

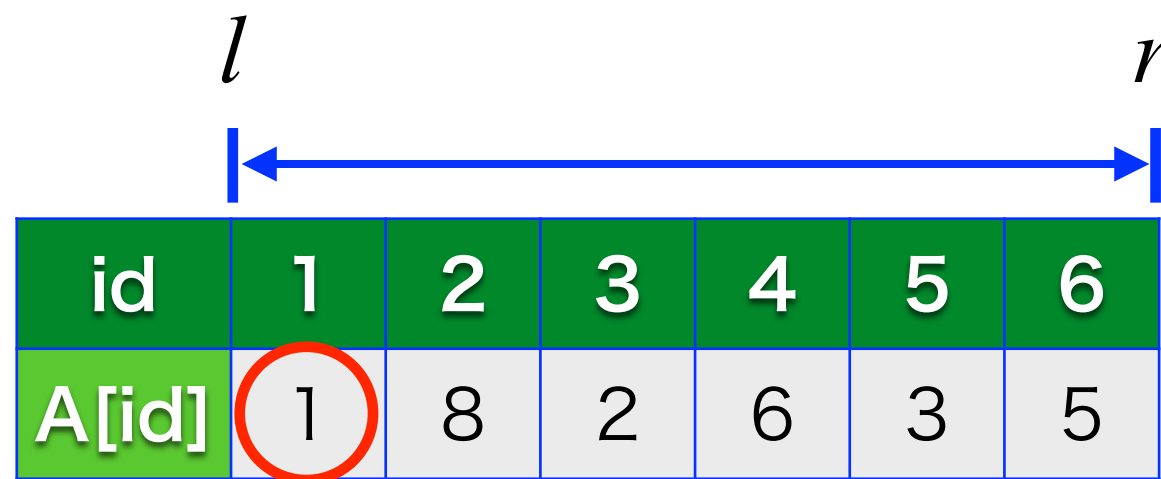
# RMQ クエリ処理

北海道大学大学院 情報科学研究科

博士1年 井上 祐馬

# RMQ

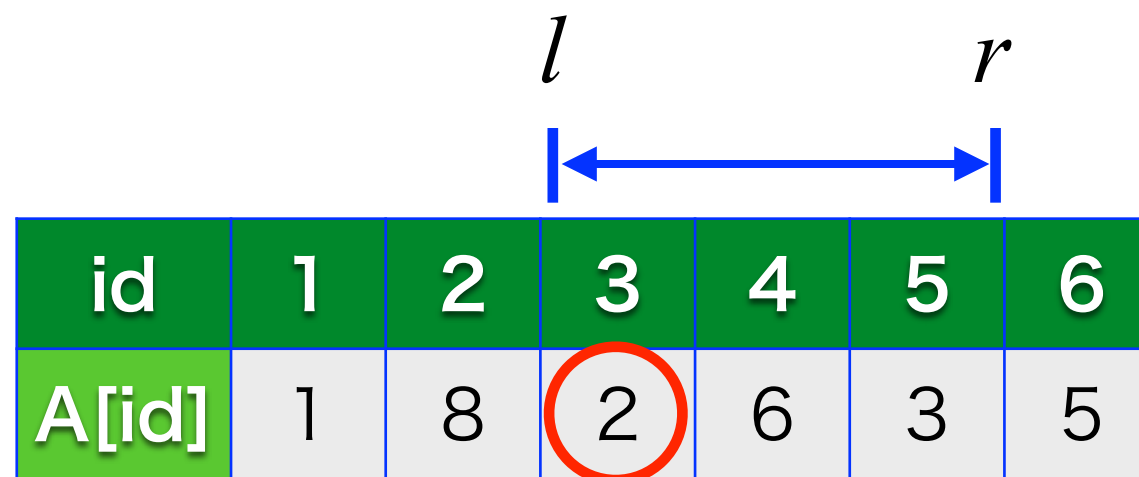
- RMQ: Range Minimum Query (区間最小値)
  - 列  $A[1:n]$  上の区間  $[l, r]$  に対するクエリ  $\text{RMQ}(l, r)$
  - $A[l:r]$  中での最小値  $A[i]$  を返す
  - 基本的に最大値クエリも同様に処理できる



