セグ木・BIT

完全理解ガイド

アルゴリズム × C++ 実装

目次

Part 1

なぜ必要か？

問題設定

配列 `a[0..n-1]` に対して **Q 回**の操作を繰り返す：

操作 1：点更新

```
a[i] = x
```

（特定の要素を変更する）

操作 2：区間クエリ

```
query(l, r) = a[l] ○ a[l+1] ○ ... ○ a[r]
```

`○` は任意の演算（min, sum, gcd 等）

> **具体例:** 長さ $N=10^5$ の配列に $Q=10^5$ 回クエリ
> ナイーブだと毎回 O(N) → **計 10^{10} 回 = TLE 確定**

ナイーブ実装の限界

```
// 点更新: O(1) ← 速い
a[i] = x;

// 区間クエリ: O(n) ← 遅い!!
int result = INF;
for (int j = l; j <= r; j++) result = min(result, a[j]);
```

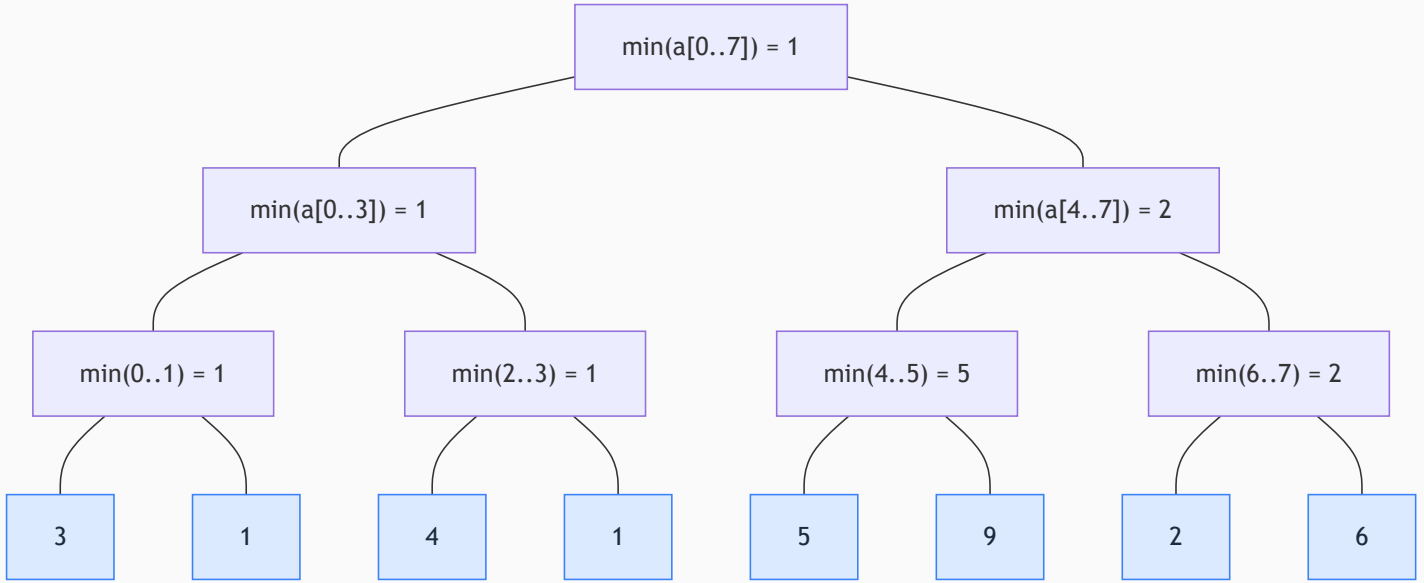
操作	ナイーブ	目標
点更新	**O(1)**	O(log n)
区間クエリ	**O(n)**	O(log n)
Q 回合計	**O(Qn)**	**O(Q log n)**

> N = Q = 10^5 のとき：**10^{10} 回** → **$10^5 \times 17 \approx 1.7 \times 10^6$ 回**（6000倍高速）

