Data Structures (III)

Bryan Chung {kobebryan9}

2024-03-23

Reference

- This slide is mainly adapted from Data Structures (III) slides (2023) by Gabriel
  - https://assets.hkoi.org/training2023/ds-iii.pdf
- OI Wiki
  - https://oi-wiki.org/
- Algorithms for Competitive Programming (CP-Algorithms)
  - https://cp-algorithms.com/index.html
- ITMO Academy: pilot course in Codeforces
  - https://codeforces.com/edu/course/2

Agenda

Sparse Table

- Range Minimum Query (RMQ)
- Binary Lifting

Segment Tree

- Update & Query
- Lazy Propagation

Binary Indexed Tree (Fenwick Tree)

- “Prefix Sum with Efficient Update”

Problem list (Segment Tree)

https://codeforces.com/blog/entry/22616

https://codeforces.com/blog/entry/71925

https://codeforces.com/edu/course/2/lesson/4

https://codeforces.com/edu/course/2/lesson/5

https://codeforces.com/blog/entry/57319

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

https://codeforces.com/gym/104090/problem/M

https://loj.ac/p/2269

Range Minimum Query (RMQ)

Given an array A of N integers and Q queries

Given L and R for each query, find the minimum value in A[L], A[L + 1], …, A[R]

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

L = 1, R = 2 → min value = 3

L = 2, R = 4 → min value = 1

L = 4, R = 5 → min value = 2

Range Minimum Query (RMQ)

Naive Solution: for every query, loop from L to R and take min

Time Complexity: O(QN)

Time Limit Exceeded when QN is large :(

How can we do better?

Range Minimum Query (RMQ)

Answer: use sparse table for sure :)

Sparse Table

Let's solve an easier problem first: assume every query's range is 2^x for some x

E.g. L = 2, R = 9 → range = 9 - 2 + 1 = 8, which is 2^3

If we precompute all possible ranges for every starting position, we can answer the query in O(1)

Formally, we want to precompute $f(i, x) = \min(A[j] \mid i <= j < (i + 2^x))$

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

Sparse Table

Since $x <= \log_2(N)$, we only need $O(N \log N)$ space to store all possible values

Also, we can compute all values in $O(N \log N)$

Main idea: compute $f(1..N, x + 1)$ from $f(1..N, x)$

Sparse Table

Consider A = {3, 4, 1, 5, 2}

For x = 0,

f(i, 0) = min value from A[i] to A[i + 2^0 - 1] = A[i]

E.g. f(2, 0) = A[2]

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1					
x = 0	3	4	1	5	2

Sparse Table

Consider A = {3, 4, 1, 5, 2}

Now for x = 1,

f(i, 1) = min value from A[i] to A[i + 2^1 - 1] = min(A[i], A[i + 1])

∴ f(1, 1) = min(A[1], A[2])

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1	3				
x = 0	3	4	1	5	2

Sparse Table

f(2, 1) = min(A[2], A[3])

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1	3	1			
x = 0	3	4	1	5	2

Sparse Table

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1	3	1	1		
x = 0	3	4	1	5	2

Sparse Table

f(4, 1) = min(A[4], A[5])

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1	3	1	1	2	
x = 0	3	4	1	5	2

Sparse Table

We don't need f(5, 1) = min(A[5], A[6]) as we don't have A[6]

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2					
x = 1	3	1	1	2	
x = 0	3	4	1	5	2

Sparse Table

Now for x = 2,

f(i, 2) = min value from A[i] to A[i + 2^2 - 1] = min{A[i], A[i + 1], A[i + 2], A[i + 3]}

Instead of looping from i to i + 3, we can compute f(i, 2) from f(i, 1) and f(i + 2, 1)!

f(1, 2) = min(f(1, 1), f(3, 1)), f(1, 1) = min(A[1], A[2]), f(3, 1) = min(A[3], A[4])

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2	1				
x = 1	3	1	1	2	
x = 0	3	4	1	5	2

Sparse Table

In general, $f(i, x + 1) = \min(f(i, x), f(i + 2^x, x))$

$\therefore f(2, 2) = \min(f(2, 1), f(4, 1))$

	i = 1	i = 2	i = 3	i = 4	i = 5
x = 2	1	1			
x = 1	3	1	1	2	
x = 0	3	4	1	5	2

Sparse Table

We solved the easier version where every query's range is 2^x

How about for queries with arbitrary range?

Sparse Table

Observation: min{a, b, c} = min(min(a, b), min(b, c))

Overlapped ranges does not affect the result of min value!