id	1	2	3	4	5	6
A[id]	1	8	2	6	3	5

# RMQ

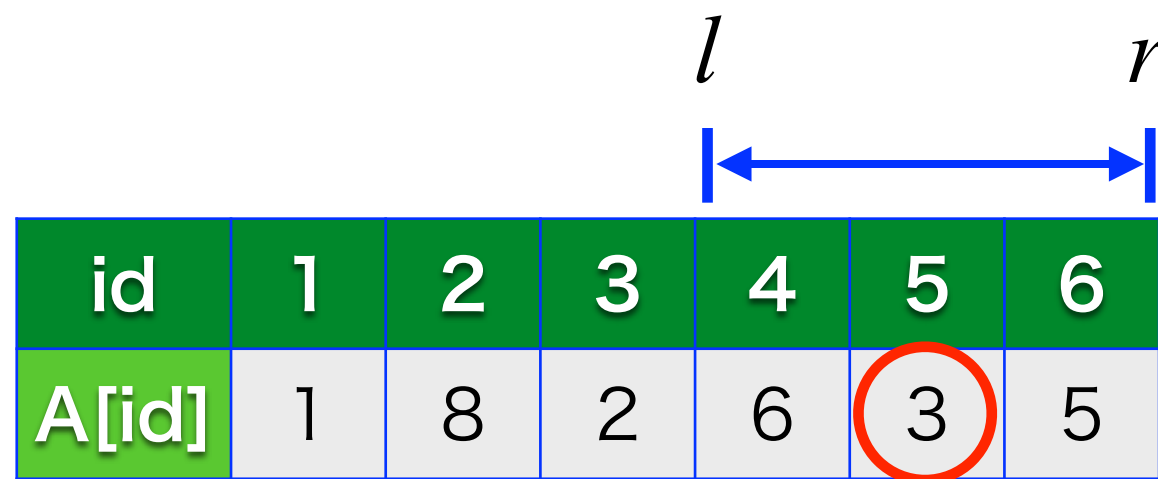
- RMQ: Range Minimum Query (区間最小値)
  - 列  $A[1:n]$  上の区間  $[l, r]$  に対するクエリ  $\text{RMQ}(l, r)$
  - $A[l:r]$  中での最小値  $A[i]$  を返す
  - 基本的に最大値クエリも同様に処理できる



id	1	2	3	4	5	6
A[id]	1	8	2	6	3	5

# RMQ

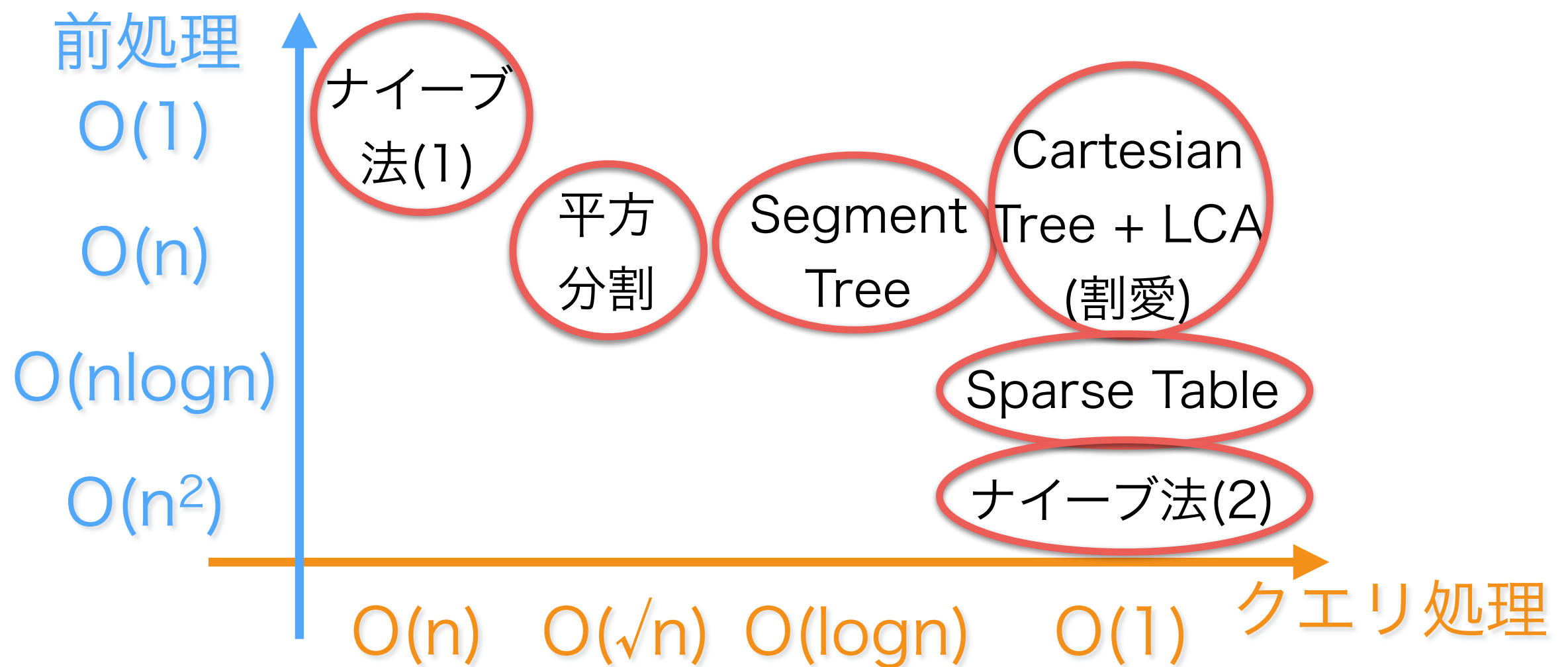
- RMQ: Range Minimum Query (区間最小値)
  - 列  $A[1:n]$  上の区間  $[l, r]$  に対するクエリ  $\text{RMQ}(l, r)$
  - $A[l:r]$  中での最小値  $A[i]$  を返す
  - 基本的に最大値クエリも同様に処理できる



id	1	2	3	4	5	6
A[id]	1	8	2	6	3	5

# RMQを処理するアルゴリズム

- ・ クエリ処理アルゴリズムの計算量:
  - ・ 前処理時間
  - ・ 1つのクエリの処理時間



# RMQアルゴリズム

- ・ 平方分割
- ・ Segment Tree
- ・ Sparse Table

# RMQアルゴリズム

- ・ 平方分割
- ・ Segment Tree
- ・ Sparse Table

# バケット分割

- ・ 長さ  $n$  の列を長さ  $s$  のバケットに分割
  - ・ バケットの数  $B = n/s$
  - ・ 各バケットの最小値を  $O(n)$  で前計算
- ・ 全体の RMQ クエリは、以下のクエリに分割される
  - ・ バケットの区間に対する RMQ クエリ1回
  - ・ バケット内部に対する RMQ クエリ2回

