HCPC 勉強会 (2016/12/13)

「Binary Indexed Tree」

北海道大学情報エレクトロニクス学科
情報理工学コースB3 杉江祐哉

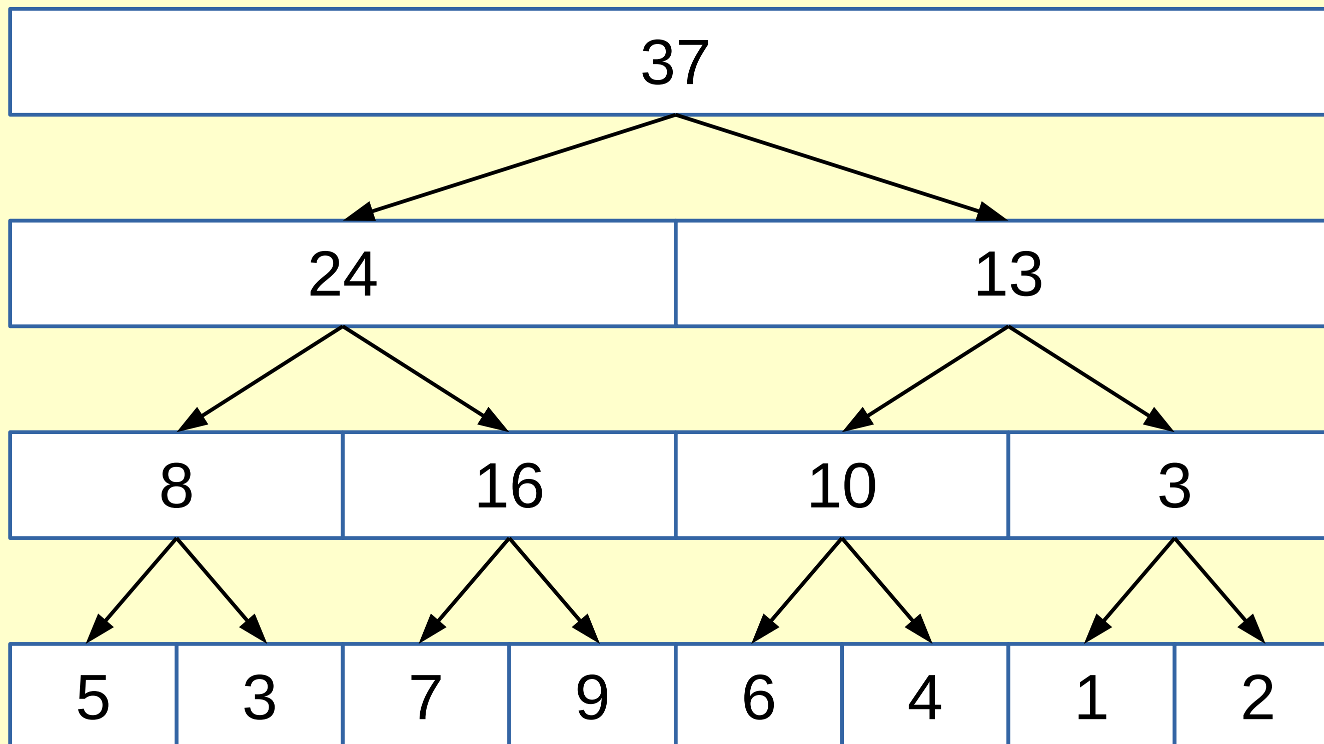
Binary Indexed Tree #とは

次のことが実現できるデータ構造！

- i が与えられたとき、
  累積和 $a_1 + a_2 + \cdots + a_i$ を計算
- i と x が与えられたとき、a_i に x を足す

セグメント木による表現

- これはセグメント木でも実現できる！

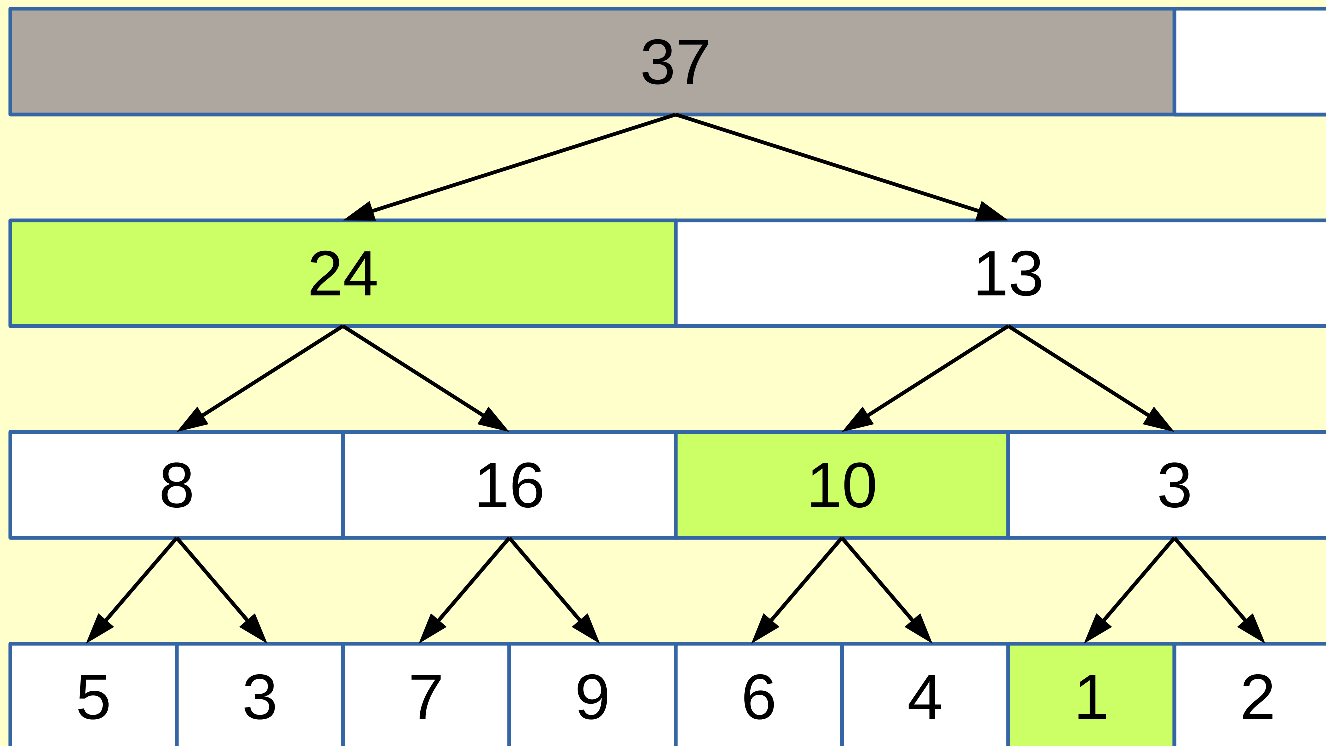


各節点は、対応する区間の和を持っている

計算クエリ

- 区間を完全に被覆するまで下がっていく