So we just need **two** values from the sparse table:

min(A[L..y]) & min(A[x..R]), and x can be ≤ y

Sparse Table

E.g. A = {3, 4, 1, 5, 2}, L = 2, R = 4

f(2, 1) = min(A[2], A[3])

f(3, 1) = min(A[3], A[4])

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

Sparse Table

Let k be the maximum integer such that $R - L + 1 \geq 2^k$

$[L, L + 2^k - 1]$ and $[R - 2^k + 1, R]$ must cover all positions from L to R

$\therefore$ Answer = $\min(f(L, k), f(R - 2^k + 1, k))$

Sparse Table

E.g. $L = 5, R = 16 \rightarrow R - L + 1 = 12$

$k = 3$ $(2^3 = 8 \leq 12, 2^4 = 16 > 12)$

$f(5, 3) = \min(A[5..12])$
$f(9, 3) = \min(A[9..16])$

$\text{Answer} = \min(f(5, 3), f(9, 3))$

Pseudocode

```
precompute()
  for i = 1 to N
    ST[i][0] = A[i]
  for x = 0 to ⌊log2(N)⌋ - 1
    for i = 1 to N - (2^(x + 1) - 1)
      ST[i][x + 1] = min(ST[i][x], ST[i + 2^x][x])
```

Pseudocode

```
query(L, R)
   k = ⌊log₂(R - L + 1)⌋
   return min(ST[L][k], ST[R - 2^k + 1][k])
```

Reminder: in order to calculate $\log_2$ in C++ efficiently, either use `std::__lg`, or precompute by `logN[1] = 0, logN[i] = logN[i / 2] + 1`

Sparse Table

We solved the problem with O(N log N) precomputation and O(1) query!

If you still remember, the reason why we can have O(1) query is that **overlapped ranges** does not affect the result of min value

Therefore, as long as the value of an operation would not be affected by **overlapped ranges**, we can have O(1) query using sparse table

- max / and / or / gcd
- any operation that is idempotent

Sparse Table

Although we cannot have O(1) query for sum / product / xor, we can still have O(log N) query, as we only need at most O(log N) values from the sparse table

E.g. $L = 7, R = 19 \rightarrow R - L + 1 = 13 \rightarrow 1101_{(2)}$
We need [7, 14], [15, 18] & [19, 19] (i.e. f(7, 3), f(15, 2) & f(19, 0))

Pseudocode

query(L, R)
  range = R - L + 1
  sum = 0
  for i = $\lfloor \log_2(\text{range}) \rfloor$ downto 0
    if i^{th} bit of range is 1
      sum += ST[L][i]
      L += 2^i
  return sum

Binary Lifting

In fact, the way we compute the sum with sparse table is called “binary lifting”

This technique is very useful in many different problems

One of them is lowest common ancestor (LCA)

Lowest Common Ancestor (LCA)

$f(u, i) = 2^i$-th ancestor of node u

$f(u, 0)$ = parent of u

For finding LCA of u & v, without loss of generality, assume $dep(v) \geq dep(u)$