解決のキーアイデア

核心: あらかじめ部分区間の結果を計算しておき、使い回す

配列: `[3, 1, 4, 1, 5, 9, 2, 6]`



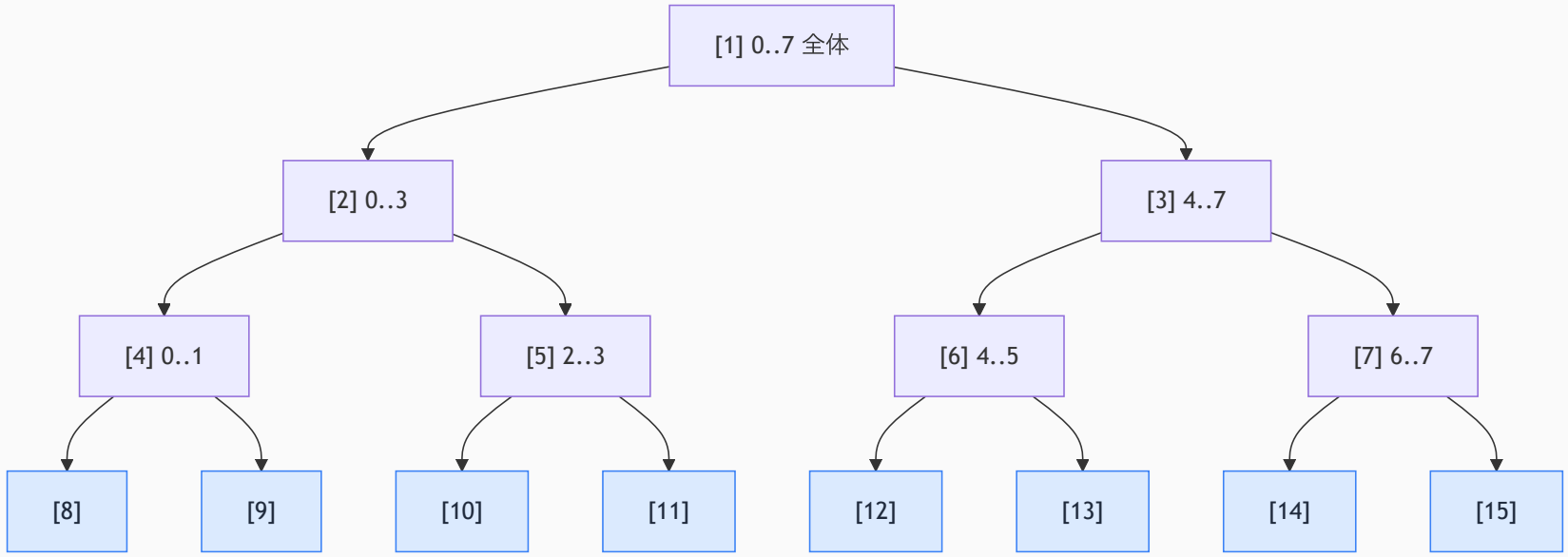
- **min(a[2..5])** が欲しい → `min(2..3)=1` と `min(4..5)=5` を合わせて → **1**
- 8要素全部見ずに **2ノード** で完了（O(log n)）
- 値を変更しても、影響するのは **根までの3ノード** だけ（O(log n)）