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
a[1:n]	1	2	1	3	4	3	5	3	6	3	1	8	4	9	11	4	1	5	6	3	2	5	2	5	2	1	9	8	4	1
b[1:B]	1			3			3			1			4			1			2			2						1		



# 平方分割

- ・ 長さ  $n$  の列を長さ  $\sqrt{n}$  のバケットに分割
  - ・ バケットの数  $B = n/\sqrt{n} = \sqrt{n}$
- ・ 全体の RMQ クエリは、以下のクエリに分割される
  - ・ バケットの区間に対する RMQ クエリ1回
    - $\sqrt{n}$ 個の最小値インデックスを比較  $O(\sqrt{n})$
  - ・ バケット内部に対する RMQ クエリ2回
    - $\sqrt{n}$ 個の最小値インデックスを比較  $O(\sqrt{n})$
- ・ 全体でも  $O(\sqrt{n})$

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
a[1:n]	1	2	1	3	4	3	5	3	6	3	1	8	4	9	11	4	1	5	6	3	2	5	2	5	2	1	9	8	4	1
b[1:√n]	1					3					1					1					2					1				

Diagram illustrating square root decomposition on an array. The array is divided into buckets of size  $\sqrt{n}$ . A query for the minimum value in the range [1, 30] is shown, involving bucket-level and intra-bucket queries.

# 平方分割の実装 (構築)

```
class Bucket{
    //sqrtN: bucket size, B: the number of bucket
    int n, sqrtN, B;
    vector<int> val, bucket;
public:
    Bucket(vector<int> a):val(a){
        n = a.size(); sqrtN = ceil(sqrt(n));
        B = (n + sqrtN - 1) / sqrtN;

        bucket.resize(B, INF);
        for(int i=0; i<B; i++){
            for(int j=sqrtN*i; j<sqrtN*(i+1); j++){
                if(j>=n) break;
                bucket[i] = min(bucket[i], val[j]);
            }
        }
    }
}
```

# 平方分割の実装 (クエリ)

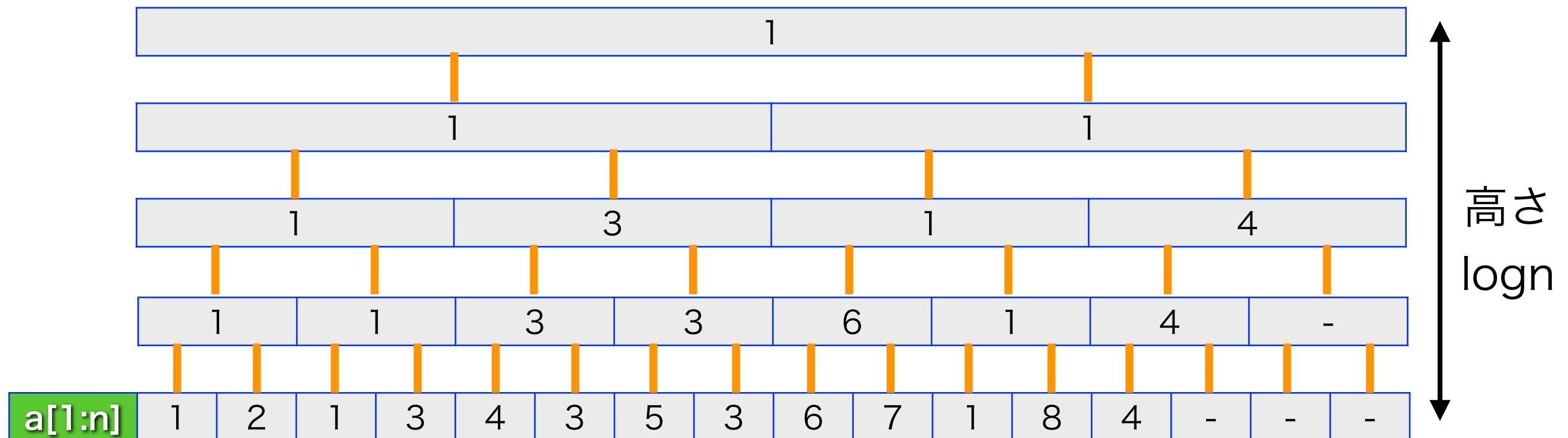
```
int RMQ(int l, int r){  
    //index in bucket  
    int L = (l + sqrtN - 1) / sqrtN + 1, R = r / sqrtN;  
  
    int res = INF;  
    if(L<=R){  
        for(int i=l;i<sqrtN*L;i++)res = min(res, val[i]);  
        for(int i=L;i<R;i++)res = min(res, bucket[i]);  
        for(int i=sqrtN*R;i<r;i++)res = min(res, val[i]);  
    }else{  
        //interval is included in one bucket  
        for(int i=l;i<r;i++)res = min(res, val[i]);  
    }  
  
    return res;  
}
```

