セグメント木， 遅延評価セグメント木を理解する

BV21013 堀毛晴輝

May 22, 2022

芝浦工業大学 数理科学研究会

研究背景

研究背景

競技プログラミングで戦うために必要なデータ構造のなかに， セグメント木と遅延評価セグメント木がある．原理・実装を知り， 理解することで実際に必要な場面で使いこなすことができる．

これらを使いこなすことができる高度 IT 人材となるべく， 今回の研究の題材とした．

セグメント木

セグメント木とは

セグメント木とは， モノイドを満たす代数構造に対し使用できるデータ構造である．

モノイド

集合 S とその上の二項演算 $\cdot : S \times S \to S$ に対し，

- 結合律：任意の $a, b, c \in S$ に対して， $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- 単位元の存在：ある $e \in S$ が存在し， 任意の $a \in S$ に対して $e \cdot a = a \cdot e = a$

を満たすならば， 組 $(S, \cdot, e)$ をモノイドという．

セグメント木とは

セグメント木とは， モノイドを満たす代数構造に対し使用できるデータ構造である．

モノイド

集合 S とその上の二項演算 $\cdot : S \times S \to S$ に対し，

- 結合律：任意の $a, b, c \in S$ に対して， $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- 単位元の存在：ある $e \in S$ が存在し， 任意の $a \in S$ に対して $e \cdot a = a \cdot e = a$

を満たすならば， 組 $(S, \cdot, e)$ をモノイドという．

長さ n の S の配列に対して，

- 要素の 1 点変更
- 区間の要素の総積の取得

の操作を高速に (具体的にはともに $O(\log n)$ で) 処理できる．

たとえば · · ·

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：i, x が与えられる． a_i を x に書き換える
- クエリ 2：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力

たとえば · · ·

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：i, x が与えられる． a_i を x に書き換える
- クエリ 2：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力

愚直解

長さ n の配列を用意し， クエリ 1 では対応する箇所を書き換え， クエリ 2 では $a_l, a_{l+1}, \cdots, a_r$ を順に見ていき最小値を求める．

たとえば · · ·

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1 : i, x が与えられる． a_i を x に書き換える
- クエリ 2 : l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力

愚直解

長さ n の配列を用意し， クエリ 1 では対応する箇所を書き換え， クエリ 2 では $a_l, a_{l+1}, \cdots, a_r$ を順に見ていき最小値を求める．

愚直解だと最悪時の計算量が $\Theta(NQ)$ になってしまうが， セグメント木を使うと $O(Q \log N)$ で解ける．(各命令にかかる時間が 10^{-8} 秒とすると， 100 秒くらいかかる処理が 0.02 秒くらいで終わるようになる．)

原理

完全二分木

二分木であり葉以外の節天がすべて 2 つの子を持ち， 根から葉までの深さが等しい木構造を完全二分木という．深さが 1 だけ異なる葉が存在し， その葉が木全体の左側に詰めてあるような二分木も (おおよそ) 完全二分木という．

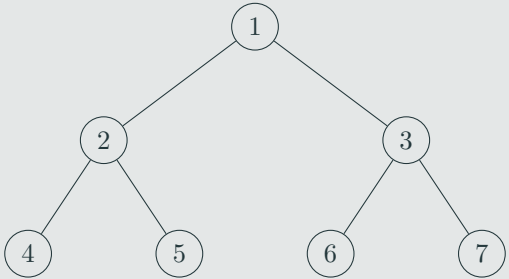


図 1: 完全二分木

原理

根の添え字を 1 とし， 幅優先探索で訪れる順に添え字を振る．

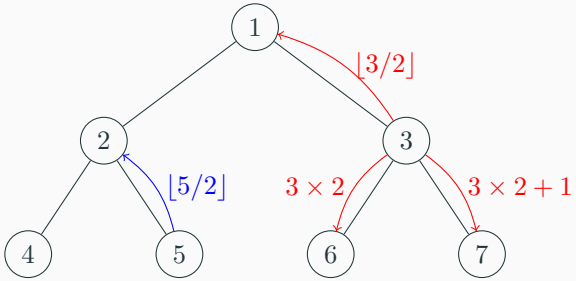


図 2: 親, 右/左の子へのアクセス

節点 i に対して， 親は $\lfloor i/2 \rfloor$ で， 左の子は $i \times 2$, 右の子は $i \times 2 + 1$ でアクセスできる．

原理

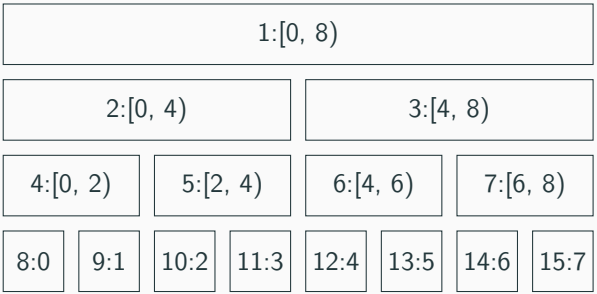


図 3: 完全二分木が管理する区間

親子関係は図 1 と同じである． 各節点はその範囲の積を管理する． たとえば， 節点 5 は $a_2 \cdot a_3$ を， 節点 3 は $a_4 \cdot a_5 \cdot a_6 \cdot a_7$ の値となるようにする．

更新クエリ

図 3 において， たとえば a_2 を更新するとき．..

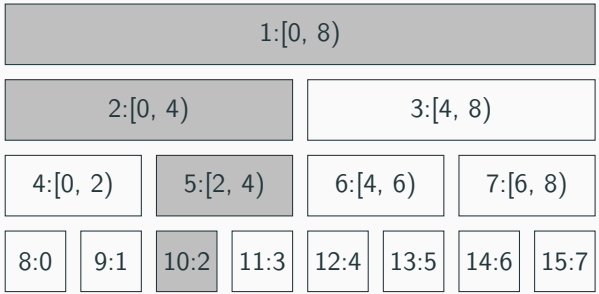


図 4: a_2 を更新するときに更新する必要がある節点

更新する必要がある節点は $\log n$ 個なので， 計算量は $O(\log n)$.

区間積クエリ

図 3 において， たとえば $[1, 6)$ の範囲の積を求めるとき．..