First, we lift node v up such that $dep(v) = dep(u)$

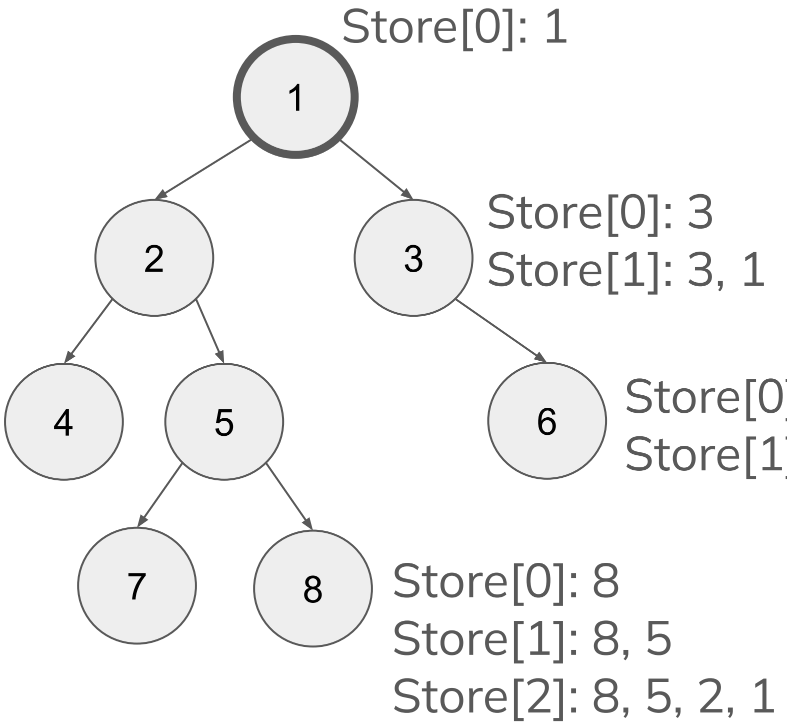
If $v = u$, then u is the LCA

Otherwise, for i from high to low, if $f(u, i) \neq f(v, i)$, we lift up node u and node v together, i.e. $u = f(u, i)$ & $v = f(v, i)$

At last, $f(u, 0)$ is the answer

More details: attend next week's Graph(III) training sessions, or refer to last year's slide

Sparse Table on Tree



Practice Problems

https://judge.hkoi.org/task/M2112

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

https://judge.hkoi.org/task/NP1313

https://judge.hkoi.org/task/T114

Further Readings

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

https://oi-wiki.org/ds/sparse-table/

https://oi-wiki.org/topic/rmq/

Segment Tree

Now we take a look at the range sum query problem with update

Given an array A of N integers and Q operations

Type 1: given x & val, update A[x] = val

Type 2: given L & R, query sum A[L..R]

Since sparse table doesn't support update, every type 1 operation requires recomputing the table, which is not efficient enough to pass the time limit in usual

Segment Tree

Segment Tree is a more flexible data structure for solving this problem

It is a binary tree

Every node store the information of an interval [L, R]

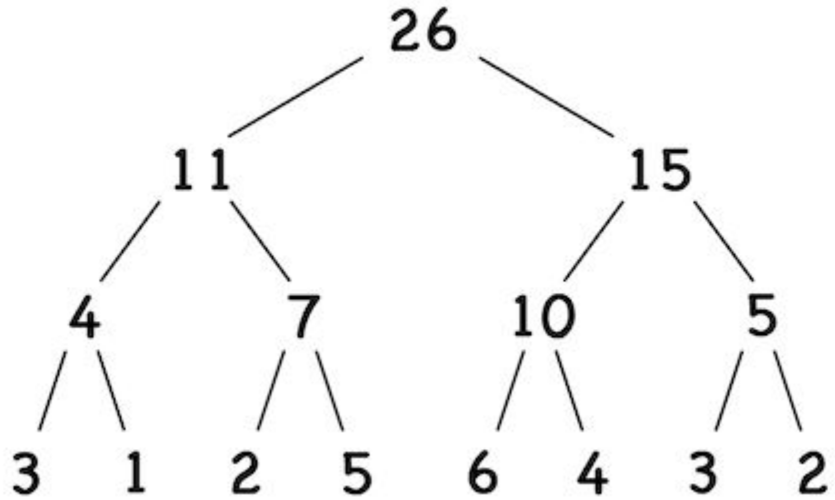
Let mid = L + (R - L) / 2

Its left child store the information of [L, mid]

Its right child store the information of [mid + 1, R]

Segment Tree

A = {3, 1, 2, 5, 6, 4, 3, 2}



Source: https://codeforces.com/edu/course/2/lesson/4

Segment Tree

First, let's walk through how to build the segment tree for an array A

It can be constructed using recursion

In general, if current node's id is x,

x * 2 is used as left child's id

x * 2 + 1 is used as right child's id

So we can compute the id directly and don't have to store them separately

Pseudocode

```
build(id, L, R)
  if L = R
    Node[id] ← A[L]
    return
  mid ← L + (R - L) / 2
  build(id * 2, L, mid)
  build(id * 2 + 1, mid + 1, R)
  Node[id] ← Node[id * 2] + Node[id * 2 + 1]

build(1, 1, N)
```

Segment Tree

For length of A = 2^x, total number of nodes = 2N - 1

For length of A ≠ 2^x, largest id can exceed 2N, so we usually declare an array of size 4N for the segment tree

Time & Space Complexity: O(N)