# RMQアルゴリズム

- ・ 平方分割
- ・ **Segment Tree**
- ・ Sparse Table

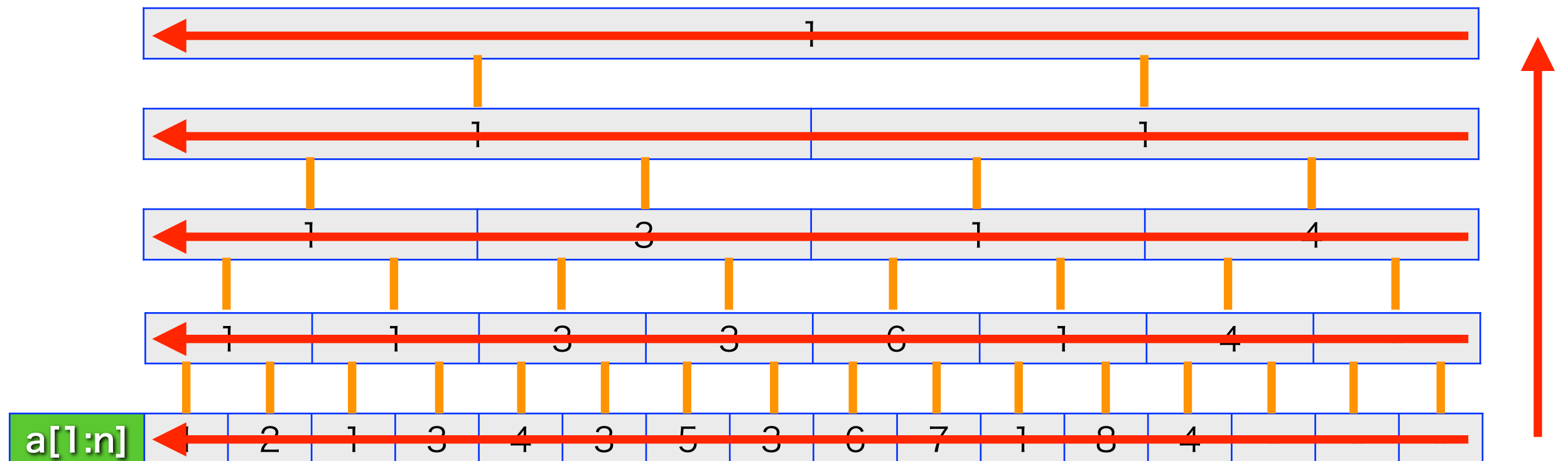
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ 一見  $O(n \log n)$  空間に見えて、実は  $2n-1$  節点



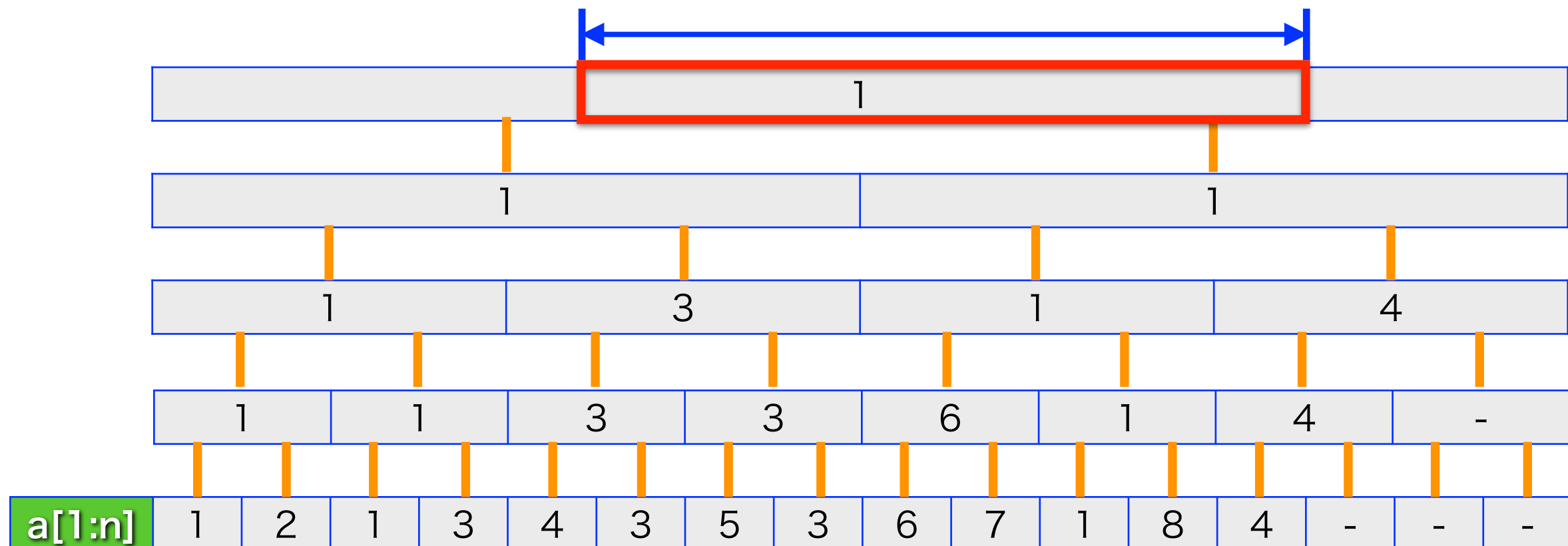
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ 構築：葉の方から順に、2つの子の最小値を計算  $O(n)$