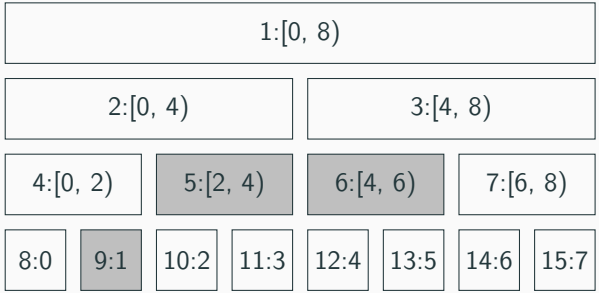


図 5: 区間 $[1, 6)$ の積を求める

色を付けた節点の積を取ることで求めることができる．

区間積クエリ

どの節点の値の積を取るか？ (計算回数をできるだけ減らしたいので， 浅い節点の積で表したい．..)

左端

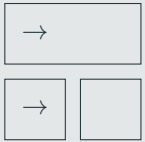


図 6: 左にいるとき

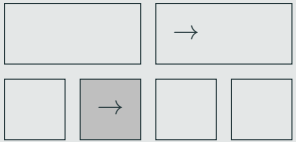


図 7: 右にいるとき

左端が左にいるとき (図 6), その親の節点を左端にしても問題ないのでなにもせずに左端を親に更新する．左端が右にいるとき (図 7), その親の節点では範囲外の値が含まれてしまうので， 左端の値を積として取ってから左端をひとつ右の親に更新する．

区間積クエリ

右端

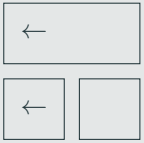


図 8: 左にいるとき

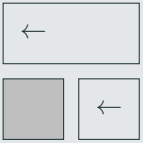
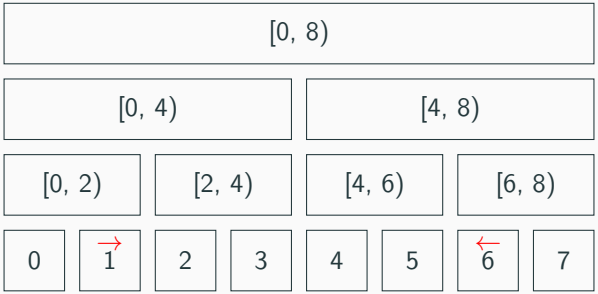


図 9: 右にいるとき

右端が左にいるとき (図 8), その親の節点を右端にしても問題ないので何もせずに右端を親に更新する．右端が右にいるとき (図 9), その親の節点では範囲内の値を含めることができないので， 右端の左の値を積として取ってから右端を親に更新する．

$[L, R)$ について， l, r を L, R に対応する葉とする． この処理を $l < r$ を満たさなくなるまで続ければ $[L, R)$ の区間の積をもとめることができる．

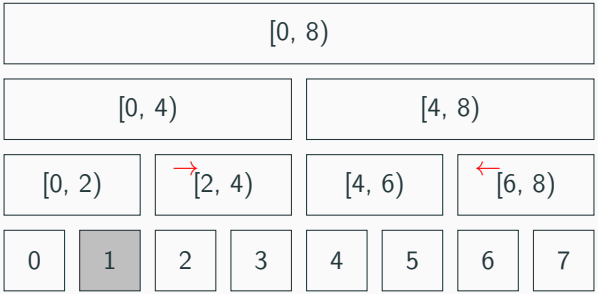
$[1, 6)$ の範囲の積を求めるとき．．．



左の積：e

右の積：e

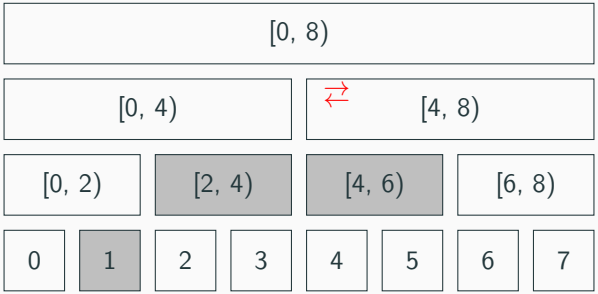
$[1, 6)$ の範囲の積を求めるとき．．．



左の積：a_1

右の積：e

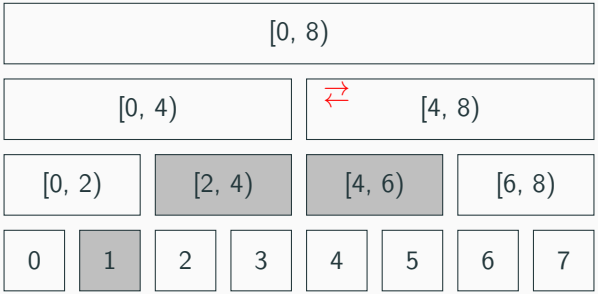
$[1, 6)$ の範囲の積を求めるとき．．．



左の積：$a_1 \cdot (a_2 \cdot a_3)$

右の積：$(a_4 \cdot a_5)$

$[1, 6)$ の範囲の積を求めるとき．．．



左の積： $a_1 \cdot (a_2 \cdot a_3)$

右の積： $(a_4 \cdot a_5)$

$[1, 6)$ の範囲の積は， これらの積をとり $(a_1 \cdot (a_2 \cdot a_3)) \cdot (a_4 \cdot a_5)$ で求めることができる．

最大でも上る回数は $\log n$ 回なので， 計算量は $O(\log n)$.

実装

区間の積の取得の実装のみ触れる． 実際のコードは資料を参照．

```
1     // S : モノイドの型       S op(S a, S b) : 二項演算
2     // S e() : 単位元
3     // vector<S> d : 完全二分木各節点に区間の値が格納されている.
4     // int size : 葉の数. にを足すとの値が格納されている葉の添え字となる
          isizei.
5     S prod(int l, int r) { // [l, r)の範囲の積を求める
6         S sml = e(), smr = e();
7         l += size; r += size;
8         while(l < r) {
9             if(l & 1) sml = op(sml, d[l++]);
10            if(r & 1) smr = op(d[--r], smr);
11            l >>= 1; r >>= 1;
12        }
13        return op(sml, smr);
14    }
```

実装

- 7 行目で l, r をその値が格納されている葉の添え字に更新．
- 節点 i が左の子か右の子か， は下位 1bit を見ることでわかる． 右の子であったときに sml/smr を積を取ったものに更新．
- 11 行目で 1bit 右にシフトすることで親の添え字に更新している．
- sml/smr と左右で分けて値を持っておき， 最後にそれらの積を返すことにより演算が可換でなくてもよい実装となっている．たとえば， 行列を乗せることも可能．

RMQ(Range Minimum Query)