Part 2

セグメント木

セグメント木の構造

完全二分木の **配列上の表現**（内部ノード 1 始まり / `update`・`query` の引数は 0-indexed）：

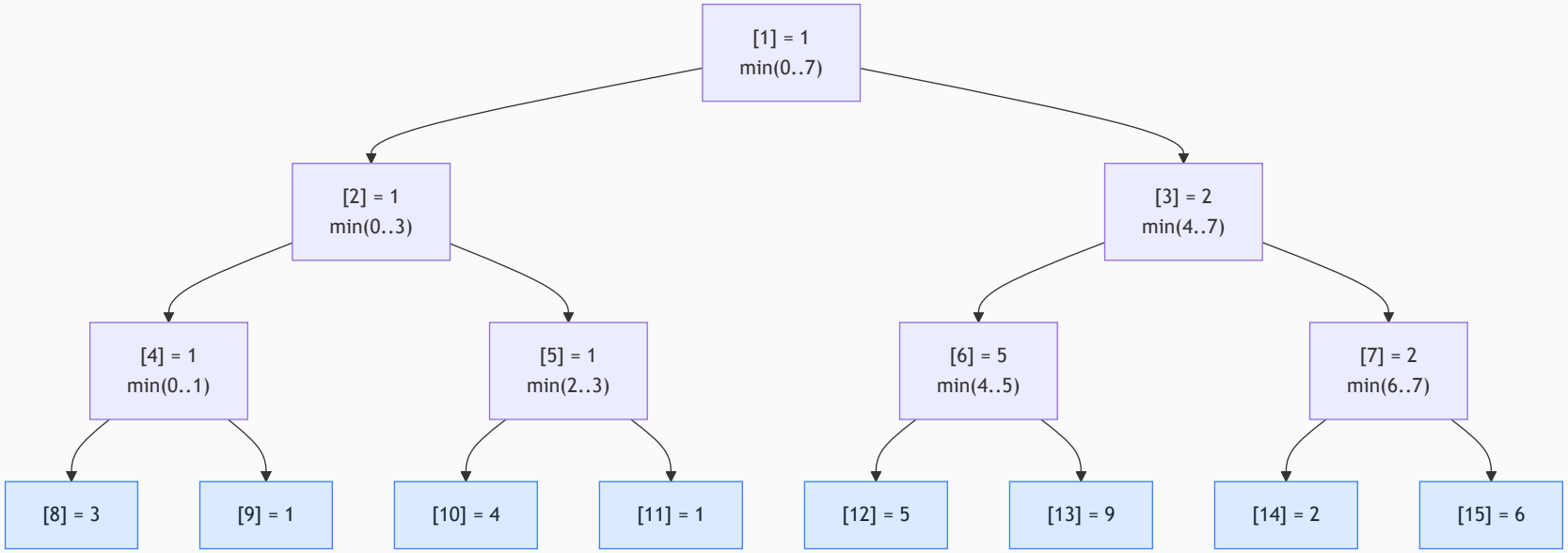


- **サイズ**: 配列 n → tree 配列 **2n**（`tree[n..2n-1]` が葉 = `a[0..n-1]`）
- **親子**: 子 = `tree[2i]`, `tree[2i+1]` / 親 = `tree[i/2]`
- **n は 2 の冪乗でなくてよい**（葉を `tree[n..2n-1]` に連続配置するため、任意の n で Query/Update が正しく動作する）

具体例: min のセグ木

配列 `a = [3, 1, 4, 1, 5, 9, 2, 6]`（n=8）：

`tree[] = [_, 1, 1, 2, 1, 1, 5, 2, 3, 1, 4, 1, 5, 9, 2, 6]`（`tree[0]` 未使用 / `tree[1]` = root / `tree[8..15]` = 葉）



構築（Build）：O(n)

```
SegTree(vector<T>& a, T e, function<T(T,T)> op)
    : n(a.size()), e(e), op(op), d(2 * a.size(), e) {

    // ① 葉に元の配列の値をコピー
    for (int i = 0; i < n; i++) d[i + n] = a[i];

    // ② 葉に近い方から順に、親 = op(左の子, 右の子) を計算
    for (int i = n - 1; i >= 1; i--)
        d[i] = op(d[2*i], d[2*i+1]);
}
```

```
a = [3, 1, 4, 1, 5, 9, 2, 6] n=8

① d[8]=3, d[9]=1, d[10]=4, d[11]=1, d[12]=5, ...
② d[7]=min(d[14],d[15])=min(2,6)=2 ← i=7 から開始
d[6]=min(d[12],d[13])=min(5,9)=5
...
d[1]=min(d[2],d[3])=min(1,2)=1 ← 根が完成
```

点更新（Update）：O(log n)

```
void update(int i, T x) {
    d[i + n] = x;                                  // ① 葉を書き換え
    for (i = (i + n) / 2; i >= 1; i /= 2)     // ② 親を順に再計算
        d[i] = op(d[2*i], d[2*i+1]);
}
```

a[2] を 4 → 0 に変更する場合（n=8）：

① d[10] = 0 ← 葉を更新

② d[5] = op(d[10], d[11]) = min(0, 1) = 0 ← 親

d[2] = op(d[4], d[5]) = min(1, 0) = 0 ← さらに親

d[1] = op(d[2], d[3]) = min(0, 2) = 0 ← 根

更新パス: d[10] → d[5] → d[2] → d[1]（3ステップ = log₂8）

> コード中の `i/2` は「親ノードに移動」を意味する（`>> 1` と同じ）

区間クエリ（Query）：O(log n)

```
// [l, r) の区間を求める（l は含む、r は含まない）
T query(int l, int r) {
    T vl = e, vr = e;              // 左右から集めた結果
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
        if (l % 2 == 1) vl = op(vl, d[l++]); // l が右の子なら確定
        if (r % 2 == 1) vr = op(d[--r], vr); // r が右の子なら確定
    }
    return op(vl, vr);
}
```

