walk through how to build the segment tree when we receive the array A.

We can recursively construct it. For convenience, if the current node is with id X, then we use X*2 as left child and X*2 + 1 as right child.

Segment Tree

```
build(id, x, y)
    If x = y then
        node[id] = A[x]
        return
    mid = (x + y) / 2
    build(id * 2, x, mid)
    build(id * 2 + 1, mid + 1, y)
    node[id] = min(node[id * 2], node[id * 2 + 1])
build(1, 1, N)
```

Segment Tree

We know if the length of A is a power of two, total number of node will be N + (N - 1) which is approximately 2 * N.

When length of A is not power of two, we need to be safe and set the size of the node array as 4 * N. (as we try to make it at least larger than 2^k where 2^k > N)

So the space complexity of segment tree is O(N)

Segment Tree

Ok now the question is how do we query efficiently using segment tree.

Again, we can use a recursive approach.

Segment Tree

```
query(id, x, y, L, R)  // node[id] responsible for range [x, y] and now query[L, R]
    if L > y or R < x // no intersection of [x, y] and [L, R]
        return inf // don't want to affect the answer so return inf
    if L <= x and y <= R // [x, y] is fully inside [L, R]
        return node[id]
    mid = (L + R) / 2
    return min(query(id * 2, x, mid, L, R), query(id * 2 + 1, mid + 1, y, L, R))

query(1, 1, N, L, R)
```

Segment Tree

The query time complexity is O(log N).

Why? Because in each level, we will only use at most two node. Notice that for the node we use in each level, they must be consecutive. So if there are three nodes, we can merge two of the three nodes and use it's parent instead.

Segment Tree

Ok, the remaining question is how to handle the update operation.

We also can solve it using a recursive approach, which is similar to building the segment tree.

Segment Tree

```
update(id, x, y, pos, val)
    If x = y
        node[id] = val
        return
    mid = (x + y) / 2
    If pos <= mid -> update(id * 2, x, mid, pos, val)
    Else -> update(id * 2 + 1, mid + 1, y, pos, val)
    node[id] = min(node[id * 2], node[id * 2 + 1])
```

Segment Tree

Cool, now we can solve range query and point update. (also point query and range update)

Also, just like sparse table, the information stored in the node is not limited to min/max. And it is more flexible since when we query, there is no overlapping interval unlike sparse table.

You can store something like prefix min/max, hash sum, dp table etc.

Practice Problem

https://judge.hkoi.org/task/M0921

https://judge.hkoi.org/task/M0923

https://judge.hkoi.org/task/T152

https://codeforces.com/contest/438/problem/D

Extra Reading

https://cp-algorithms.com/data_structures/segment_tree.html

Lazy Propagation

Imagine we now modify the update query.

It now will give three integers L, R, val and update a[i] += val where L <= i <= R.

We can't simply solve it with the code above since it involves both range query and range update. (why?)

Lazy Propagation

But we can solve this problem lazily.

We store some intermediate information in each node, and once we need to evaluate its children, we propagate the information to the children, that's how the name 'Lazy Propagation' comes from as we only evaluate the extra information when we need to.

Lazy Propagation

push(from, to)

    node[to] += lazy[from]

    lazy[to] += lazy[from]

Lazy[id] = value we have updated in current node id, but not its left and right child.

Lazy Propagation

```
query(id, x, y, L, R)
    ...
    push(id, id * 2)
    push(id, id * 2 + 1)
    lazy[id] = 0 // already propagated the information, we can clear it now
    return min(query(id * 2, x, mid, L, R), query(id * 2 + 1, mid + 1, y, L, R))
```

Lazy Propagation

```
update(id, x, y, L, R, val)
    if L > y or R < X return
    if L <= x and y <= R
        Node[id] += val
        Lazy[id] += val
        return
    ...
```

```
push(id, id * 2)
push(id, id * 2 + 1)
lazy[id] = 0
update(id * 2, x, mid, L, R, val)
update(id * 2 + 1, mid + 1, y, L, R, val)
node[id] = min(node[id * 2], node[id * 2 + 1]
```

Lazy Propagation

Again, lazy propagation is not limited to min/max/sum/product, it can solve sum + product, set a range to the same value etc.