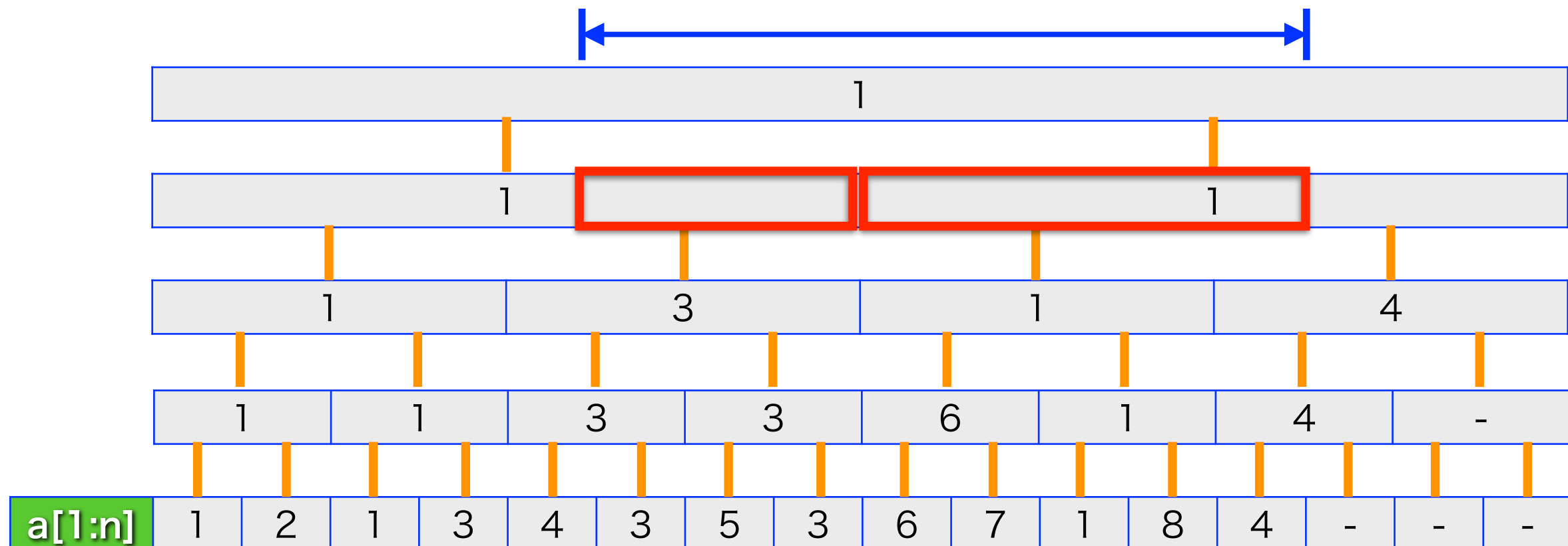
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ クエリ：クエリ区間を節点に対応するように分割し、  
その中の最小値を返す  $O(\log n)$



# Segment Tree

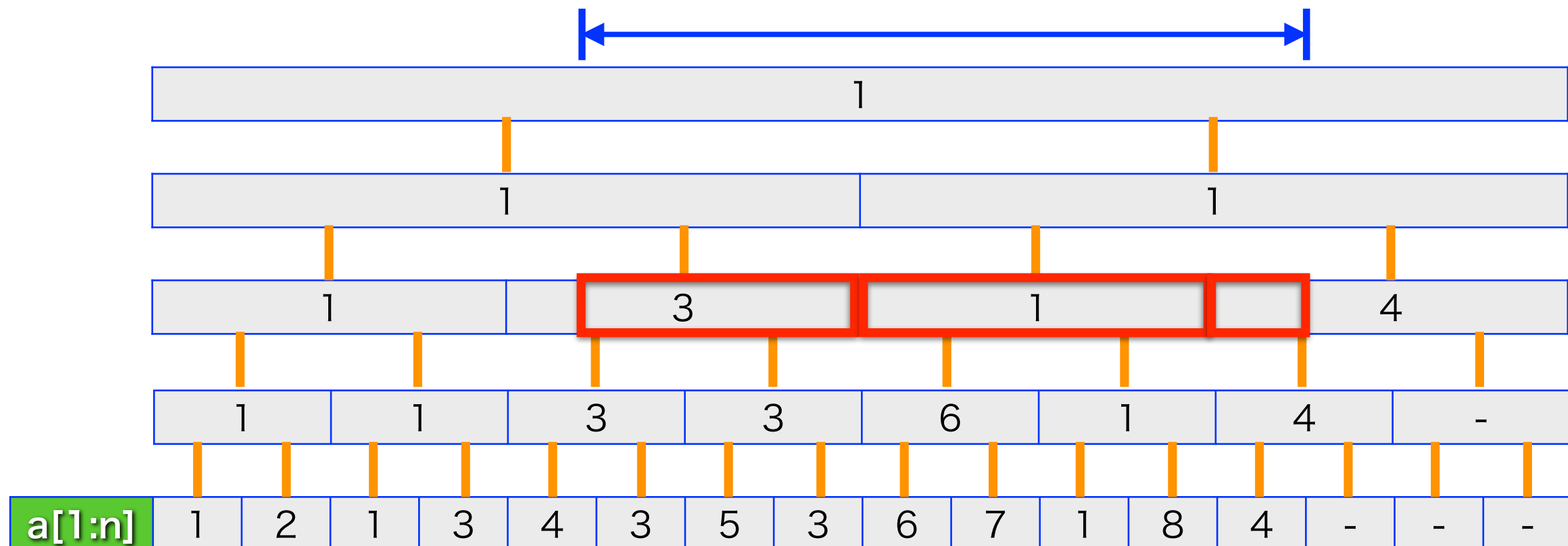
- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ クエリ：クエリ区間を節点に対応するように分割し、  
その中の最小値を返す  $O(\log n)$