Segment Tree

Now, let's walk through how to query efficiently using segment tree

Again, it can be done by recursion

Pseudocode

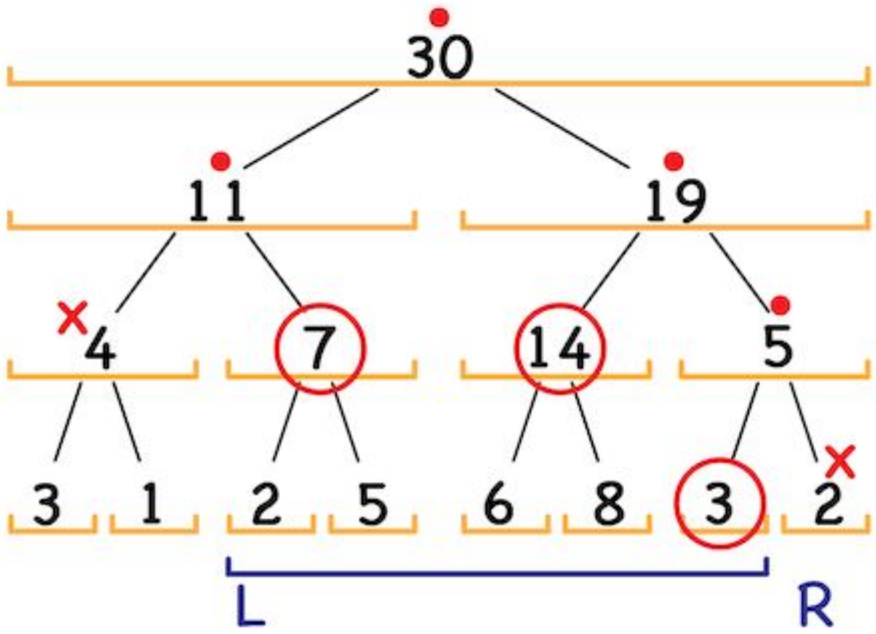
```
query(id, L, R, QL, QR) // range [L, R], query range [QL, QR]
  if QR < L or R < QL // no intersection between [L, R] & [QL, QR]
    return 0
  if QL ≤ L and R ≤ QR // [L, R] is fully inside [QL, QR]
    return Node[id]
  mid ← L + (R - L) / 2
  return query(id * 2, L, mid, QL, QR) + query(id * 2 + 1, mid + 1, R, QL, QR)

query(1, 1, N, QL, QR)
```

Segment Tree

The time complexity for query is O(log N)

Why? Because in each level of the segment tree, we will only use at most two nodes. Notice that for each 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 their parent instead.



Source: https://codeforces.com/edu/course/2/lesson/4

Segment Tree

Finally, let’s walk through how to update in the segment tree

Yes, we can still use recursion to do it

Pseudocode

```
update(id, L, R, x, val)
  if L = R
    Node[id] ← val
    return
  mid ← L + (R - L) / 2
  if x ≤ mid
    update(id * 2, L, mid, x, val)
  else
    update(id * 2 + 1, mid + 1, R, x, val)
  Node[id] ← Node[id * 2] + Node[id * 2 + 1]

update(1, 1, N, x, val)
```

Segment Tree

We solved range query with point update (also point query with range update)

For segment tree, the information stored in the node is much more flexible than sparse table, since there is no overlapped intervals

You can store prefix min / max, hash sum, dp table, etc., as long as the operation satisfies some properties of monoid

Practice Problems

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

Lazy Propagation

Now let's deal with both range update & range query

Given QL, QR, val, update A[i] += val for i = QL to QR

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

Lazy Propagation

We can solve this problem lazily

Instead of updating every index, we store the intermediate information in some intervals, which are exactly the intervals covered in range query

For each node, only when we need to access its children, we propagate the information to the children

That's why it's called "lazy propagation"

Pseudocode

```
push_down(id, L, mid, R)
  Node[id * 2] += lazy[id] * (mid - L + 1)
  Node[id * 2 + 1] += lazy[id] * (R - mid)
  lazy[id * 2] += lazy[id]
  lazy[id * 2 + 1] += lazy[id]
  lazy[id] = 0
```

lazy[id] is the value we have updated in current node id, but not in its children

Pseudocode

```
query(id, L, R, QL, QR) // range [L, R], query range [QL, QR]
  if QR < L or R < QL // no intersection between [L, R] & [QL, QR]
    return 0
  if QL ≤ L and R ≤ QR // [L, R] is fully inside [QL, QR]
    return Node[id]
  mid ← L + (R - L) / 2
  push_down(id, L, mid, R) // push down only when we need to access children
  return query(id * 2, L, mid, QL, QR) + query(id * 2 + 1, mid + 1, R, QL, QR)

query(1, 1, N, QL, QR)
```