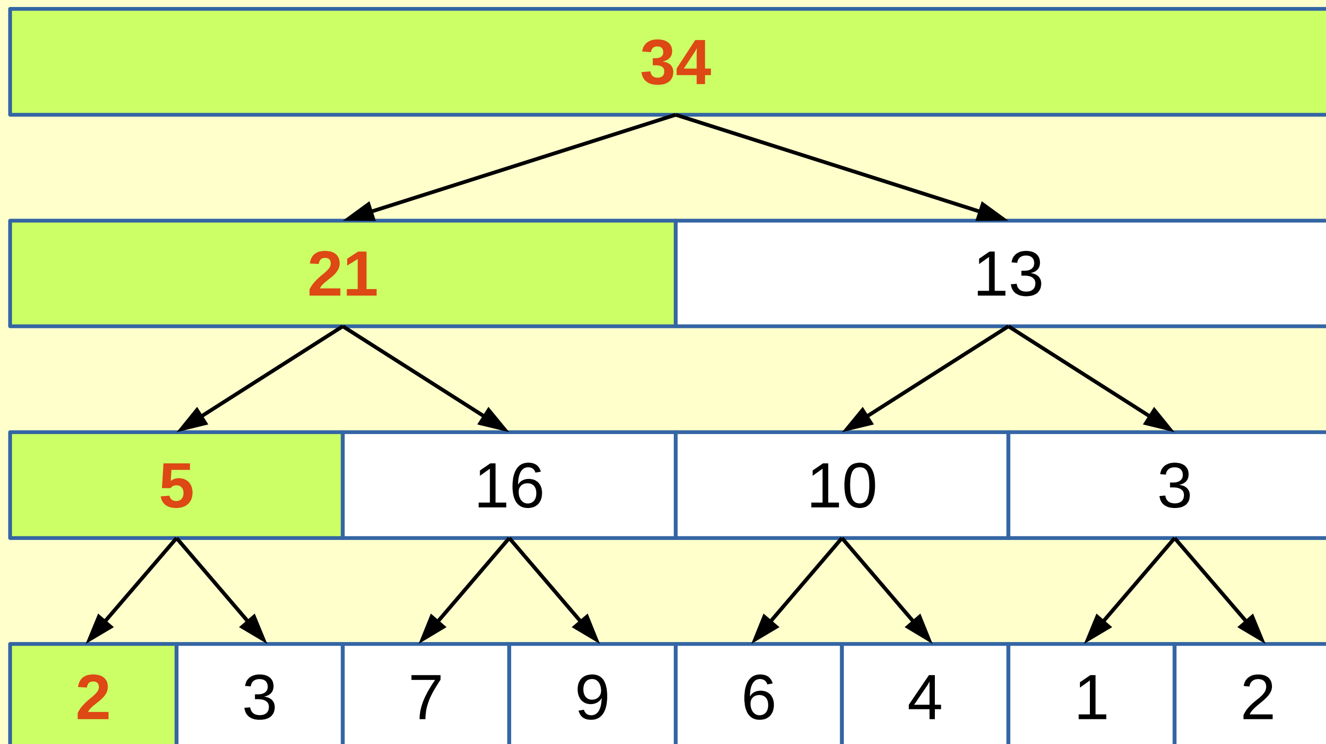
(例) $a_1 + a_2 + \cdots + a_7$



24 + 10 + 1 = 35 が答え！

更新クエリ

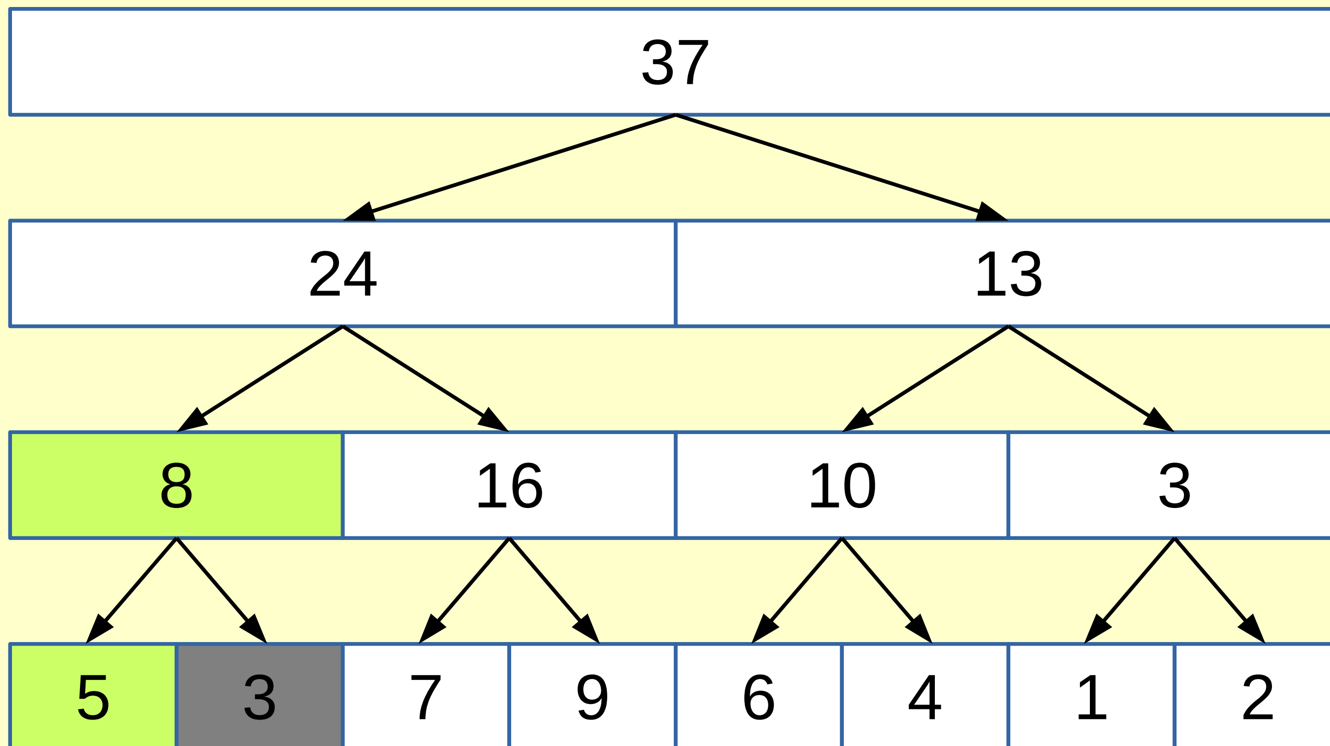
- 関係ある区間を更新していく
  (例) a_1 を 5 から 2 に変える (-3 足す)



緑で塗られた区間が更新される！

セグメント木の無駄なところ (1)

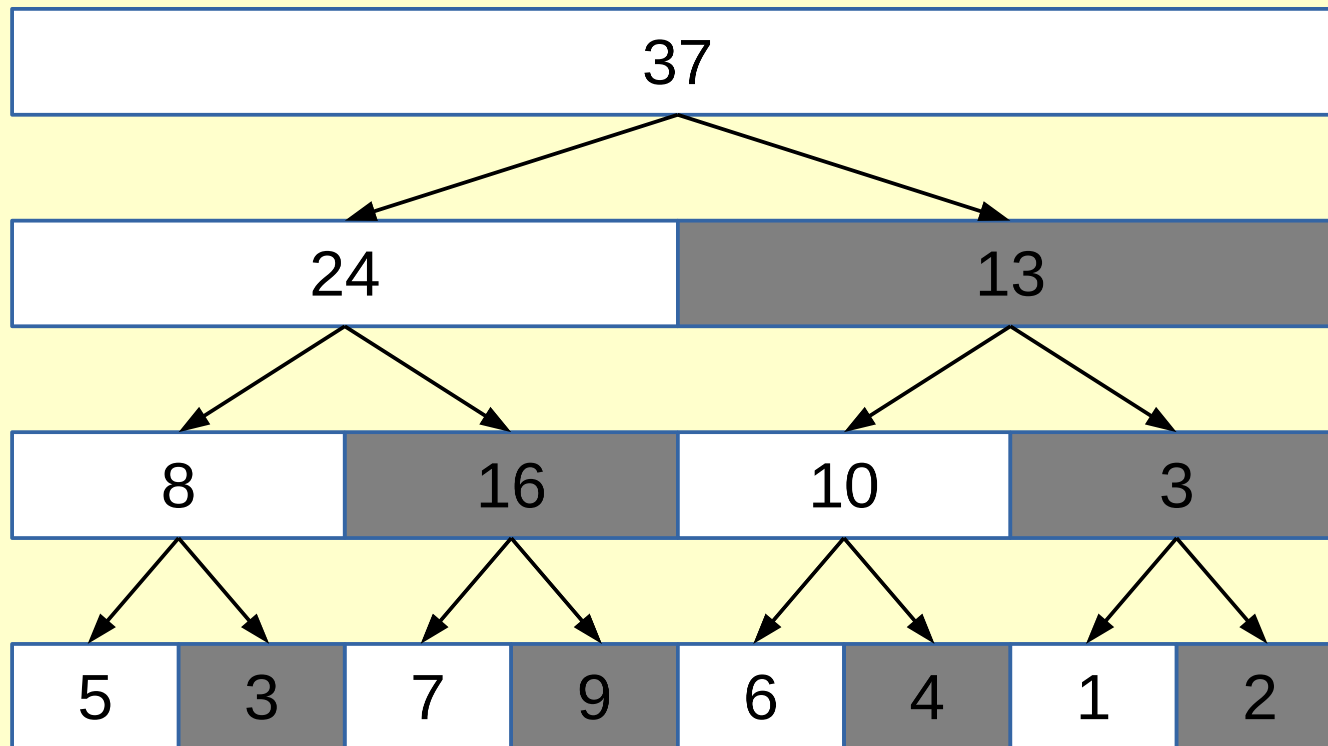
- (s から t までの和)
  = (1 から t までの和) – (1 から (s-1) までの和)



たとえば、「3」の情報は左と上の情報から
得られるので、おぼえる必要がない！

セグメント木の無駄なところ (2)