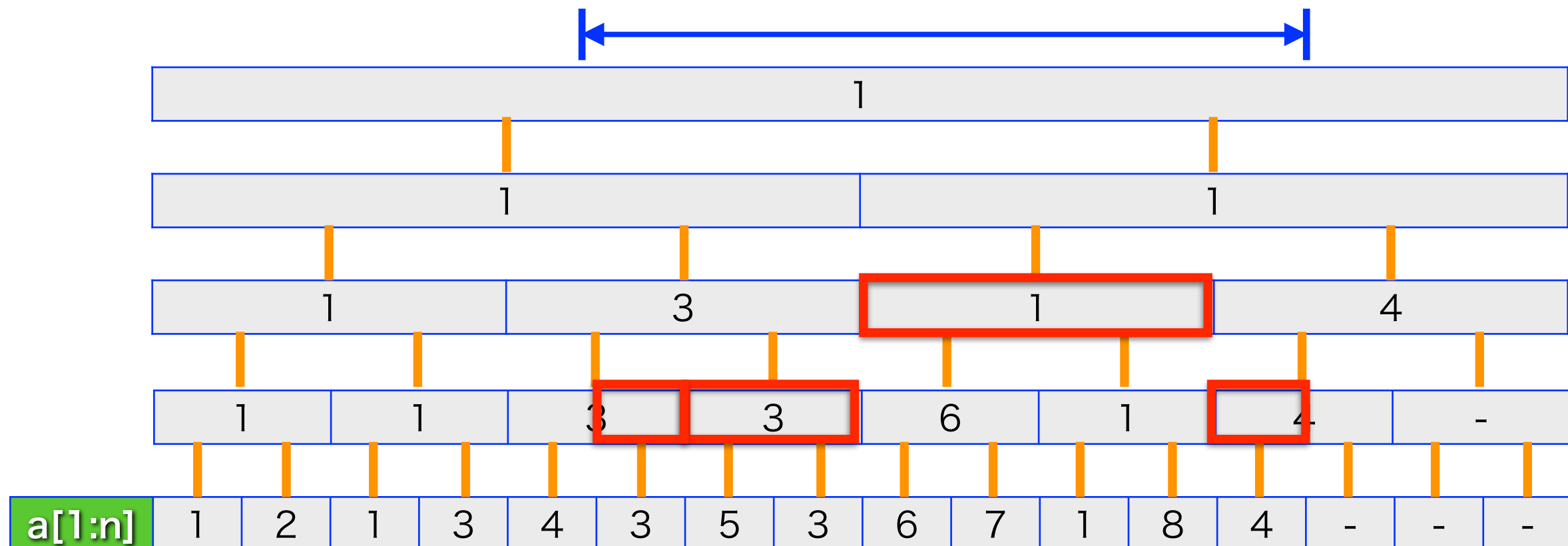
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ クエリ：クエリ区間を節点に対応するように分割し、  
その中の最小値を返す  $O(\log n)$



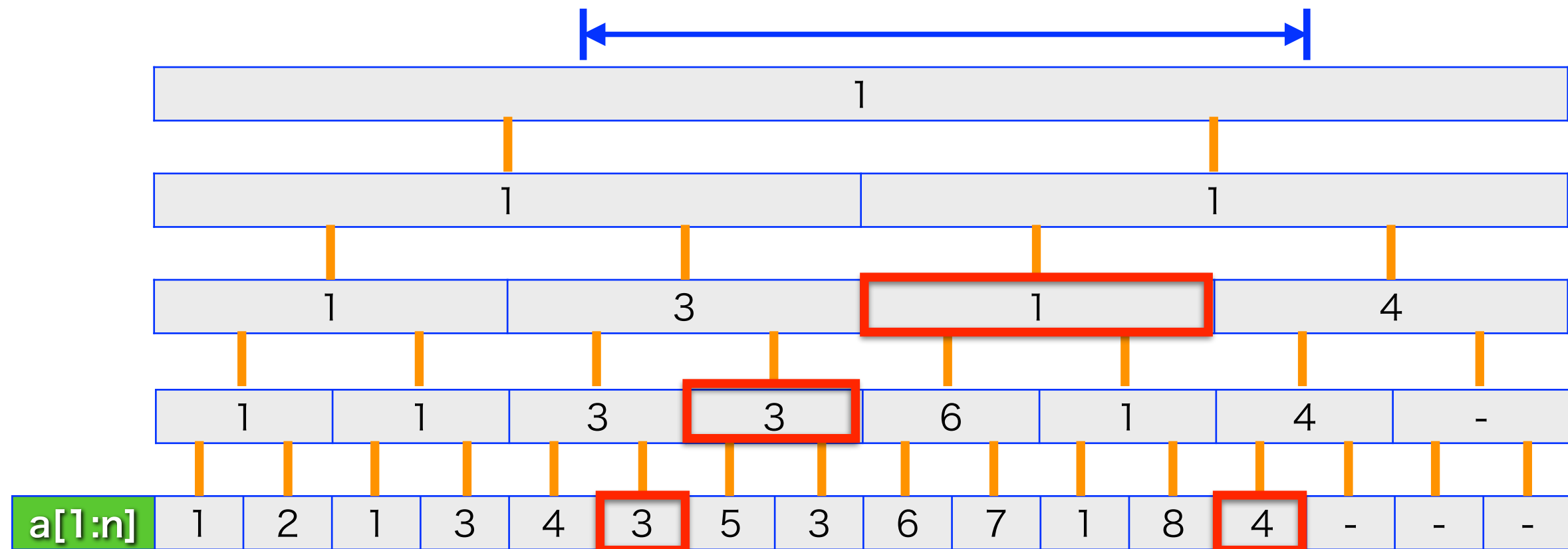
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ クエリ：クエリ区間を節点に対応するように分割し、  
その中の最小値を返す  $O(\log n)$