And the time complexity of lazy propagation is also easy to see, as it is only invoked at most twice when we call the update/query, so it is also O(log N).

Lazy Propagation

Segment tree + lazy propagation is so flexible and sometimes it can solve tasks that are not intuitively segment tree problem.

Practice Problem

https://judge.hkoi.org/task/T192

https://codeforces.com/contest/446/problem/C

Extra Reading

https://cp-algorithms.com/data_structures/segment_tree.html#toc-tgt-10

Binary Indexed Tree

Almost all of the things that you can do in BIT can also be done with segment tree. So the advantage of BIT is easier to code. It uses less lines and storage.

The main idea is using the binary representation of the id.

Binary Indexed Tree

Define lowbit(x) as the value of the rightmost bit in binary representation of x.

Let $x = 22 = 10110_2$, $lowbit(x) = 00010_2 = 2$.

Node x maintain information for [x – lowbit(x) + 1, x].

We can find lowbit(x) in O(1) using lowbit(x) = x & -x.

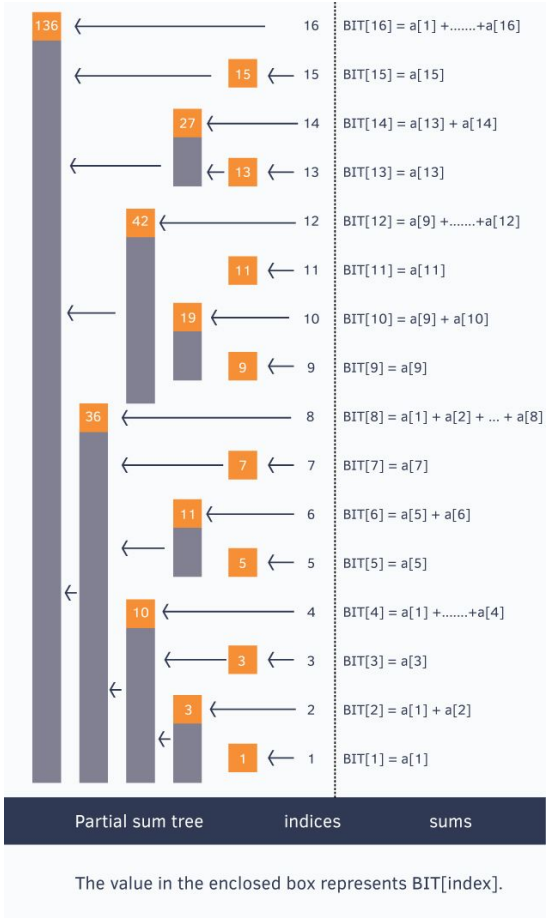
Binary Indexed Tree

Simple problem, Given an Integer array A with length N. We have two types of operation.

1. Two integers id and val are given. Update A[id] += val.
2. An integer id is given. Find A[1] + ... + A[id].

Binary Indexed Tree

https://www.hackerearth.com/practice/notes/binary-indexed-tree-or-fenwick-tree/



香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Binary Indexed Tree

```
add(id, val)
    while id <= N
        node[id] += val
        id += id & -id
```

```
sum(id)
    res = 0
    while id > 0
        res += node[id]
        id -= id & -id
    return res
```

Practice Problem

https://codeforces.com/problemset/problem/830/B

2D Data Structure

We can extend our segment tree/BIT into a 2D data structure, where each node is another segment tree/BIT.

For example, in 2D BIT, each node is another BIT. We just have to slightly modify our add and get function.

2D Data Structure

```
add(x, y, val)
    while x <= N
        tmp = y
        while y <= M
            node[x][y] += val
            y += y & -y
        x += x & -x
        y = tmp
```

```
get(x, y, val) // sum[1..x, 1..y]
    res = 0
    while x > 0
        tmp = y
        while y > 0
            res += node[x][y]
            y -= y & -y
        x -= x & -x
        y = tmp
    return res
```

2D Data Structure

2D data structures generally has a higher time complexity and memory storage. It is quite rare to see a problem that requires 2D data structure to solve. But still, if you know how to implement a 1D data structure, 2D data structure is not that difficult to implement, tho sometime it is quite tedious.

Practice Problem

https://judge.hkoi.org/task/I0111