半開区間 `[l, r)` = 「l 以上 r **未満**」。 `query(2, 6)` → a[2], a[3], a[4], a[5] の4要素

`l % 2 == 1` = l が**右の子** → 親へ上がると区間外の左兄弟を巻き込む → d[l] を確定して `l++`（右隣の部分木へ）

`r % 2 == 1` = r が**右の子** → r 自体は範囲外なので `--r`（= r−1、左の子）を確定してから親へ

`op(d[--r], vr)` の順（d[--r] を左）は、非可換演算でも左→右の順序を保つため

Query の動き（具体例）

`query(1, 6)` = a[1]〜a[5] の min（n=8, `[l,r)` なので a[1..5]）

```
初期: l = 1+8 = 9, r = 6+8 = 14
step1: l=9(奇数=右の子) → vl = op(vl, d[9])=a[1]; l=10
r=14(偶数) → skip. l=10/2=5, r=14/2=7
step2: l=5(奇数) → vl = op(vl, d[5])=min(a[1], min(a[2..3]))
r=7(奇数) → vr = op(d[6], vr)=min(a[4..5]); r=6
l=3, r=3 → l==r 終了
結果: op(vl, vr) = min(a[1], min(a[2..3]), min(a[4..5])) = a[1..5] の min ✓
```

> `vl` / `vr` の2変数分離は非可換演算（行列積等）でも正しい順序を保つため

セグ木：完全な C++ 実装

```cpp
template<typename T>
struct SegTree {
    int n;
    vector<T> d;
    T e;                   // 単位元: op(e, x) == x となる値
    function<T(T,T)> op;   // 二項演算

    SegTree(int n, T e, function<T(T,T)> op)
        : n(n), e(e), op(op), d(2*n, e) {}

    SegTree(vector<T>& a, T e, function<T(T,T)> op)
        : n(a.size()), e(e), op(op), d(2*a.size(), e) {
        for (int i = 0; i < n; i++) d[i+n] = a[i];
        for (int i = n-1; i >= 1; i--) d[i] = op(d[2*i], d[2*i+1]);
    }

    void update(int i, T x) {                        // a[i] = x
        d[i+n] = x;
        for (i = (i+n)/2; i >= 1; i /= 2)
            d[i] = op(d[2*i], d[2*i+1]);
    }

    T query(int l, int r) {                           // [l, r)
        T vl = e, vr = e;
        for (l += n, r += n; l < r; l /= 2, r /= 2) {
            if (l % 2 == 1) vl = op(vl, d[l++]);
            if (r % 2 == 1) vr = op(d[--r], vr);
        }
        return op(vl, vr);
    }
};
```

セグ木の使い方

```
int n = 8;

// 区間 min（単位元 = INT_MAX: min(INT_MAX, x) == x）
SegTree<int> seg(n, INT_MAX, [](int a, int b){ return min(a, b); });

// 区間 sum（単位元 = 0: 0 + x == x）
SegTree<long long> seg(n, 0LL, [](long long a, long long b){ return a+b; });

seg.update(3, 5);           // a[3] = 5
cout << seg.query(0, n);    // a[0..n-1] 全体の min
cout << seg.query(2, 7);    // a[2..6] の min（[2,7) = 2,3,4,5,6）
```

単位元 = 演算の「影響しない値」。`op(e, x) == x` を満たすもの

min → `INT_MAX`、sum → `0`、max → `INT_MIN`、gcd → `0`、xor → `0`

注意: `query(l, r)` は半開区間 `[l, r)`。a[r] は**含まない**

Part 3

BIT（Fenwick Tree）

BIT の発想

BIT = **B**inary **I**ndexed **T**ree（二分インデックス木）。別名 Fenwick Tree。
セグ木より **実装が短くて定数倍が小さい**（高速）。