Pseudocode

```
update(id, L, R, QL, QR, val) // range [L, R], update range [QL, QR]
  if QR < L or R < QL // no intersection between [L, R] & [QL, QR]
    return
  if QL ≤ L and R ≤ QR // [L, R] is fully inside [QL, QR]
    Node[id] += val * (R - L + 1) // update fully covered interval
    lazy[id] += val // store intermediate information in fully covered interval
    return
  mid ← L + (R - L) / 2
  push_down(id, L, mid, R) // push down only when we need to access children
  update(id * 2, L, mid, QL, QR, val)
  update(id * 2 + 1, mid + 1, R, QL, QR, val)
  Node[id] ← Node[id * 2] + Node[id * 2 + 1]

update(1, 1, N, QL, QR, val)
```

Practice Problems

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

https://judge.hkoi.org/task/T213

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

Further Readings

https://cp-algorithms.com/data_structures/segment_tree.html
https://oi-wiki.org/ds/seg/
https://oi-wiki.org/geometry/scanning/
https://codeforces.com/edu/course/2/lesson/4
https://codeforces.com/edu/course/2/lesson/5
https://codeforces.com/blog/entry/18051
https://www.luogu.com.cn/blog/cyffff/talk-about-segument-trees-split
https://www.luogu.com.cn/blog/foreverlasting/xian-duan-shu-fen-zhi-zong-jie
https://atcoder.github.io/ac-library/production/document_en/segtree.html
https://atcoder.github.io/ac-library/production/document_en/lazysegtree.html
https://codeforces.com/blog/entry/57319

Binary Indexed Tree (Fenwick Tree)

All the stuffs you can do with BIT can be done with segment tree

So what the advantages of BIT?

- Less code
- Less space
- Smaller constant factor

Binary Indexed Tree (Fenwick Tree)

As you can see in the agenda, it is described as “Prefix Sum with Efficient Update”

The reason why it is efficient is that it utilizes the binary representation of the id

Binary Indexed Tree (Fenwick Tree)

Let lowbit(x) be the value of the rightmost bit in binary representation of x

E.g. x = 22 = $10110_{(2)}$, lowbit(x) = $00010_{(2)}$ = 2

In BIT, node x stores the information of interval [x - lowbit(x) + 1, x]

How to compute lowbit(x) efficiently?

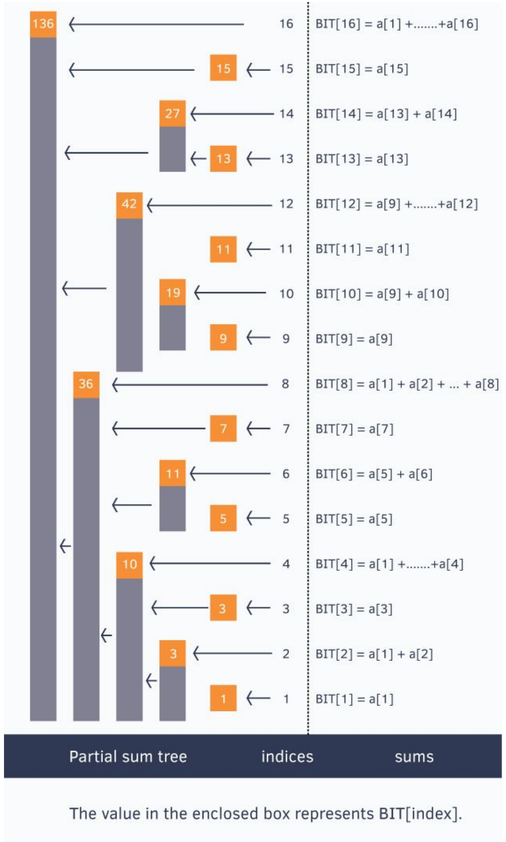
Answer: lowbit(x) = x & -x

Yes, O(1) operation

Binary Indexed Tree (Fenwick Tree)

Let’s solve the range sum query problem again with BIT this time

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



Pseudocode

```
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 segment tree / BIT into a 2D data structure, where each node is another segment tree / BIT

Pseudocode

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

```
sum(x, 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 structure 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 1D data structure, 2D data structure is not that difficult to implement, although sometime it is quite tedious

Practice Problem

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

Further Readings

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

https://oi-wiki.org/ds/fenwick/

https://www.luogu.com.cn/blog/kingxbz/shu-zhuang-shuo-zu-zong-ru-men-dao-ru-fen

https://www.luogu.com.cn/blog/countercurrent-time/qian-tan-shu-zhuang-shuo-zu-you-hua