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：i, x が与えられる． a_i を x に書き換える
- クエリ 2：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力

RMQ(Range Minimum Query)

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：i, x が与えられる． a_i を x に書き換える
- クエリ 2：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力

解

集合 $S = \{x|\ -10^9 \leq x \leq 10^9$ または $x = \infty\}$ とする．
モノイド $(S, \cdot : (a, b) \mapsto \min(a, b), \infty)$ をセグメント木に載せることで $O(Q \log N)$ で解くことができる．

詳しい実装は資料を参照．

遅延評価セグメント木

遅延評価セグメント木とは

モノイド S と， S への作用素モノイド F に対し使用できるデータ構造である．

右作用/作用素モノイド

モノイド G と集合 X に対し， X への G による右作用とは， 写像

$$\psi : G \times X \to X,\ (g, x) \mapsto \psi(g, x)$$

であって， 次の 2 条件を満たすもののことをいう．

(1). 任意の $x \in X$ に対し， $\psi(e, x) = x$. ただし， e は G の単位元．

(2). 任意の $g, h \in G, x \in X$ に対し， $\psi(gh, x) = \psi(h, \psi(g, x))$

また， 作用に関するモノイドを作用素モノイドという．

また， $x_i, x_j \in S, f \in F$ に対して， $\psi(f, x_i \cdot x_j) = \psi(f, x_i) \cdot \psi(f, x_j)$ が成り立つ必要がある．

遅延評価セグメント木

長さ n の S の配列に対して，

- 要素の 1 点変更
- 区間の要素の総積の取得
- 区間に $f \in F$ を作用させる

の操作を高速に (具体的にはすべて $O(\log n)$ で) 処理できる．

たとえば...

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力
- クエリ 2：l, r, x が与えられる． $a_l, a_{l+1}, \cdots, a_r$ すべてを x に更新する．

たとえば...

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力
- クエリ 2：l, r, x が与えられる． $a_l, a_{l+1}, \cdots, a_r$ すべてを x に更新する．

愚直解

長さ n の配列を用意し， クエリ 1 では $a_l, a_{l+1}, \cdots, a_r$ を順に見ていき最小値を求め， クエリ 2 では $a_l, a_{l+1}, \cdots, a_r$ を順に更新していく．

たとえば...

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力
- クエリ 2：l, r, x が与えられる． $a_l, a_{l+1}, \cdots, a_r$ すべてを x に更新する．

愚直解

長さ n の配列を用意し， クエリ 1 では $a_l, a_{l+1}, \cdots, a_r$ を順に見ていき最小値を求め， クエリ 2 では $a_l, a_{l+1}, \cdots, a_r$ を順に更新していく．

愚直解だと最悪時の計算量が $\Theta(NQ)$ になってしまうが， 遅延評価セグメント木を使うと $O(Q \log N)$ で解ける．(各命令にかかる時間が 10^{-8} 秒とすると， 100 秒くらいかかる処理が 0.02 秒くらいで終わるようになる．)