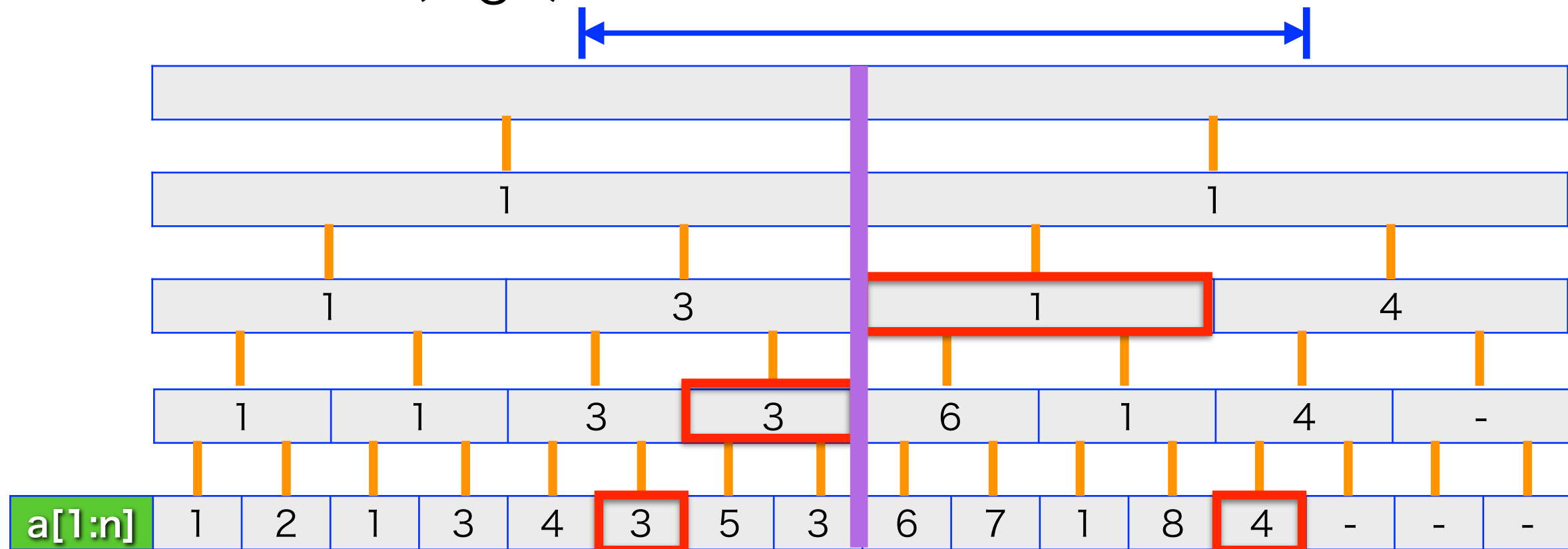
# Segment Tree

- ・ 列を2分木の葉に割り当てる
- ・ 各節点は2つの子のうち最小値を保持
- ・ クエリ：クエリ区間を節点に対応するように分割し、  
その中の最小値を返す  $O(\log n)$



# Segment Tree

- クエリ：クエリ区間を節点に対応するように分割し、その中の最小値を返す  $O(\log n)$
- 最初の分割後の下降において、2つの子のうち片方は必ず区間一致、もしくは範囲外となり、それ以上降りない  
= 高々  $O(\log n)$  回の節点参照



# Segment Tree の実装 (構築)

```
class SegmentTree{
    int n;
    //represent tree as array
    //parent of k : (k-1)/2, child of k : 2*k+1, 2*k+2
    vector<int> node;

public:
    SegmentTree(vector<int> a){
        int n_ = a.size();

        //round to power of 2
        n=1;
        while(n<n_)n*=2;

        //initialize tree
        node.resize(2*n-1, INF);

        //fill data into tree
        for(int i=0;i<n_;i++)node[n-1+i] = a[i];
        for(int i=n-2;i>=0;i--)node[i] = min(node[2*i+1], node[2*i+2]);
    }
}
```