使える演算

- 区間 **sum**（加減算）
- 区間 **XOR**

使えない演算

- 区間 **min / max**
- 区間 **gcd**

なぜ制限がある？ BIT は **前置和（prefix sum）** を管理する構造。
区間 `[l, r]` の和 = `sum(r) - sum(l-1)` と **引き算で求める**。
min/max には引き算に相当する逆演算がないから使えない。

BIT は 1-indexed（配列の添字が 1 から始まる）。`i=0` を渡すと `i += i & -i` = `0 += 0` で**無限ループ**になるので注意！

セグ木 vs BIT：設計思想の違い

セグ木

元の要素を葉に置き、上に集約

```
min(0..7) = 1          ← 根
 ├─ min(0..3) = 1
 │    ├─ min(0..1) = 1
 │    │    ├─ 3    ← 元の要素（葉）
 │    │    └─ 1    ← 元の要素（葉）
 ...
```

葉 = **元の配列要素そのもの**

→ min / max / gcd **何でも OK**

BIT

前置和をパーツに分割して管理

```
tree[1] = a[1]          (1個)
tree[2] = a[1..2]       (2個)
tree[4] = a[1..4]       (4個)
tree[8] = a[1..8]       (8個)
```

各ノード = **部分和パーツ**

葉・内部ノードの区別なし

→ **引き算できる演算のみ**対応

> **本質**: BIT は `sum(r) - sum(l-1)` という引き算で区間和を求める。前置和さえ高速に得られれば十分なので「葉 = 元の要素」を持つ必要がない。この制約が「可逆演算のみ対応」に直結する。

前置和で区間和を解く

`S[i] = a[1] + a[2] + ... + a[i]`（前置和）を使うと区間和が **引き算1回** で求まる:

> **a[l..r] の和 = S[r] − S[l−1]**
> 例: a = [_, 3, 1, 4, 1, 5, ...]（1-indexed）→ a[2..5] = S[5]−S[1] = 14−3 = **11** ✓

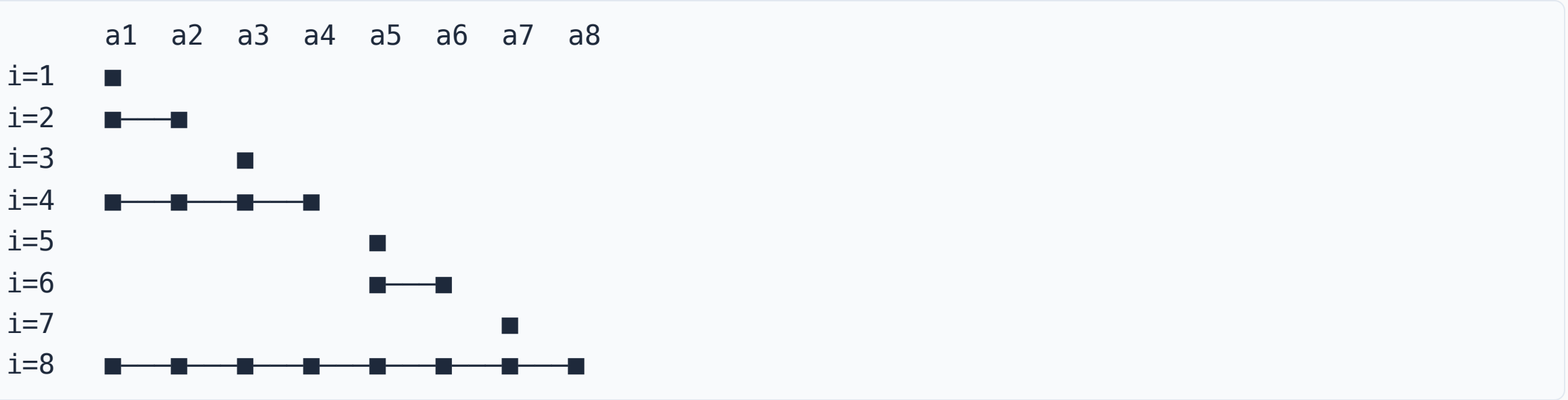
ナイーブ前置和の問題:

操作	ナイーブ前置和	BIT
区間クエリ	**O(1)** — S[r]−S[l−1]	O(log n)
点更新 a[i]+=v	**O(n)** — S[i..n] 全更新	**O(log n)**