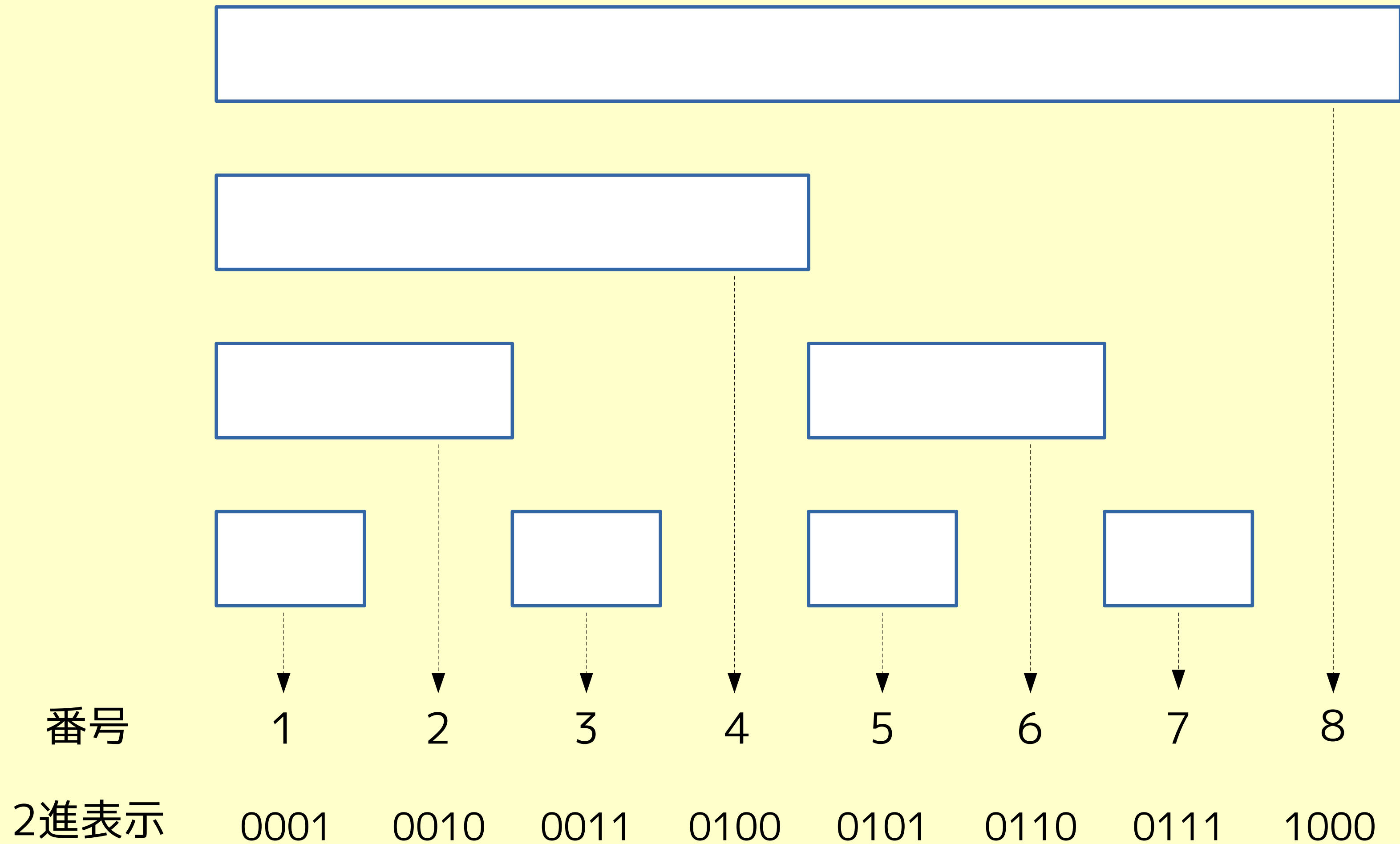
- 「左と上を見たら計算できるし、いらなくね？」となるような箇所はこんなにある



この考え方を使うと、BITのデータ構造にたどりつけるぞ！

BIT のしくみ (1)