原理

以下， x_i, y_i を S の元， f, g, h を F の元， id を F の単位元とする．

各モノイドの二項演算を，

$$S \times S \to S;\ (x_i, x_j) \mapsto x_i x_j$$

$$F \times F \to F;\ (f, g) \mapsto fg$$

と書き， 作用は，

$$S \times F \to S;\ (x_i, f) \mapsto x_i^f$$

と書く．

遅延評価セグメント木では， セグメント木と同様に S を管理する完全二分木と， その区間への作用を管理する完全二分木の 2 つを用いて処理を行う．

具体例とともに， どのような手順を踏んでいるかを見る．

初期状態

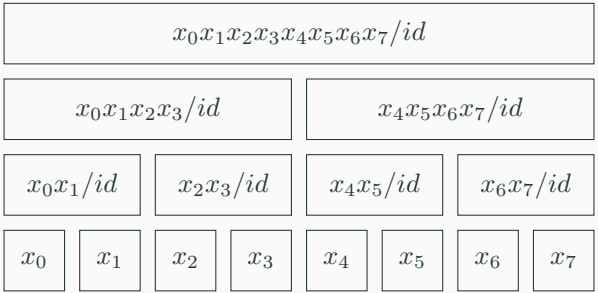


図 10: 初期状態

I. $[1, 6)$ に f を作用させる.

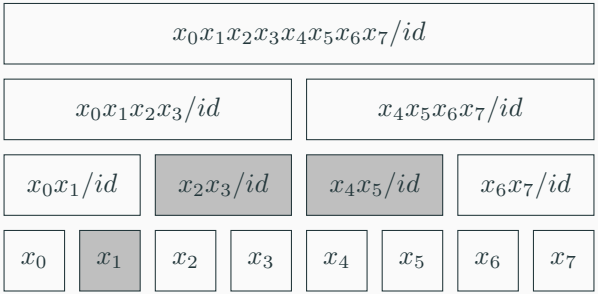


図 11: $[1, 6)$ に f を作用

I. $[1, 6)$ に f を作用させる.

- 上から伝搬

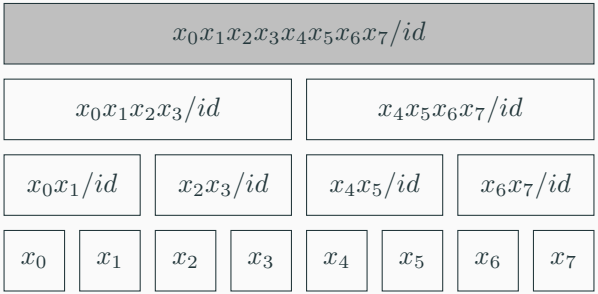


図 11: $[1, 6)$ に f を作用

I. $[1,6)$ に f を作用させる.

- 上から伝搬

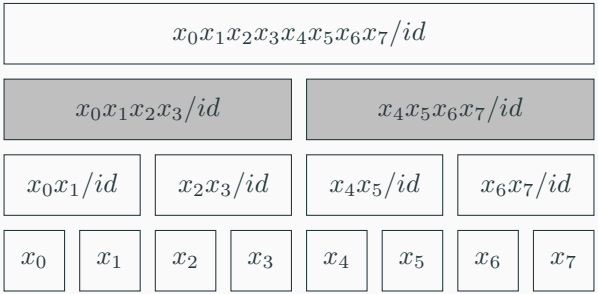


図 11: $[1,6)$ に f を作用

I. $[1, 6)$ に f を作用させる.

- 上から伝搬

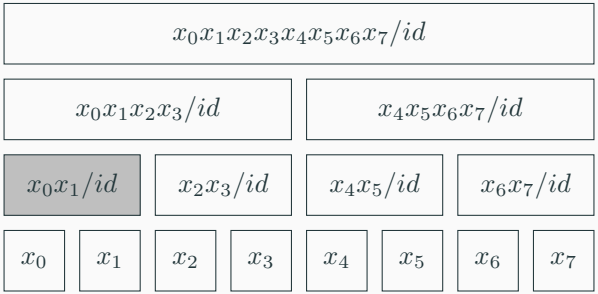


図 11: $[1, 6)$ に f を作用

I. $[1, 6)$ に f を作用させる.

- 上から伝搬
- $[1, 6)$ に f を作用

図 11: $[1, 6)$ に f を作用

I. $[1,6)$ に f を作用させる.

- 上から伝搬
- $[1,6)$ に f を作用

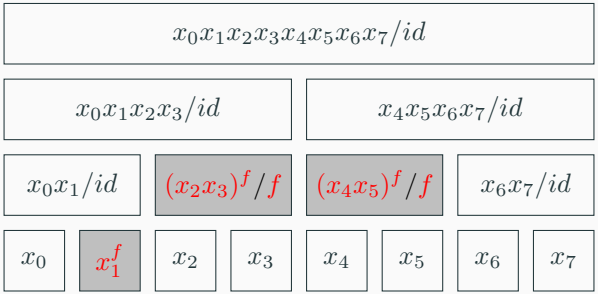


図 11: $[1,6)$ に f を作用

I. $[1,6)$ に f を作用させる.

- 上から伝搬
- $[1,6)$ に f を作用
- 上側を計算しなおし

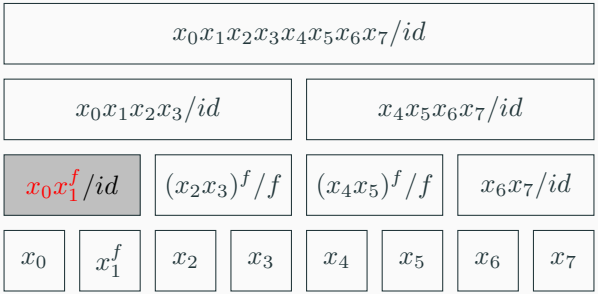


図 11: $[1,6)$ に f を作用

I. $[1,6)$ に f を作用させる.

- 上から伝搬