> **BIT のアイデア**: 全前置和を持たず「**部分和パーツ**」を巧妙に管理 → 更新を O(log n) に削減

担当範囲の直感図（n=8）

`tree[i]` は `a[i−lowbit(i)+1 .. i]` の **lowbit(i) 個分** の和を保持:



sum(1..7) クエリ

7(`0111`) → tree[7] = a[7]

6(`0110`) → tree[6] = a[5..6]

4(`0100`) → tree[4] = a[1..4]

→ **3ステップで完結**

a[3] 更新

3(`011`) → tree[3] を更新

4(`100`) → tree[4] を更新

8(`1000`) → tree[8] を更新

→ **3ノードのみ**

担当範囲 = lowbit(i) の直感

2 進数の末尾 0 の個数が「担当範囲の広さ」を決める

i	2進数	末尾 0 の個数	担当範囲
1	`0001`	0個	**1 個**
2	`0010`	1個	**2 個**
4	`0100`	2個	**4 個**
8	`1000`	3個	**8 個**
6	`0110`	1個	**2 個**

クエリ `i -= lowbit(i)`

末尾の 1 ビットを除去

→ i は必ず減少する

→ **最大 $\lfloor \log_2 n \rfloor$ 回で i=0**

更新 `i += lowbit(i)`

末尾の 0 を埋めて上位へ

→ i は必ず増加する

→ **最大 $\log_2 n$ 回で i > n**

lowbit の仕組み

なぜ使う？ `lowbit(i)` = ノード i の管理要素数（例: i=4 → 4個 `a[1..4]`、i=6 → 2個 `a[5..6]`）。引いて下降（クエリ）、足して上昇（更新）→ O(log n)。

```
lowbit(i) = i & (−i)      ← i の最も下位の「1」のビットだけ残す
```

i	2進数	-i (2の補数)	i & (-i) = lowbit
1	`0001`	`1111`	`0001` = 1
2	`0010`	`1110`	`0010` = 2
3	`0011`	`1101`	`0001` = 1
4	`0100`	`1100`	`0100` = 4
6	`0110`	`1010`	`0010` = 2

> **-i の作り方（2の補数）**：全ビット反転 → +1
> 例: i=6 = `0110` → 反転 `1001` → +1 = `1010`。 `0110 & 1010 = 0010` = 2

各インデックスの管理範囲

`tree[i]` は **`a[i - lowbit(i) + 1]`** 〜 **`a[i]`** の **lowbit(i) 個の和** を管理:

```
i=1: tree[1] = a[1] (lowbit=1, 1個)
i=2: tree[2] = a[1] + a[2] (lowbit=2, 2個)
i=3: tree[3] = a[3] (lowbit=1, 1個)
i=4: tree[4] = a[1]+a[2]+a[3]+a[4] (lowbit=4, 4個)
i=5: tree[5] = a[5] (lowbit=1, 1個)
i=6: tree[6] = a[5] + a[6] (lowbit=2, 2個)
i=7: tree[7] = a[7] (lowbit=1, 1個)
i=8: tree[8] = a[1]+...+a[8] (lowbit=8, 8個)
```

法則: 各ノードの担当区間の長さ = lowbit(i)

前置和クエリ: O(log n)

`sum(i)` = `a[1] + a[2] + ... + a[i]`

手順: i から lowbit を **引き続けて** 0 になるまで tree を足す

例: sum(7) ― a[1]+a[2]+...+a[7] を求める

i=7 (0111) → 足す: tree[7] = a[7]
i -= lowbit(7)=1 → i=6

i=6 (0110) → 足す: tree[6] = a[5]+a[6]
i -= lowbit(6)=2 → i=4

i=4 (0100) → 足す: tree[4] = a[1]+a[2]+a[3]+a[4]
i -= lowbit(4)=4 → i=0 → 終了!

答え = tree[7] + tree[6] + tree[4]
= a[7] + (a[5]+a[6]) + (a[1]+...+a[4]) = a[1]+...+a[7] ✓

点更新: O(log n)

`add(i, v)` = `a[i] += v`

手順: i に lowbit を **足し続けて** n を超えるまで tree を更新

例: add(3, v) — a[3] に v を加算（n=8）

i=3 (011) → tree[3] += v tree[3] は a[3] を含む
i += lowbit(3)=1 → i=4

i=4 (100) → tree[4] += v tree[4] は a[1..4] を含む
i += lowbit(4)=4 → i=8

i=8(1000) → tree[8] += v tree[8] は a[1..8] を含む
i += lowbit(8)=8 → i=16 > n → 終了!