- 配列に、次のように対応する部分の和を持つ

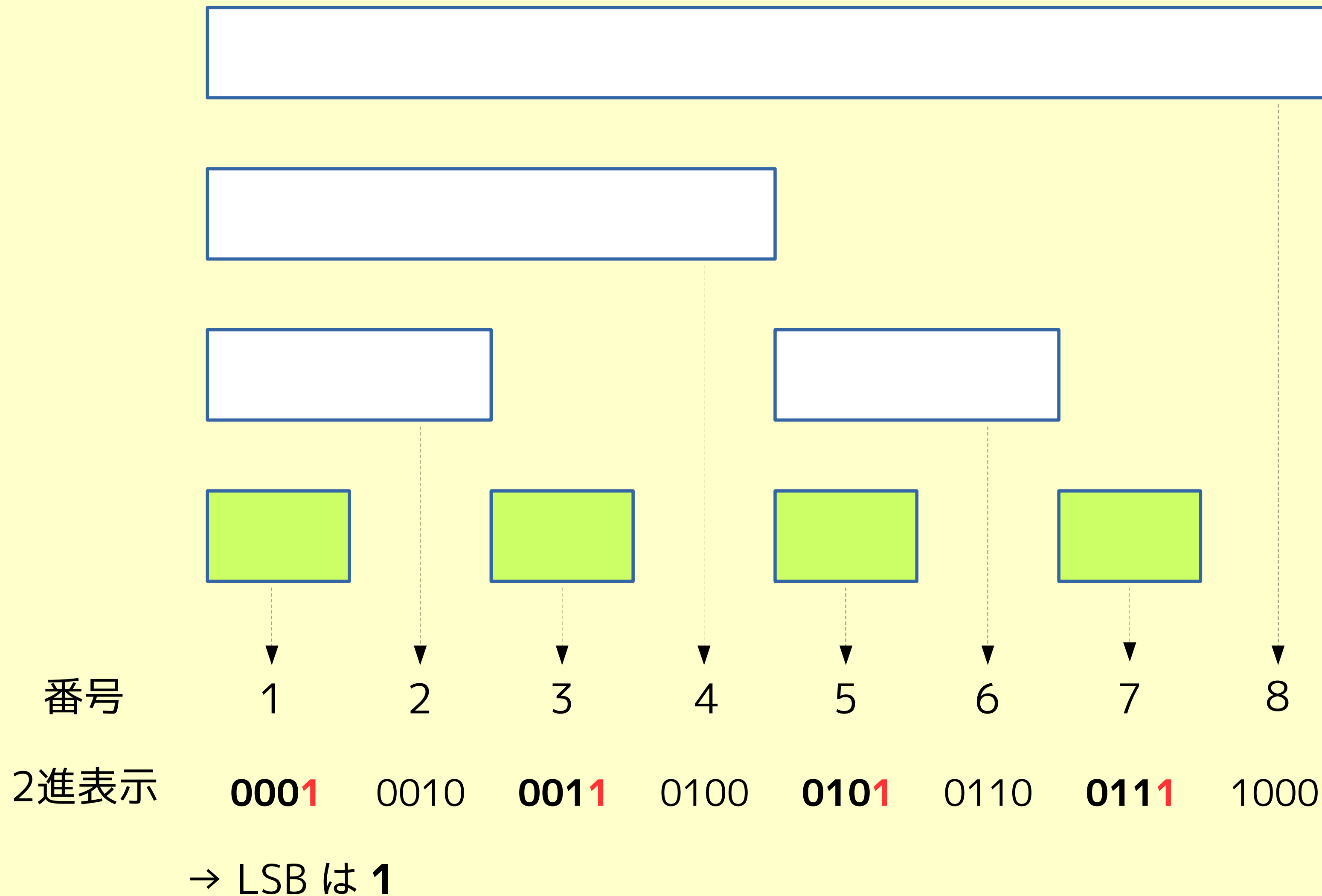


各要素の **LSB** (Least Significant Bit) は？

→ 2進数で表現した時に初めて「1」が来るのは右から何番目？

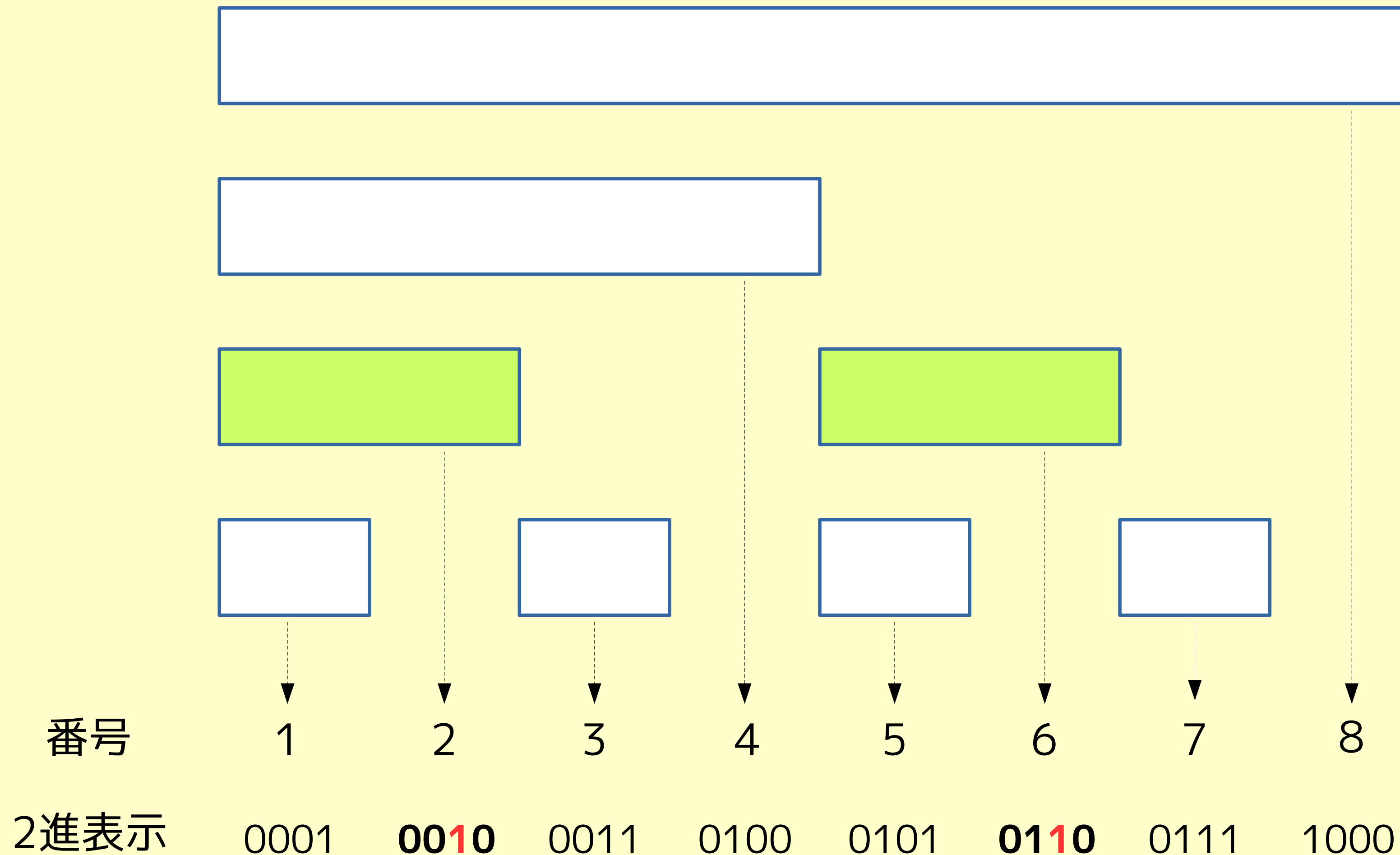
BIT のしくみ (2)

- 配列に、次のように対応する部分の和を持つ



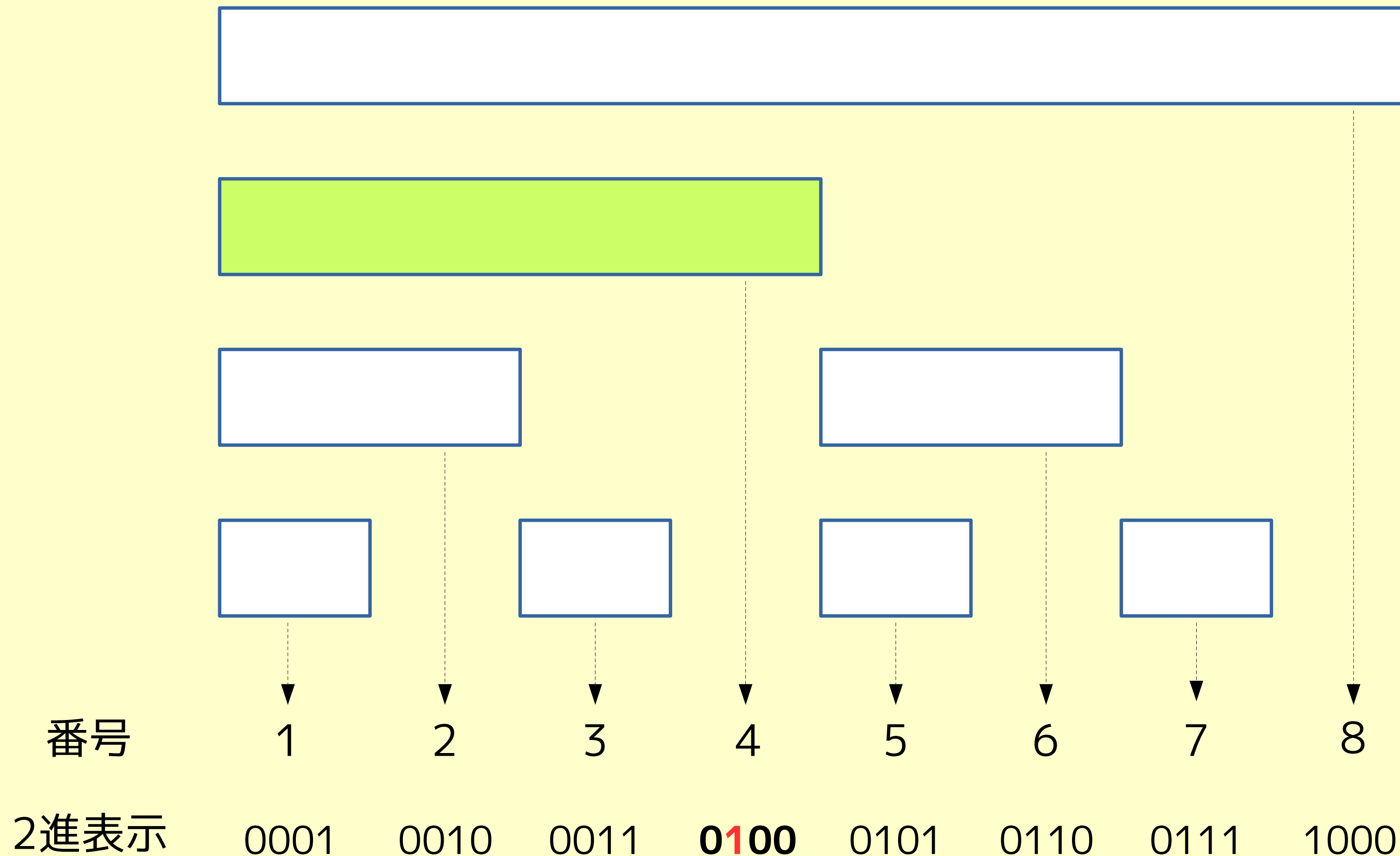
BIT のしくみ (3)

- 配列に、次のように対応する部分の和を持つ



BIT のしくみ (4)

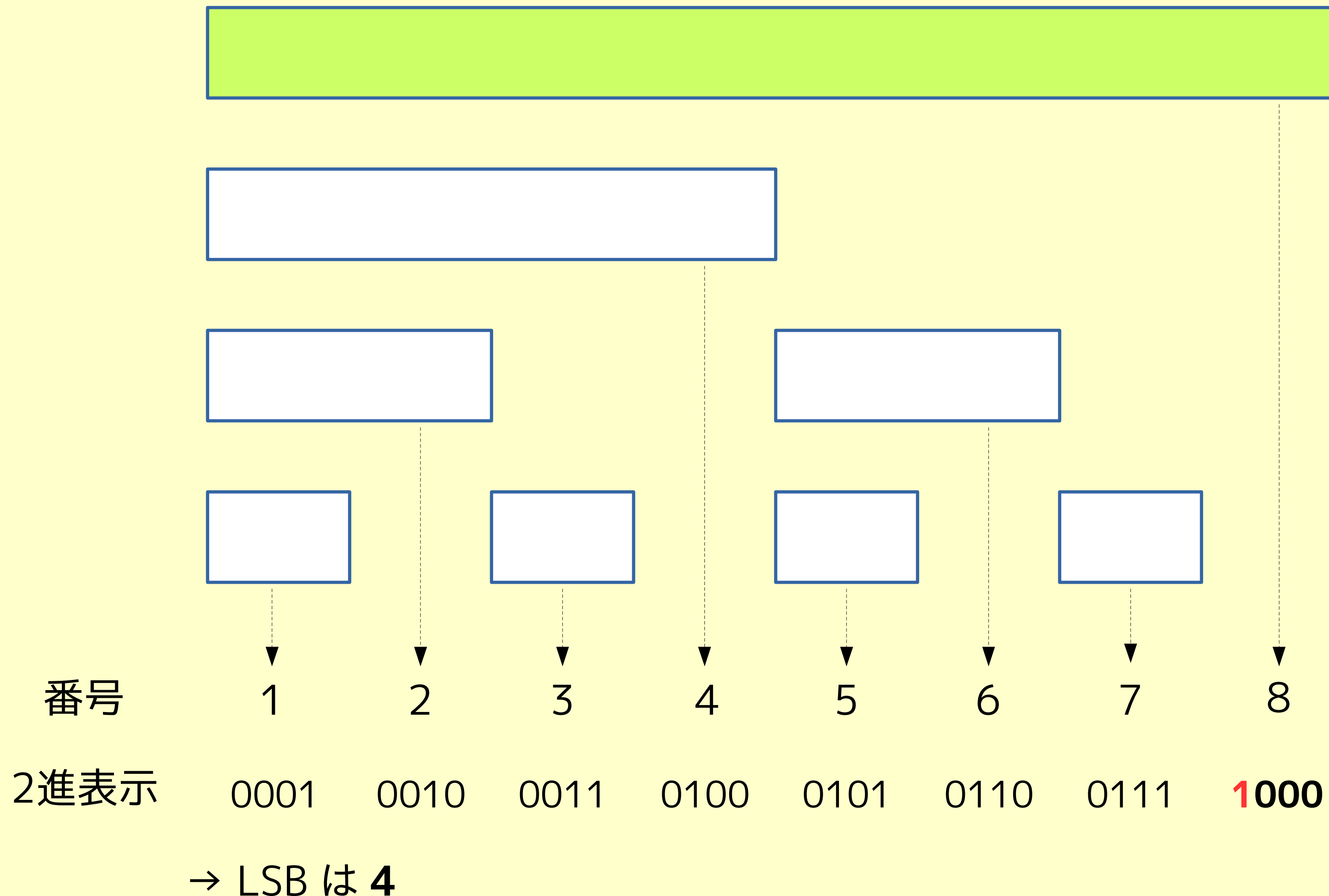
- 配列に、次のように対応する部分の和を持つ



→ LSB は **3**

BIT のしくみ (5)