- $[1,6)$ に f を作用
- 上側を計算しなおし

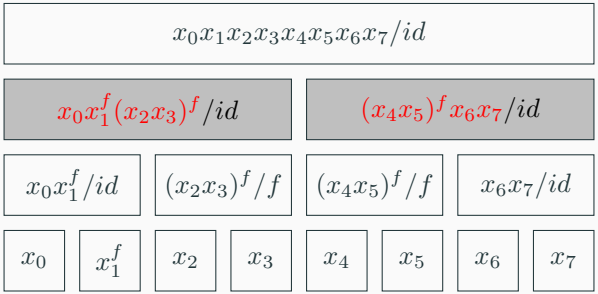


図 11: $[1,6)$ に f を作用

I. $[1,6)$ に f を作用させる.

- 上から伝搬
- $[1,6)$ に f を作用
- 上側を計算しなおし

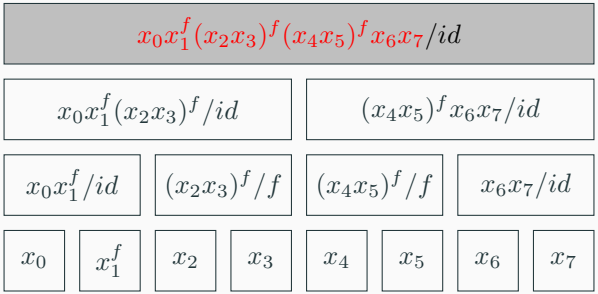


図 11: $[1,6)$ に f を作用

II. $[3,8)$ に g を作用させる.

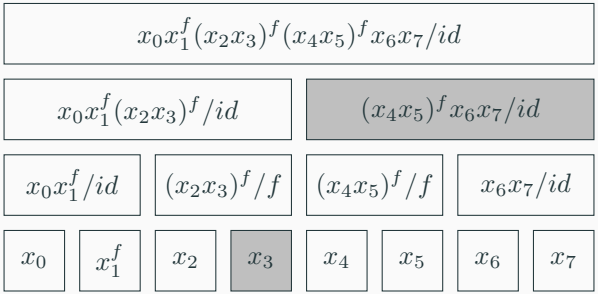


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬

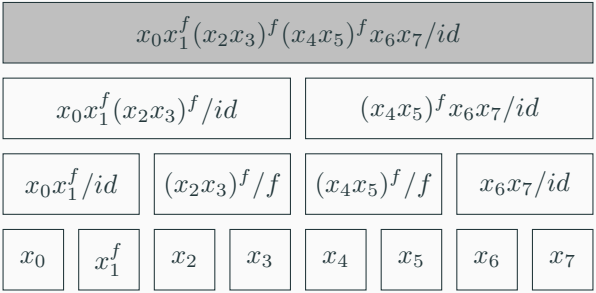


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬

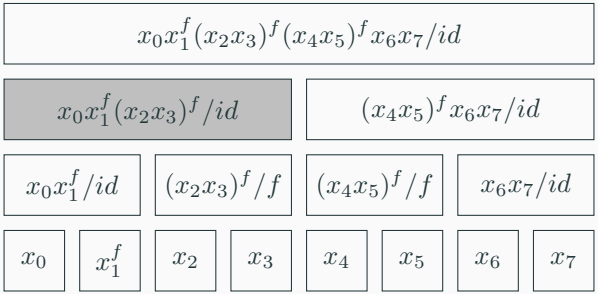


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬

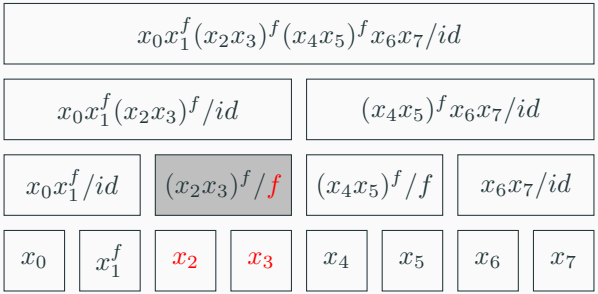


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬

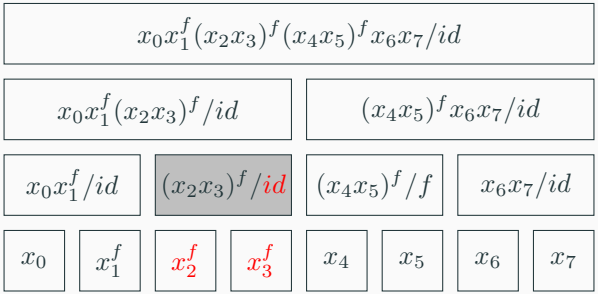


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬
- $[3,8)$ に g を作用

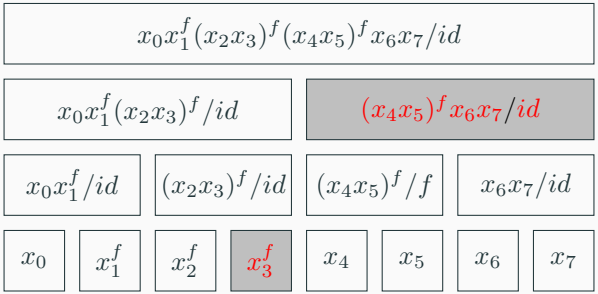


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬
- $[3,8)$ に g を作用

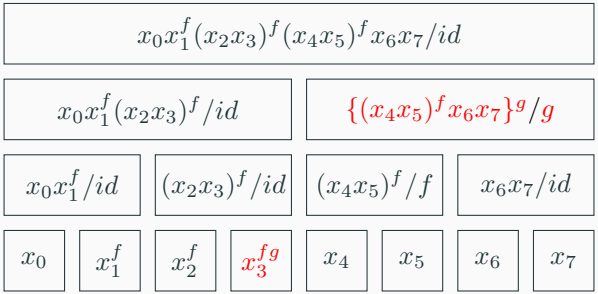


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬
- $[3,8)$ に g を作用
- 上側を計算しなおし

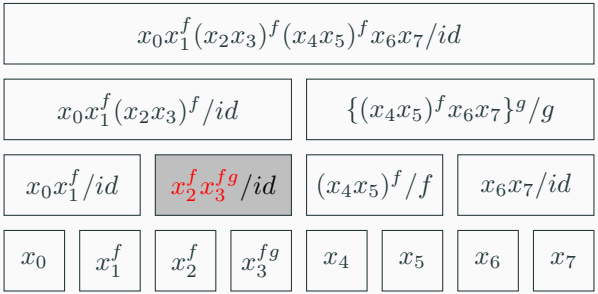


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬
- $[3,8)$ に g を作用
- 上側を計算しなおし