> **まとめ**: クエリは lowbit を **引く**（下位ビットを消す）、更新は lowbit を **足す**（上位に伝搬）

BIT：完全な C++ 実装

```
struct BIT {
    int n;
    vector<long long> tree;
    BIT(int n) : n(n), tree(n + 1, 0) {}   // ★ 1-indexed なので n+1
    void add(int i, long long v) {          // a[i] += v   （i は 1〜n）
        for (; i <= n; i += i & -i)  tree[i] += v;
    }
    long long sum(int i) {                  // a[1] + ... + a[i]
        long long res = 0;
        for (; i > 0; i -= i & -i)  res += tree[i];
        return res;
    }
    long long sum(int l, int r) {           // a[l] + ... + a[r]
        return sum(r) - sum(l - 1);         // ← ここで「引き算」を使う
    }
};
```

> `i & -i` = `lowbit(i)`。`i += i & -i` で上位伝搬、`i -= i & -i` で下位消去

BIT の使い方

```
int n = 8;
BIT bit(n);

// 初期配列 a = [_, 3, 1, 4, 1, 5, 9, 2, 6] から構築（1-indexed!）
int a[] = {0, 3, 1, 4, 1, 5, 9, 2, 6};
for (int i = 1; i <= n; i++) bit.add(i, a[i]);

bit.sum(1, 5);          // a[1]+...+a[5] = 3+1+4+1+5 = 14
bit.add(3, 10);         // a[3] += 10  → a[3] = 14
bit.sum(1, 5);          // 3+1+14+1+5 = 24
```

> **よくある罠**: 0-indexed の配列を扱うとき、BIT に渡す前に **+1** するのを忘れない！
> `bit.add(i + 1, a[i]);` `bit.sum(l + 1, r + 1);`

応用: 転倒数（Inversion Count）

転倒数 = 配列中で `i < j` なのに `a[i] > a[j]` となるペアの数（= バブルソートの交換回数）

```
[3, 1, 4, 2] の転倒数:
  (3,1) (3,2) (4,2) → 転倒数 = 3
```

```cpp
// 左から順に「自分より大きい値が既に何個あるか」を BIT で数える
long long inv = 0;
for (int i = 0; i < n; i++) {
    inv += bit.sum(a[i] + 1, n);   // a[i] より大きい値の個数
    bit.add(a[i], 1);               // a[i] を「登録済み」にする
}
```

> このコードは値が **1〜n の整数**であることを前提とする。
> **0・負数が含まれる場合**や**値域が n を超える場合**は必ず **座標圧縮**（値を 1〜n にマッピング）してから BIT を使う。
> `a[i] = 0` を渡すと `bit.add(0, 1)` でバグになるので注意！