- 配列に、次のように対応する部分の和を持つ



区間の長さと LSB

まとめると

- **区間の長さ**は、配列のインデックスを
  2進表示した時の**LSB**と関係がある！
  → 各操作を**ビット演算**で表現できる！

- でも、具体的にどうやったら実現できるの？

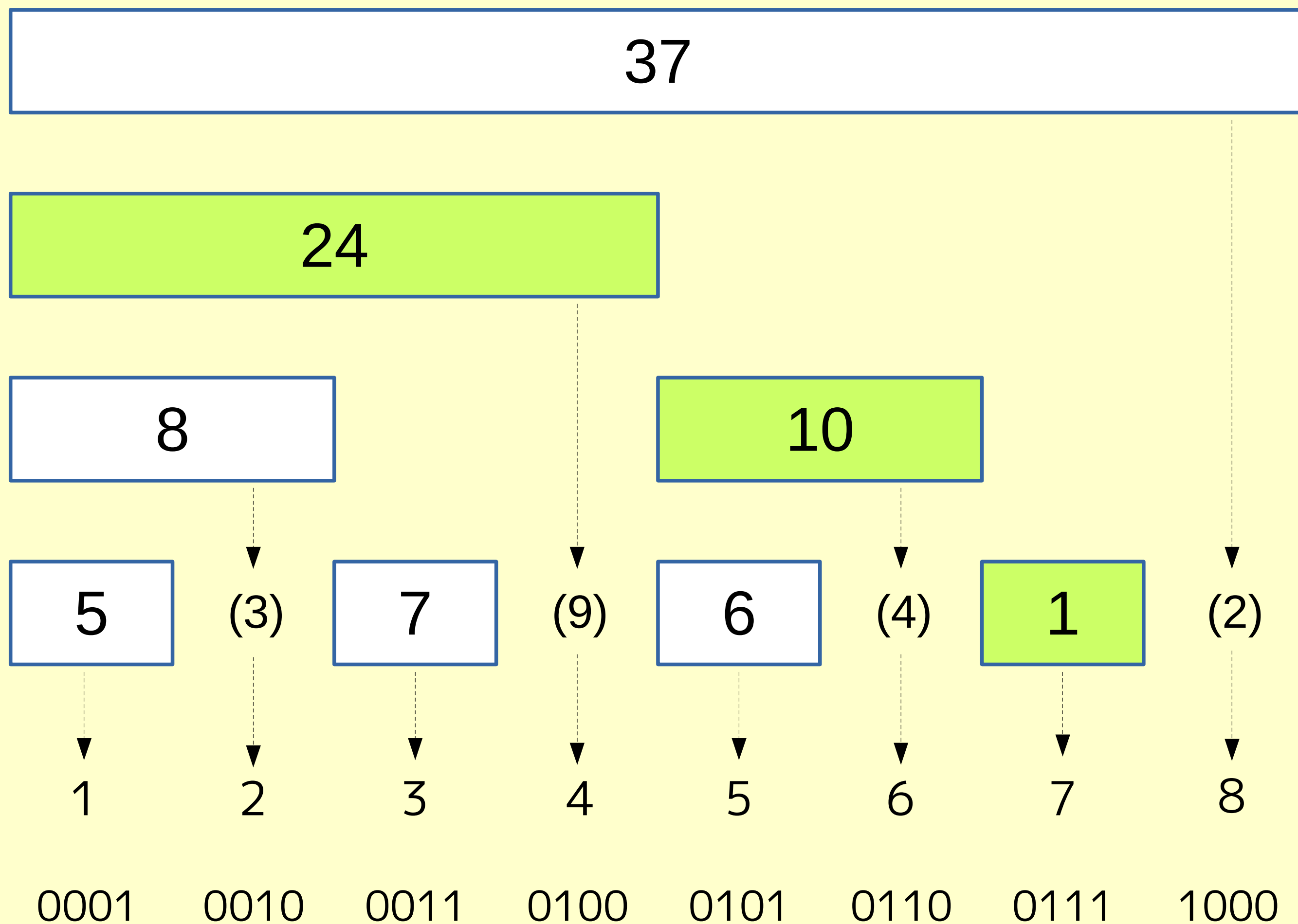
BIT での和の計算

- $a_1 + a_2 + \cdots + a_i$ の計算
  → 0になるまで**LSBを減算**しながら足していく！

(例) $a_1 + a_2 + \cdots + a_7$

i	2進表示	計
7	0111	1 (0 + 1)
6	0110	11 (1 + 10)
4	0100	35 (11 + 24)
0	0000	35

緑で塗った場所の値を
足しあわせれば良い！



BIT での値の更新

- a_i に x を足す

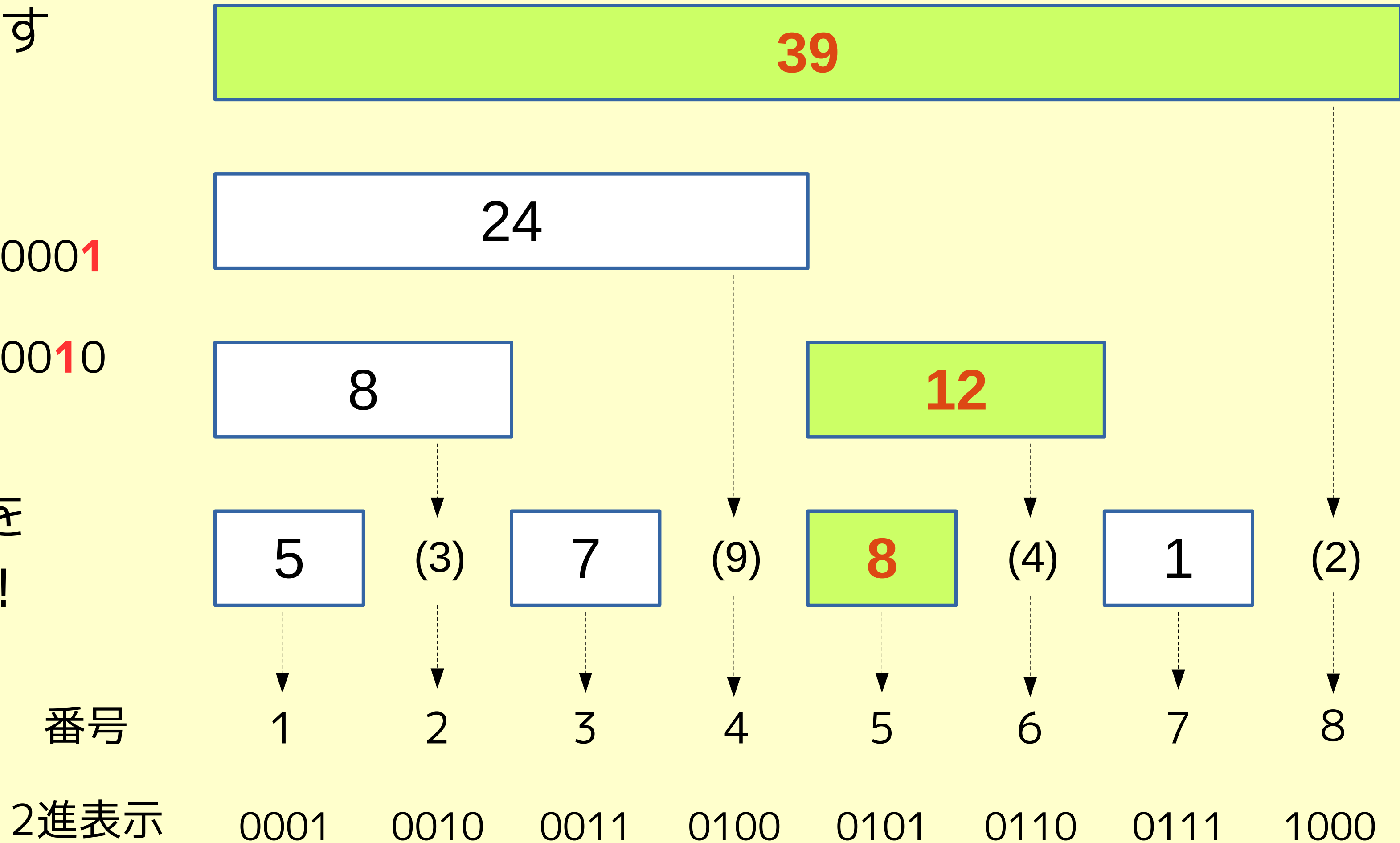
  → i に**LSBを加算**しながら更新していく！

(例) a_5 に 2 を足す

i	2進表示
5	0101
6	0110
8	1000

+ 0001

+ 0010



緑で塗った場所を
更新すれば良い！

BIT の特徴

- 要素数 N に対して **サイズ N の配列** で実現可能
- 和の計算 も 値の更新 も、
  ともに O(log n) の場所に対して行われるため、
  計算量は **O(log n)** で済む！
- 実装が比較的簡単