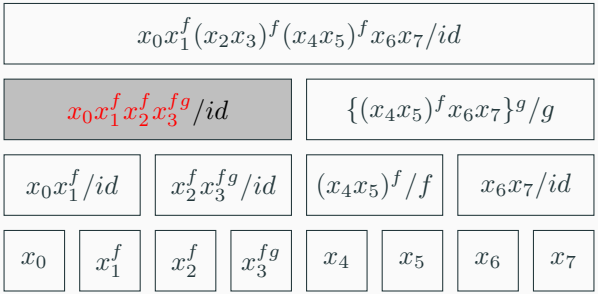


図 12: $[3,8)$ に g を作用

II. $[3,8)$ に g を作用させる.

- 上から伝搬
- $[3,8)$ に g を作用
- 上側を計算しなおし

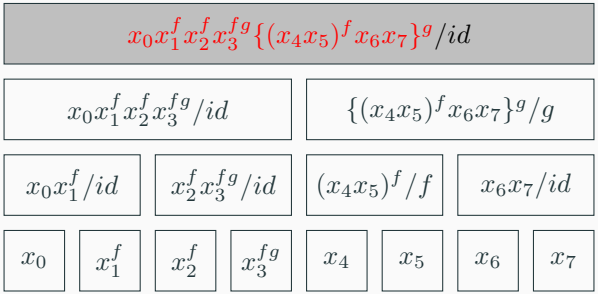


図 12: $[3,8)$ に g を作用

III. $[4, 7)$ を取得する.

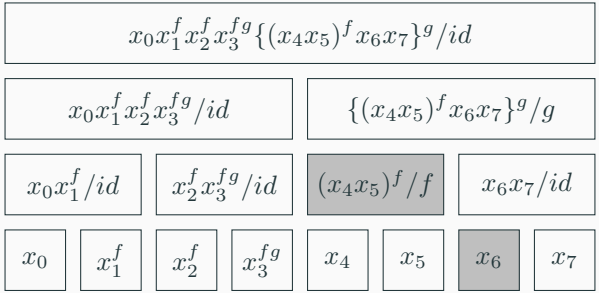


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬

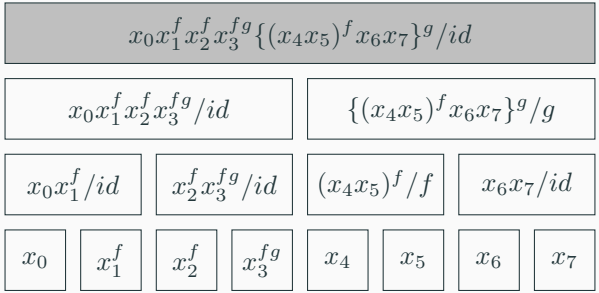


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬

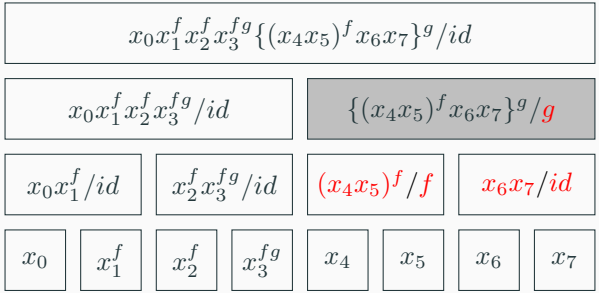


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬

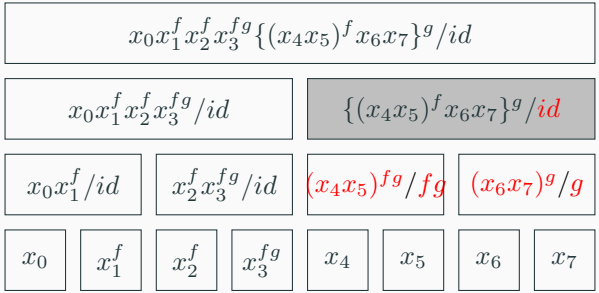


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬

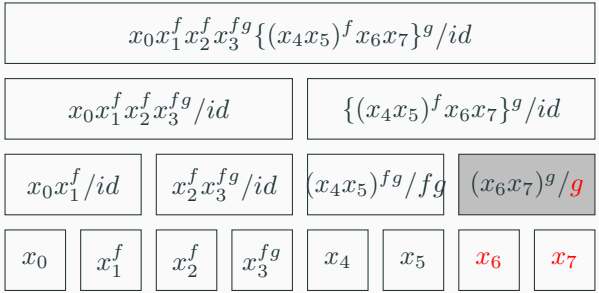


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬

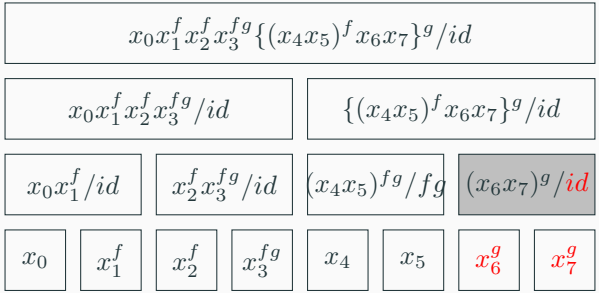


図 13: $[4, 7)$ を取得

III. $[4, 7)$ を取得する.

- 上から伝搬
- $[4, 7)$ を取得

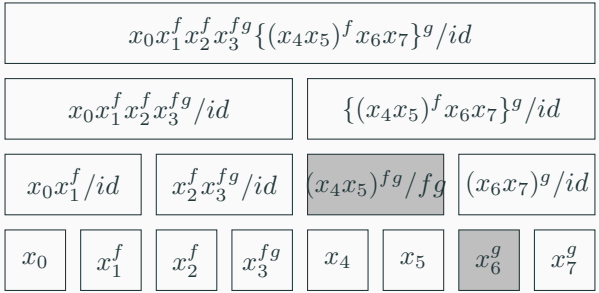


図 13: $[4, 7)$ を取得

$(x_4 x_5)^{fg} x_6^g$ が得られる.

IV. x_5 を y_5 に更新する.

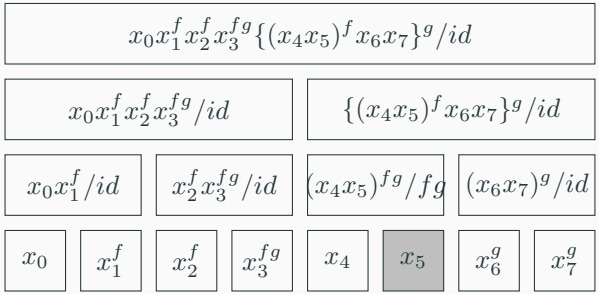


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬

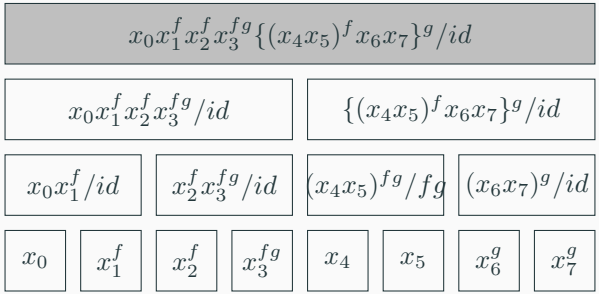


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬

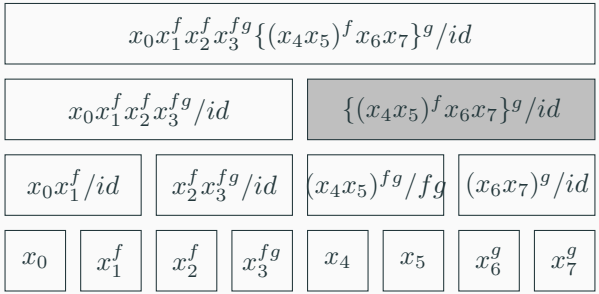


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬

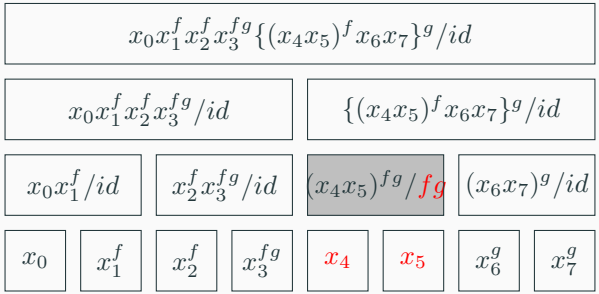


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬

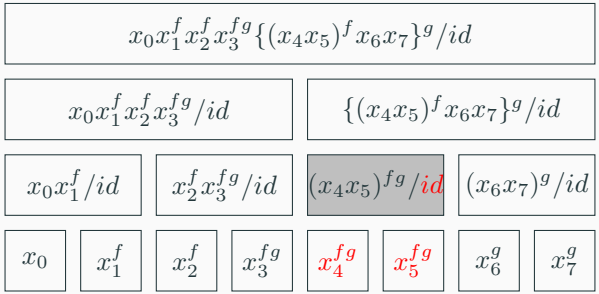


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬
- x_5 を更新 y_5 に更新する

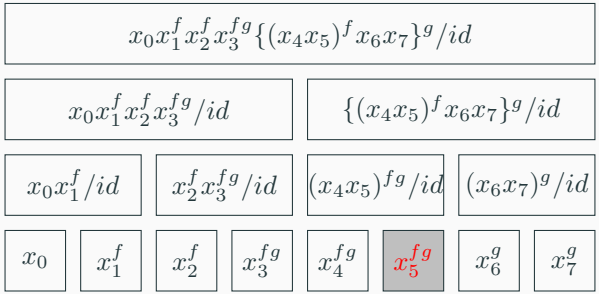


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬
- x_5 を更新 y_5 に更新する

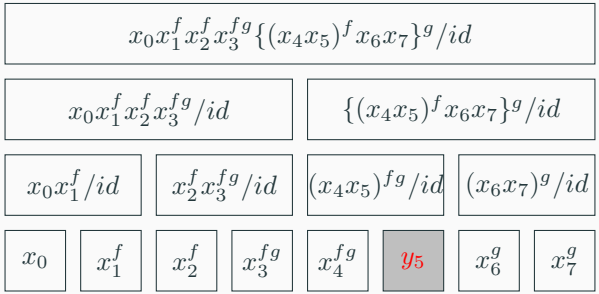


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬
- x_5 を更新 y_5 に更新する
- 上側を計算しなおし