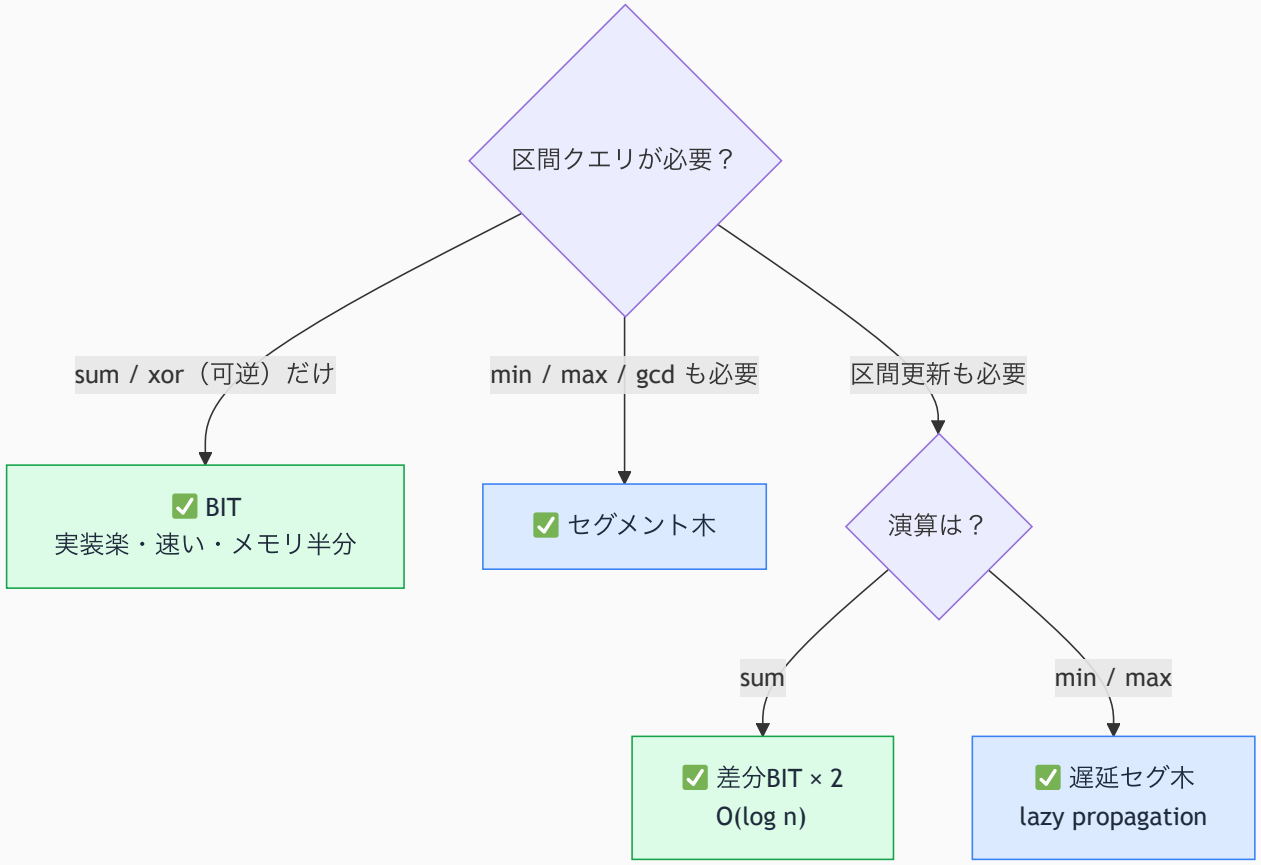
Part 4

まとめ

比較表

項目	セグメント木	BIT
点更新	O(log n)	O(log n)
区間クエリ	O(log n)	O(log n)
扱える演算	**何でも OK** (min, max, sum, gcd)	可逆演算のみ（sum / XOR が代表例）
実装量	約25行（テンプレ推奨）	**約15行**
定数倍	やや重い	**軽い**（1.5〜2倍速）
メモリ	2n	n+1
引数の添字	0-indexed	**1-indexed**（要注意）
内部配列	1-indexed（d[1]=root）	1-indexed
区間更新	遅延セグ木で対応	差分BIT×2で対応

使い分けフロー



迷ったらセグ木。単位元さえ決まれば何でも使える汎用性が強み。
速度重視ならBIT。sum だけなら BIT 一択。

発展トピック

遅延セグ木

区間更新 + 区間クエリを両方 $O(\log n)$ で処理

2次元BIT

2次元配列の区間和。$O(\log^2 n)$ で処理可能

座標圧縮 + BIT

値域が大きい転倒数問題に使用

永続セグ木

セグ木の各バージョンを保持。空間 $O(n \log n)$

セグ木上の二分探索

`walk` 操作で $O(\log n)$ の二分探索が可能

練習問題

レベル	問題	使うもの
入門	AIZU DSL-2-B (Range Sum Query)	BIT
基礎	AtCoder Library Practice B	BIT
基礎	AtCoder Library Practice J	セグ木
応用	転倒数カウント問題	座標圧縮 + BIT
応用	区間最大値 + 点更新	セグ木 (max)
発展	区間 add + 区間 min	遅延セグ木

セグ木・BIT

まとめ

- **BIT**: 区間和 + 点更新に特化。15行。最速
- **セグ木**: min/max/gcd も OK。汎用テンプレート
- 両方とも **O(log n)** で高速処理
- 迷ったら**セグ木**、sum だけなら**BIT**

次のステップ → 遅延セグ木