(素敵ですね。では実装しましょう。)

BIT の実装 (1)

先ほどの話より、ｉ の LSB を求めたい気持ちになる時が多々ある

- LSBは、**i & (- i)** で取得できる！

※この実装は両クエリで使ううえ、他の問題で役に立つ時もあるので覚えておきましょう

BIT の実装 (2)

- ソースコードぺたり (注意: 1-indexed です)

```cpp
// Binary Indexed Tree (BIT)
// Verified: AOJ DSL_2_B: Range Sum Query (intのみ)
template <typename T>
struct BIT{
private:
    vector<T> array;
    const int n;

public:
    // 初期化
    BIT(int _n) : array(_n + 1, 0), n(_n) {}

    // 1番目から i番目までの累積和を求める
    T sum(int i) {
        T s = 0;
        while(i > 0) {
            s += array[i];
            i -= i & -i;      // LSB 減算
        }
        return s;
    }
```

```cpp
    // [i, j] の要素の総和
    T sum(int i, int j) {
        T ret_i = sum(i-1);
        T ret_j = sum(j);
        return ret_j - ret_i;
    }

    // i 番目に 要素 x を追加
    void add(int i, T x) {
        while(i <= n) {
            array[i] += x;
            i += i & -i;      // LSB 加算
        }
    }
};
```

BIT の実装 (3)

実際に使ってみた

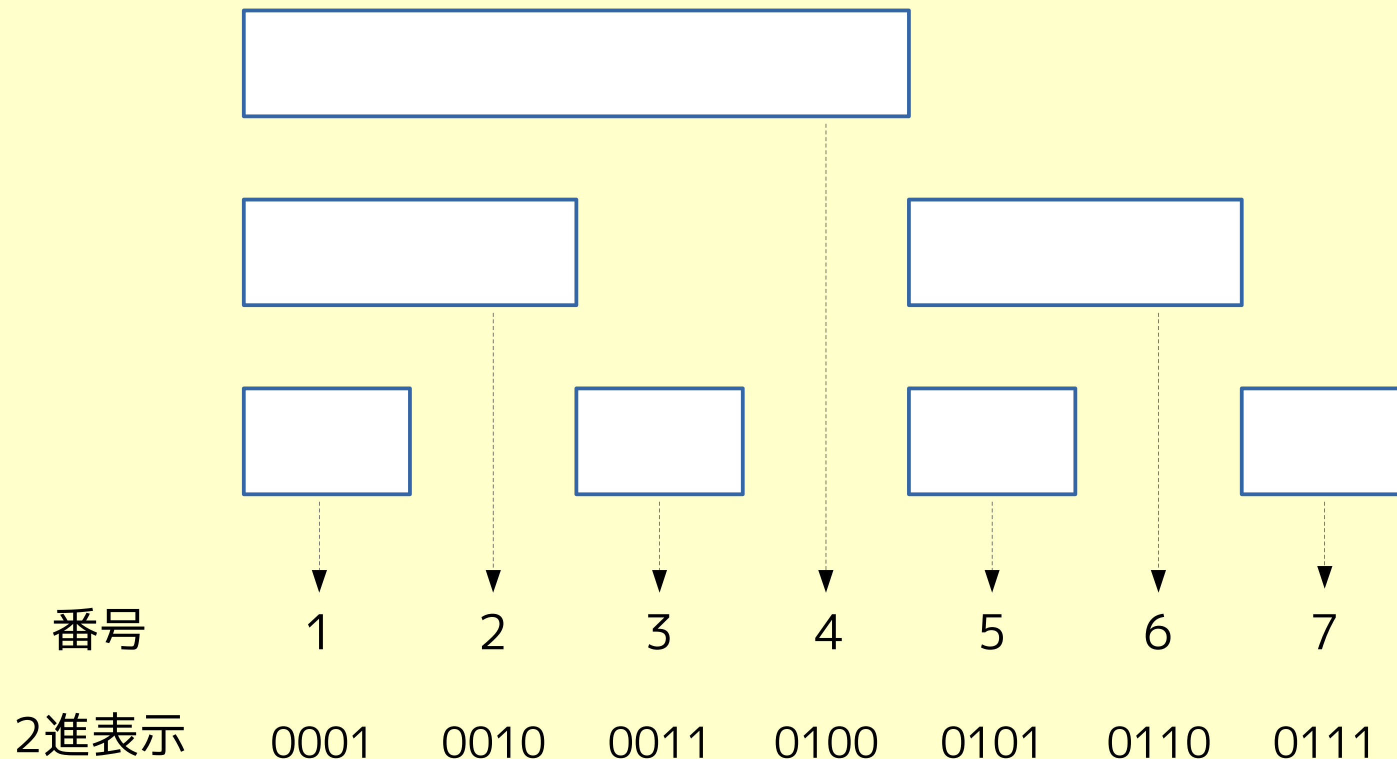
(AOJ DSL_2_B: Range Sum Query)

※問題文は典型すぎるので略 (気になったら調べてね)

```
signed main() {
    int n, q; cin >> n >> q;
    int c, x, y;
    BIT<int> bit(n);
    rep(i,0,q) {
        cin >> c >> x >> y;
        if(c == 0) bit.add(x, y);
        else cout << bit.sum(x, y) << endl;
    }
    return 0;
}
```

ちなみに

- 要素数 N は 2 のべき乗でなくても良いです
- たとえば下の BIT は要素数 7 ですが
  ちゃんと動いてくれます



2次元 BIT (1)

- 2次元への拡張は比較的簡単！
  (n × m) の BIT は、長さ (m + 1) の BIT を
  (n + 1) 個持てば良い

```
// 2次元BIT
// Verified: JOI 10 本選 Planetary Exploration
template <typename T>
struct twodimBIT{
private:
    vector< vector<T> > array;
    const int n;
    const int m;

public:
    // 初期化
    twodimBIT(int _n, int _m) : array(_n+1, vector<T>(_m+1, 0)), n(_n), m(_m) {}

    // (1, 1) から (x, y) までの累積和を求める
    T sum(int x, int y) {
        T s = 0;
        for(int i=x; i>0; i-=i&(-i))
            for(int j=y; j>0; j-=j&(-j))
                s += array[i][j];
        return s;
    }
```

2次元 BIT (2)

- ソースコードの続き
  範囲の和は、2次元累積和を取る時と同じように書けば良い！

```
    // [(x1, y1), (x2, y2)]の要素の総和
    T sum(int x1, int y1, int x2, int y2) {
        return sum(x2, y2) - sum(x1-1, y2) - sum(x2, y1-1) + sum(x1-1, y1-1);
    }

    // (x, y) に 要素 k を追加
    void add(int x, int y, T k) {
        for(int i=x; i<=n; i+=i&(-i))
            for(int j=y; j<=m; j+=j&(-j))
                array[i][j] += k;
    }
};
```

2次元 BITの問題 (1)

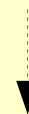
- Planetary Exploration (JOI 2010 本選)

  縦 M (1 ≦ M ≦ 1000), 横 N (1 ≦ N ≦ 1000) の盤面内の座標 (i, j) (1-indexed) に、"J", "O", "I" のいずれかの 1 文字が書き込まれている。領域の情報が、左上 (a, b), 右下 (c, d) として k 回 (1 ≦ k ≦ 100000) 与えられるので、各領域内に "J", "O", "I" がそれぞれいくつあるかを出力せよ。

(Sample input)

```
4 7
1
JIOJOIJ
IOJOIJO
JOIJOOI
OOJJIJO
2 2 3 6
```

	1	2	3	4	5	6	7
1	J	I	O	J	O	I	J
2	I	O	J	O	I	J	O
3	J	O	I	J	O	O	I
4	O	O	J	J	I	J	O



(Sample output)

```
3 5 2
```

2次元 BITの問題 (2)