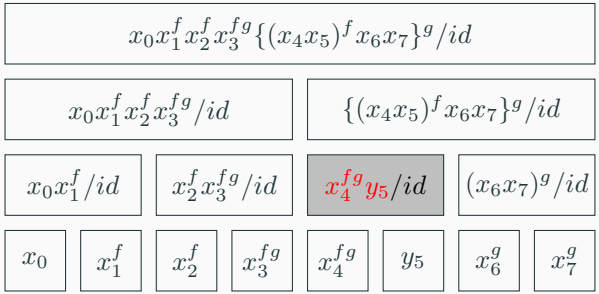


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬
- x_5 を更新 y_5 に更新する
- 上側を計算しなおし

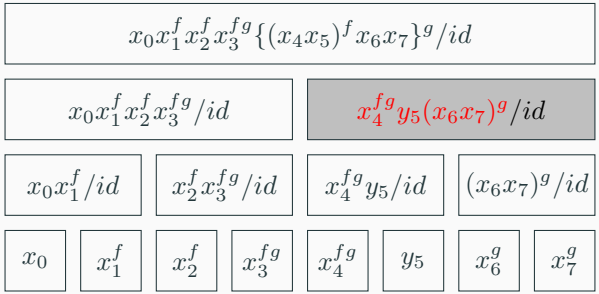


図 14: x_5 を y_5 に更新

IV. x_5 を y_5 に更新する.

- 上から伝搬
- x_5 を更新 y_5 に更新する
- 上側を計算しなおし

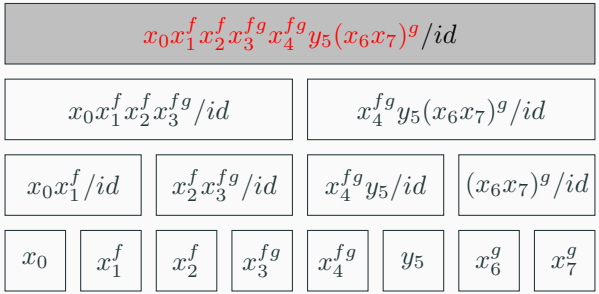


図 14: x_5 を y_5 に更新

原理

- 区間作用
  1. 上から下へ作用を伝搬
  2. 区間に作用
  3. 下から上へ更新
- 区間作用
  1. 上から下へ作用を伝搬
  2. 区間を取得
- 1 点更新
  1. 上から下へ作用を伝搬
  2. 更新
  3. 下から上へ更新

を順に行うことでクエリを処理できる.

実装

作用の伝搬の実装のみ触れる． 実際のコードは資料を参照．

```
1     // void push(int k) : の節点に作用している値をの子へ伝搬 kk
2     // [l, r] の区間に対して, 上から下へ作用を伝搬
3     l += size; r += size;
4     for(int i = log; i >= 1; i--) {
5         if(((l >> i) << i) != l) push(l >> i);
6         if(((r >> i) << i) != r) push(r >> i);
7     }
```

`if(((l >> i) << i) != l), if(((r >> i) << i) != r)`

がどのようなことをしているのかを見ていく．

実装

たとえば， $[2, 6)$ の区間取得をおこなうとき，

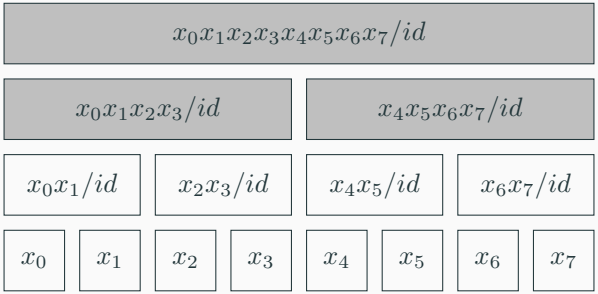


図 15: [2,6) の区間取得を行うときの伝搬を行わないといけない節点

x_2x_3 の節点と x_4x_5 の節点の二項演算を返すため， 伝搬は色をつけた節点でのみ行えばよい．つまり， x_2x_3 と x_6x_7 の節点では伝搬をする必要はない．どのような場合に伝搬をする必要がないかを考える．

実装

$[l, r)$ の区間について，

- l の真上にあり， 伝搬をする必要がない節点とは， その節点が持つ範囲の左端が l であるような節点である．
- r の真上にあり， 伝搬をする必要がない節点とは， その節点が持つ範囲の左端が r であるような節点である．(このとき， r より左を見たときに必要な部分までの伝搬がすべて完了する)

葉 x の真上にある節点は， i=0,1,$\cdots$,log に対して `(x >> i)` で表される． それらの節点の左端は `(x >> i) << i` であるため， これが x と一致する場合はその節点では伝搬をおこなわなくてよい．よって， `if(((l >> i) << i) != l)`, `if(((r >> i) << i) != r)` の時のみ `l >> i`, `r >> i` の節点で伝搬すればよい．

RMQ and RUQ (Range Minimum Query and Range Update Query)

配列 $A = a_0, a_1, \cdots a_{N-1}$ が与えられる． 次のクエリを Q 回処理せよ．

制約：
$1 \leq N \leq 10^5,\ 1 \leq Q \leq 10^5,\ 0 \leq l \leq r < N,\ -10^9 \leq a_i \leq 10^9\ (0 \leq i < N),\ -10^9 \leq x \leq 10^9$

クエリ

- クエリ 1：l, r が与えられる． $a_l, a_{l+1}, \cdots, a_r$ の最小値を出力
- クエリ 2：l, r, x が与えられる． $a_l, a_{l+1}, \cdots, a_r$ すべてを x に更新する．

RMQ and RUQ (Range Minimum Query and Range Update Query)

解

集合 $S = \{x|\ -10^9 \leq x \leq 10^9$ または $x = \infty\}, F = \{f|\ -10^9 \leq f \leq 10^9$ または $f = \infty\}$ とする．モノイド $(S, \cdot : (a,b) \mapsto \min(a,b), \infty)$, モノイド $(F, \times, \infty)$

$$\times : (f,g) \mapsto \begin{cases} g & \text{if } g \neq \infty \\ f & \text{else.} \end{cases}, \quad \psi(f,x) = \begin{cases} f & \text{if } f \neq \infty \\ x & \text{else.} \end{cases}$$