# Segment Tree の実装 (クエリ)

```
//return minimum value in [a,b). ( [l,r) is interval in which k is.)
int RMQ(int a,int b,int k=0,int l=0,int r=0){
    if(l>=r)r = n;
    if(r<=a || b<=l)return INF;           //out of range
    if(a<=l && r<=b)return node[k];       //interval match

    int vl = RMQ(a,b,2*k+1,l,(l+r)/2);
    int vr = RMQ(a,b,2*k+2,(l+r)/2,r);
    return min(vl,vr);
}
```

# RMQアルゴリズム

- ・ 平方分割
- ・ Segment Tree
- ・ **Sparse Table**

# Sparse Table

- ・  $i$  番目から長さ  $2^k$  の区間の最小値をそれぞれ記憶した  $O(n \log n)$  のテーブルを持つ
- ・ 構築:  $O(n \log n)$ 
  - ・  $S[i][k+1] = \min( S[i][k], S[i+2^k][k] )$

i	1	2	3	4	5	6	7	8
A[i]	2	1	8	3	7	2	5	6
S[i][0]	2	1	8	3	7	2	5	6
S[i][1]	1	1	3	3	2	2	5	-
S[i][2]	1	1	2	2	5	-	-	-
S[i][3]	1	-	-	-	-	-	-	-



# Sparse Table

- ・  $i$  番目から長さ  $2^k$  の区間の最小値をそれぞれ記憶した  $O(n \log n)$  のテーブルを持つ
- ・ クエリ:  $O(1)$
- ・  $\text{RMQ}(i, j) = \min( S[i][k], S[j-2^k+1][k] )$
- ・  $k = \text{floor}( \log(j-i) ) \leftarrow O(1)$  で求められると仮定

$i$	1	2	3	4	5	6	7	8
$A[i]$	2	1	8	3	7	2	5	6
$S[i][0]$	2	1	8	3	7	2	5	6
$S[i][1]$	1	1	3	3	2	2	5	-
$S[i][2]$	1	1	2	2	5	-	-	-
$S[i][3]$	1	-	-	-	-	-	-	-

25

# Sparse Table の実装

```
class SparseTable{
    vector< vector<int> > table;
public:
    SparseTable(vector<int> a){
        int n = a.size(), logn = 31 - __builtin_clz(n);

        //initialize table
        table.resize(n, vector<int>(logn+1, INF));
        for(int i=0; i<n; i++) table[i][0] = a[i];

        //construct table
        for(int j=0; j<logn; j++){
            for(int i=0; i<n; i++){
                table[i][j+1] = min(table[i][j], (i+(1<<j)<n)?table[i+(1<<j)][j]:INF);
            }
        }

        //return the minimum value in [l, r)
        int RMQ(int l, int r){
            int ln = 31 - __builtin_clz(r-l);
            return min(table[l][ln], table[r-(1<<ln)][ln]);
        }
    };
};
```

# RMQのアルゴリズム

	空間	前処理	クエリ	動的
平方分割	$O(n)$	$O(n)$	$O(\sqrt{n})$	$\bigcirc$ ( $O(\sqrt{n})$ )
Segment Tree	$O(n)$	$O(n)$	$O(\log n)$	$\bigcirc$ ( $O(\log n)$ )
Sparse Table	$O(n \log n)$	$O(n \log n)$	$O(1)$	$\times$ ( $O(n)$ )

# 動的クエリ処理

- ・ 値変更クエリ
  - ・ セグメント木：葉の値の変更を親に伝える  $O(\log n)$
  - ・ 平方分割：バケット最小値を更新  $O(\sqrt{n})$
- ・ 値挿入クエリ
  - ・ セグメント木：クエリの実行ができれば先に $\infty$ などで確保しておく
  - ・ 平方分割：バケットに追加、サイズが大きくなりすぎたら分割  $O(\sqrt{n})$

# Segment Tree の実装 (更新)

```
void update(int k, int v){
    k += n-1;
    node[k] = v;
    //traverse from leaf to root
    while(k>0){
        k = (k-1)/2;
        node[k] = min(node[k*2+1], node[k*2+2]);
    }
}
```

# 平方分割の実装 (更新)

```
void update(int k, int v){  
    //update of array  
    val[k] = v;  
  
    //update of bucket  
    int X = k/sqrtN;  
    bucket[X] = INF;  
    for(int i=sqrtN*X; i<sqrtN*(X+1); i++){  
        if(i>=n) break;  
        bucket[X] = min(bucket[X], val[i]);  
    }  
}
```

# その他クエリ処理

- ・ セグメント木 (平方分割) は他にも様々な問題に利用可能
  - ・ 区間和/積、区間GCD/LCM、etc...
  - ・ 結合則が成り立つ演算なら基本的にOK
- ・ 区間和に対するクエリに特化した BIT (Binary Index Tree = Fenwick Tree) という構造もある (そのうち紹介)
- ・ 1次元 (列) を多次元に拡張することもできる (そのうち紹介)
- ・ 遅延更新というテクニックで、さらに幅広いクエリを処理できる (そのうち紹介)

# まとめ

- ・ RMQ を処理する典型的アルゴリズムの紹介
  - ・ 平方分割
  - ・ セグメント木
  - ・ Sparse Table
- ・ (一部は) 動的なデータの変更にも対応できる
- ・ これらのアルゴリズム (データ構造) は他の問題にも幅広く応用される