- ソースコード (add するときのインデックス注意)

```
// 2次元BIT 略

signed main() {
    int m, n, k; cin >> m >> n >> k;
    twodimBIT<int> cntJ(m, n), cntO(m, n), cntI(m, n);
    rep(i,0,m) {
        string s; cin >> s;
        rep(j,0,n) {
            if(s[j] == 'J') cntJ.add(i+1, j+1, 1);
            if(s[j] == 'O') cntO.add(i+1, j+1, 1);
            if(s[j] == 'I') cntI.add(i+1, j+1, 1);
        }
    }
    rep(i,0,k) {
        int x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
        int ansJ = cntJ.sum(x1, y1, x2, y2);
        int ansO = cntO.sum(x1, y1, x2, y2);
        int ansI = cntI.sum(x1, y1, x2, y2);
        printf("%lld %lld %lld\n", ansJ, ansO, ansI);
    }
    return 0;
}
```

2次元 BITの問題 (3)

- ぶっちゃけ言うとこれは 2次元 BIT を使わなくても解ける (2次元累積和でも OK)
- ただし、途中で「ある座標の文字を変更する」などのクエリが来るような問題設定であれば2次元累積和で対応できない (TLE)
- つまり、2次元 BIT を使ったほうが応用が効くよ！ … ということで紹介しました。
- より高次元なBITも同じように書けるそうです (自分は書いたことない)

BIT の問題

2次元を扱った後に1次元を扱うガバガバ進行ですが、ここで蟻本の問題を取り上げます

- バブルソートの交換回数
- A Simple Problem with Integers (考察重い)

バブルソートの交換回数

1 〜 n の数を並び替えた数列 a_0, a_1, …, a_{n-1} が与えられます。この数列をバブルソートでソートするのに必要なスワップ回数を求めなさい。

(バブルソートとは、 $a_i > a_{i+1}$ であるような i を見つけてスワップすることを繰り返すアルゴリズムのことを言います)

バブルソートの交換回数

- $i < j, a_i \leqq a_j$ となるような (i, j) の組の個数は BIT を用いて高速に計算できる
- $i < j, a_i > a_j$ となるような (i, j) の組の個数も、上の結果から求められる

- 結構考察がこんがらがるので注意？

バブルソートの交換回数

- 値の範囲 1 ～ n の BIT を用いて、
  j = 0, 1, 2, …, n-1 の順に以下を行えば良い！

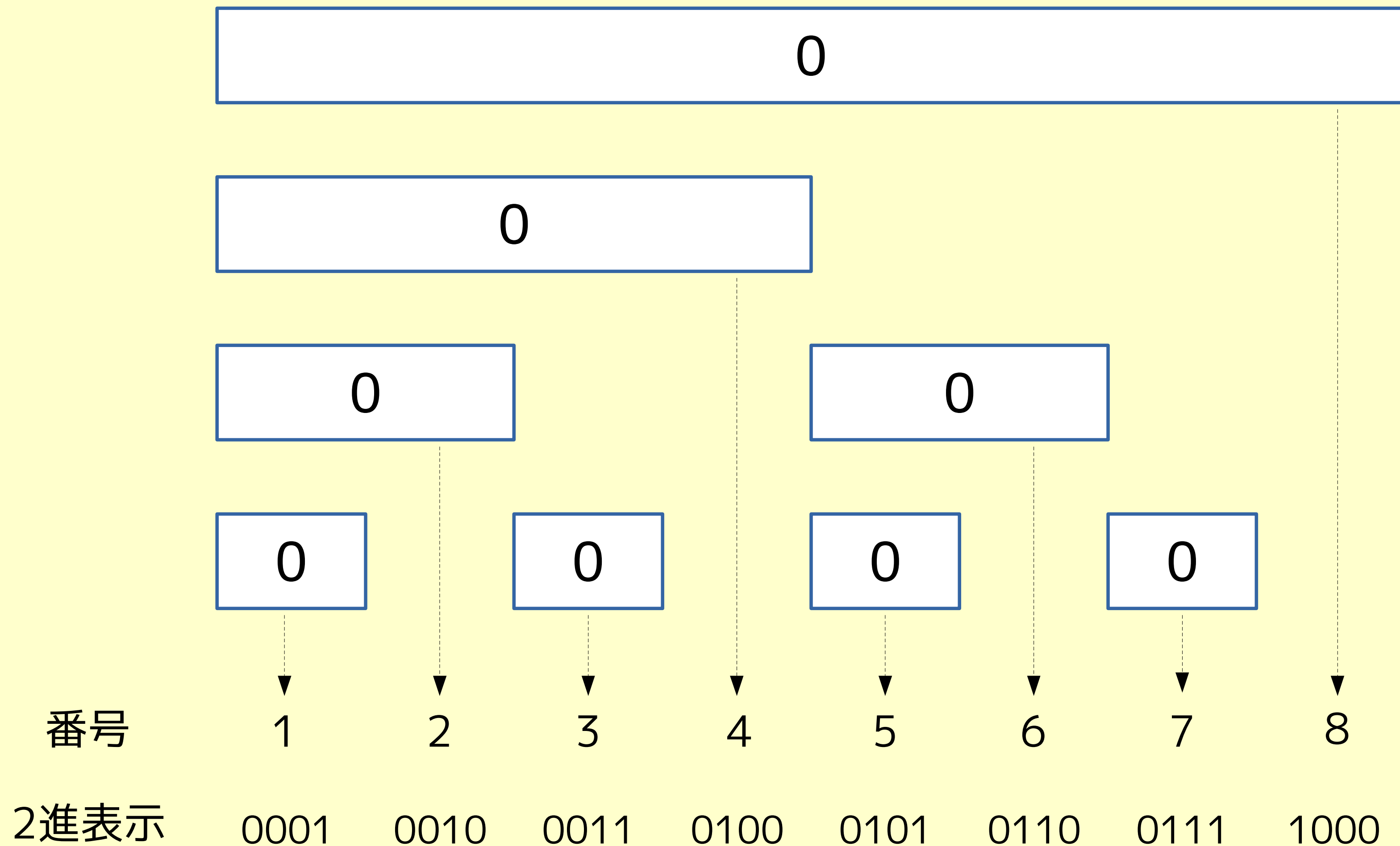
- j – (BIT の a_j までの和) を答えに加える
- BIT の場所 a_j に 1 を加算する

- 実際、どうなるの？
- → (BIT の a_j までの和) = ($i < j$, $a_i \leqq a_j$ なる i の個数) になる。実際にやってみよう

バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6}

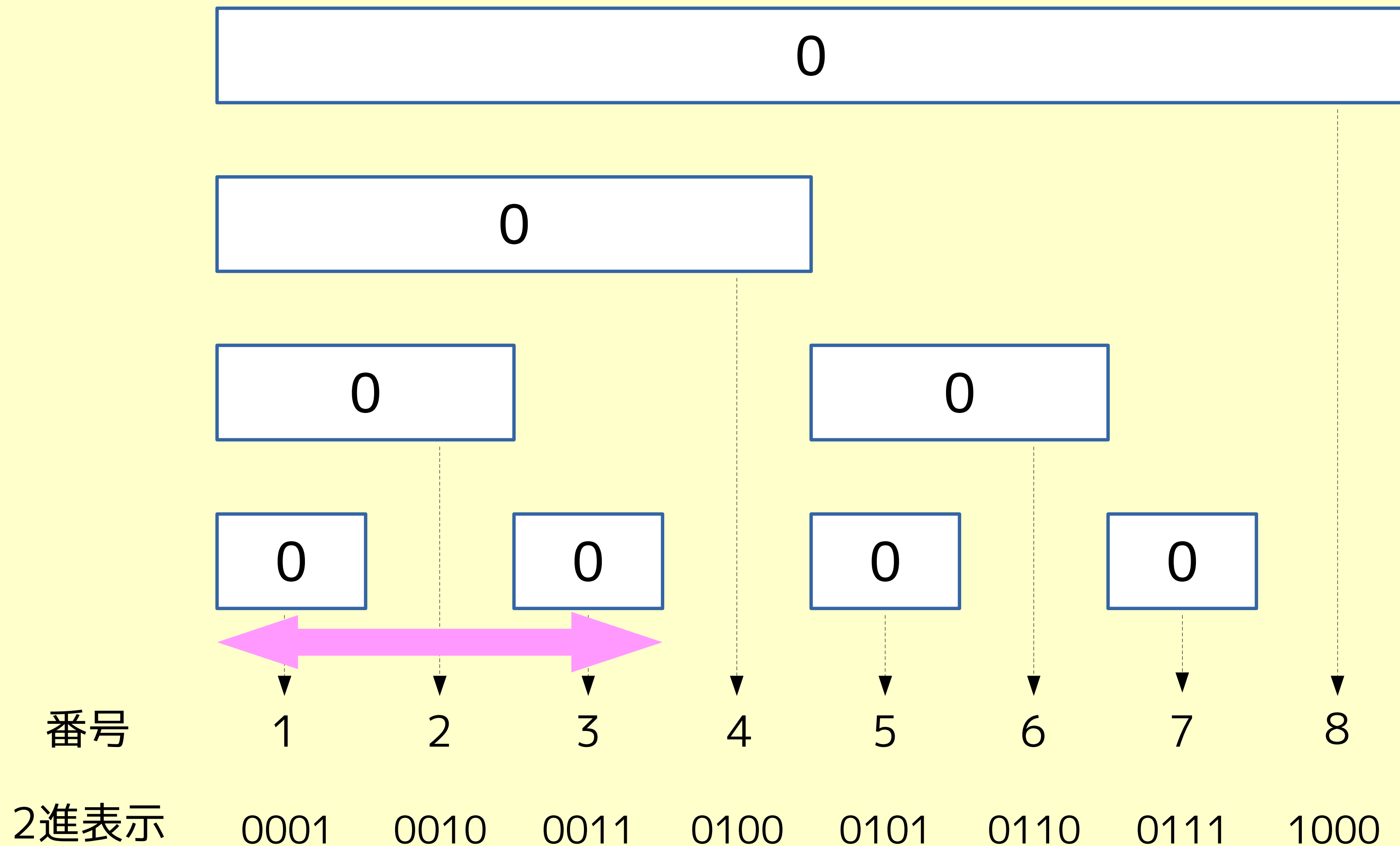
ans = 0



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 0
  - 0 – (BITのa_0 = 3 までの和 = 0) = 0 を 答えに加える