を遅延評価セグメント木に乗せることで $O(Q \log N)$ で解くことができる．

詳しい実装は資料を参照．

応用

LCA

LCA(Lowest Common Ancestor, 最近共通祖先)

根付き木において， 頂点 a と b を根に向かってたどっていったとき， 最初に合流する頂点を a, b の LCA(Lowest Common Ancestor, 最近共通祖先) という．

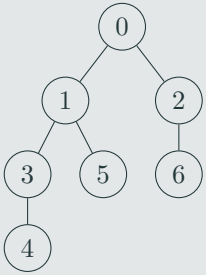


図 16: 0 を根とする根付き木の例

たとえば， この 0 を根とする根付き木に対して，

- 4, 5 の LCA は 1
- 5, 6 の LCA は 0
- 2, 6 の LCA は 2

である．

オイラーツアー

根から DFS し根に戻ってくる経路を， オイラーツアーという．

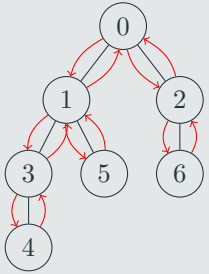


図 17: オイラーツアーの例

たとえば， この 0 を根とする根付き木に対して， オイラーツアーの経路の 1 つは，$0 \to 1 \to 3 \to 4 \to 3 \to 1 \to 5 \to 1 \to 0 \to 2 \to 6 \to 2 \to 0$ である．

LCA

オイラーツアーをする際に， 各頂点にて訪れたときのステップ数を記録しておく． 図 17 の例では，

ステップ	0	1	2	3	4	5	6	7	8	9	10	11	12
経路	0	1	3	4	3	1	5	1	0	2	6	2	0

表 1: オイラーツアーの経路

頂点	0	1	2	3	4	5	6
in	0	1	9	2	3	6	10

表 2: 各頂点に初めて訪れた時のステップ数

となる．

LCA

オイラーツアーの経路に対して， 区間 [min(in[a], in[b]), max(in[a], in[b])] を考える． たとえば区間 [in[4], in[5]] は次のようになる．

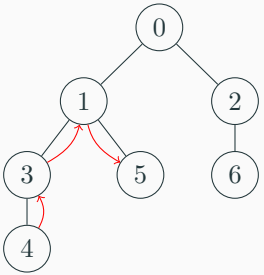


図 18: 区間 [in[4], in[5]]

[in[4], in[5]] は， $4 \to 3 \to 1 \to 5$ となる． この区間の中で深さが最小である頂点， すなわち頂点 1 が頂点 4, 5 の LCA となる．

LCA

区間 [in[3], in[6]] では， 次のようになる．

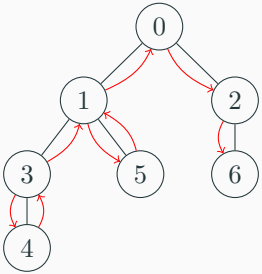


図 19: 区間 [in[3], in[6]]

$3 \to 4 \to 3 \to 1 \to 5 \to 1 \to 0 \to 2 \to 6$ となる． この中で深さが最小である頂点は頂点 0 であり， 頂点 3, 6 の LCA と一致する．

LCA

任意の 2 頂点間の LCA

頂点 a, b の LCA は， オイラーツアーした経路の区間 [min(in[a], in[b]), max(in[a], in[b])] の深さが最小の節点となる．区間の最小値を求める問題に言い換えることができて， これはセグメント木を用いることで $O(\log N)$ (N は頂点の数) で求めることができる．したがって， 任意の 2 頂点間の LCA を $O(\log N)$ で求めることができた．

LCA

任意の 2 頂点間の LCA

頂点 a, b の LCA は，オイラーツアーした経路の区間 [min(in[a], in[b]), max(in[a], in[b])] の深さが最小の節点となる．区間の最小値を求める問題に言い換えることができて，これはセグメント木を用いることで $O(\log N)$ (N は頂点の数) で求めることができる．したがって，任意の 2 頂点間の LCA を $O(\log N)$ で求めることができた．

任意の 2 頂点間の距離

根から頂点 i までの距離を $dist_i$ と書くことにする．頂点 a から b の距離は，

$$dist_a + dist_b - 2dist_{LCA(a,b)}$$

で求めることができる．

LCA

たとえば， 図 17 の例で頂点 4 から頂点 5 への距離を考える．

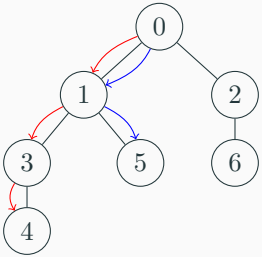


図 20: 頂点 4 から頂点 5 への距離

$dist_4$ を赤， $dist_5$ を青で示した． LCA(4, 5) $= 1$ である． LCA の定義より， $dist_4$, $dist_5$ は $dist_{LCA(4,5)}$ まで同じ経路をたどる．よって， 頂点 4 から頂点 5 への距離は，

$$dist_4 + dist_5 - 2dist_{LCA(4,5)} = 3 + 2 - 2 \cdot 1 = 3$$

と求めることができる． 実装は資料を参照．

おわりに

今後の課題

セグメント木上の二分探索には今回触れなかったため， 木上の二分探索についても今後学んでいきたい．

セグメント木・遅延評価セグメント木にのみ触れたが， このほかにも双対セグメント木や，Dinamic Segtree, Segtree Beats といったものもあるので， それらについても理解を深めていきたい．

参考文献

- AtCoder, "AtCoder Library Document", https://atcoder.github.io/ac-library/document_ja/index.html
- えびちゃん，"非再帰セグ木サイコー！　一番好きなセグ木です", https://hcpc-hokudai.github.io/archive/structure_segtree_001.pdf
- keymoon, "SegmentTree に載る代数的構造について", https://qiita.com/keymoon/items/0f929a19ed30f34ae6e8
- maspy, "SegmentTree のお勉強", https://maspypy.com/segment-tree-%E3%81%AE%E3%81%8A%E5%8B%89%E5%BC%B72
- maspy, "Euler Tour のお勉強", https://maspypy.com/euler-tour-%E3%81%AE%E3%81%8A%E5%8B%89%E5%BC%B7

ご清聴ありがとうございました.