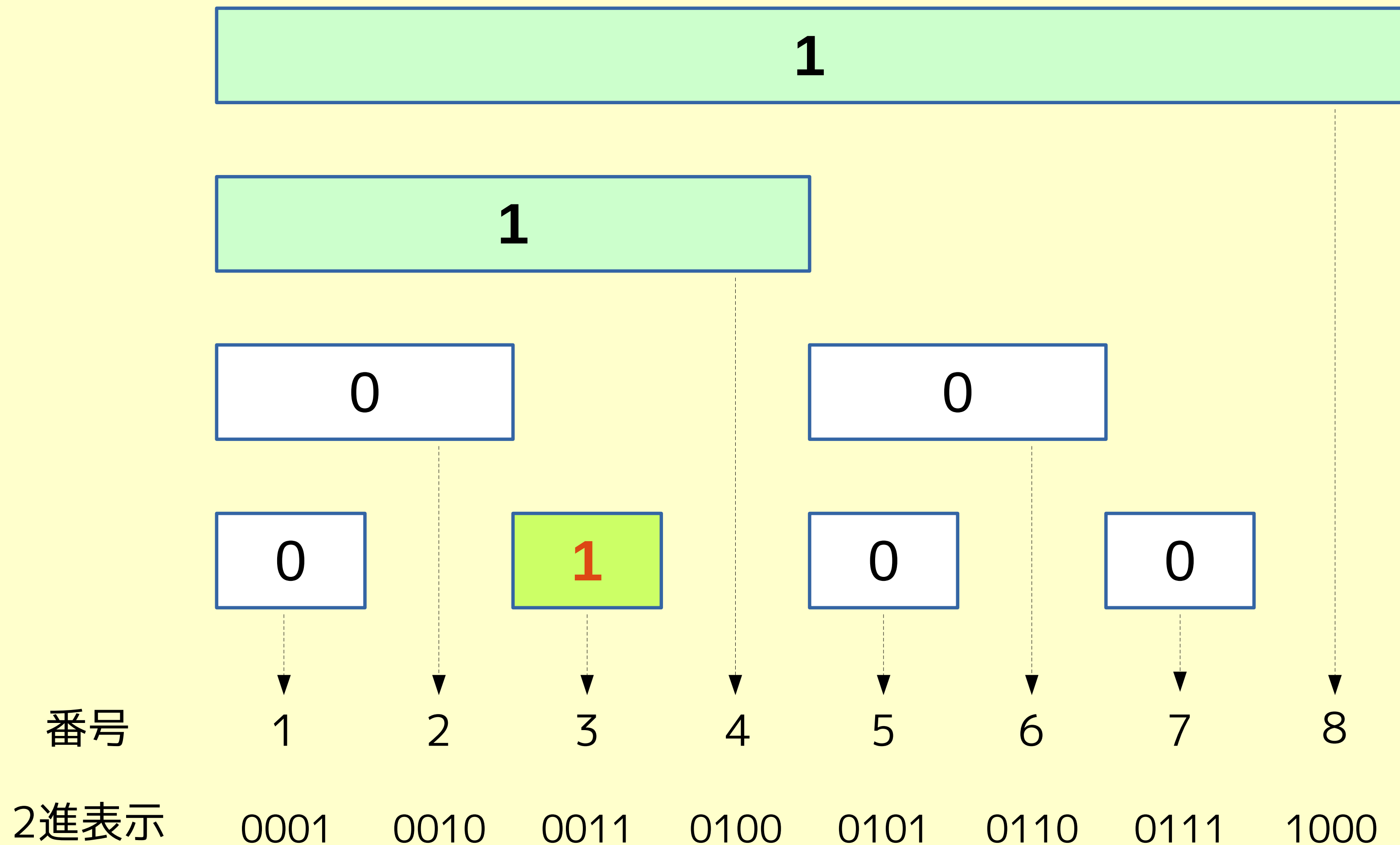
ans = 0



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 0
  - BIT の場所 a_0 = 3 に 1 を加える

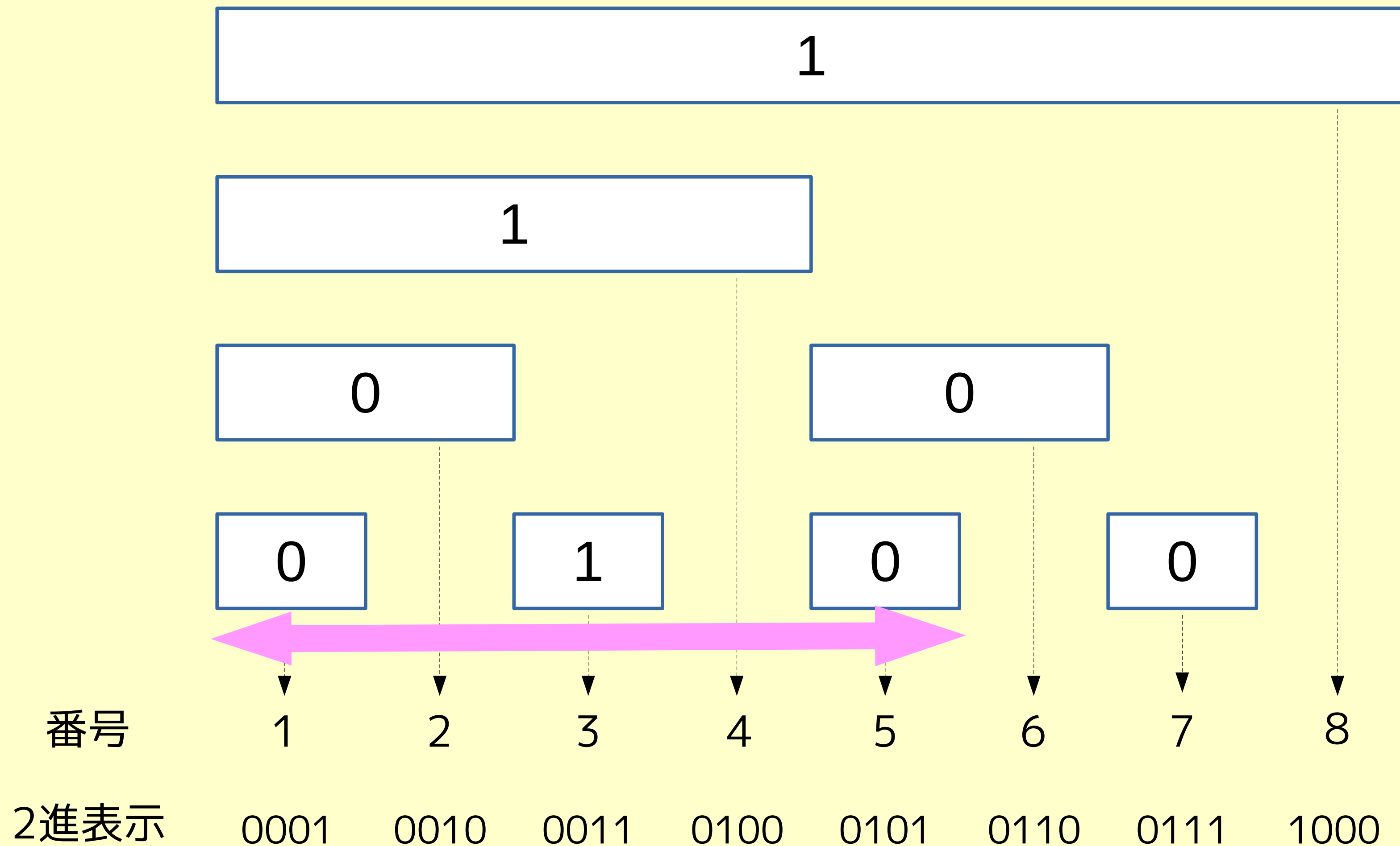
ans = 0



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 1
  - 1 – (BITのa_1 = 5 までの和 = 1) = 0 を 答えに加える

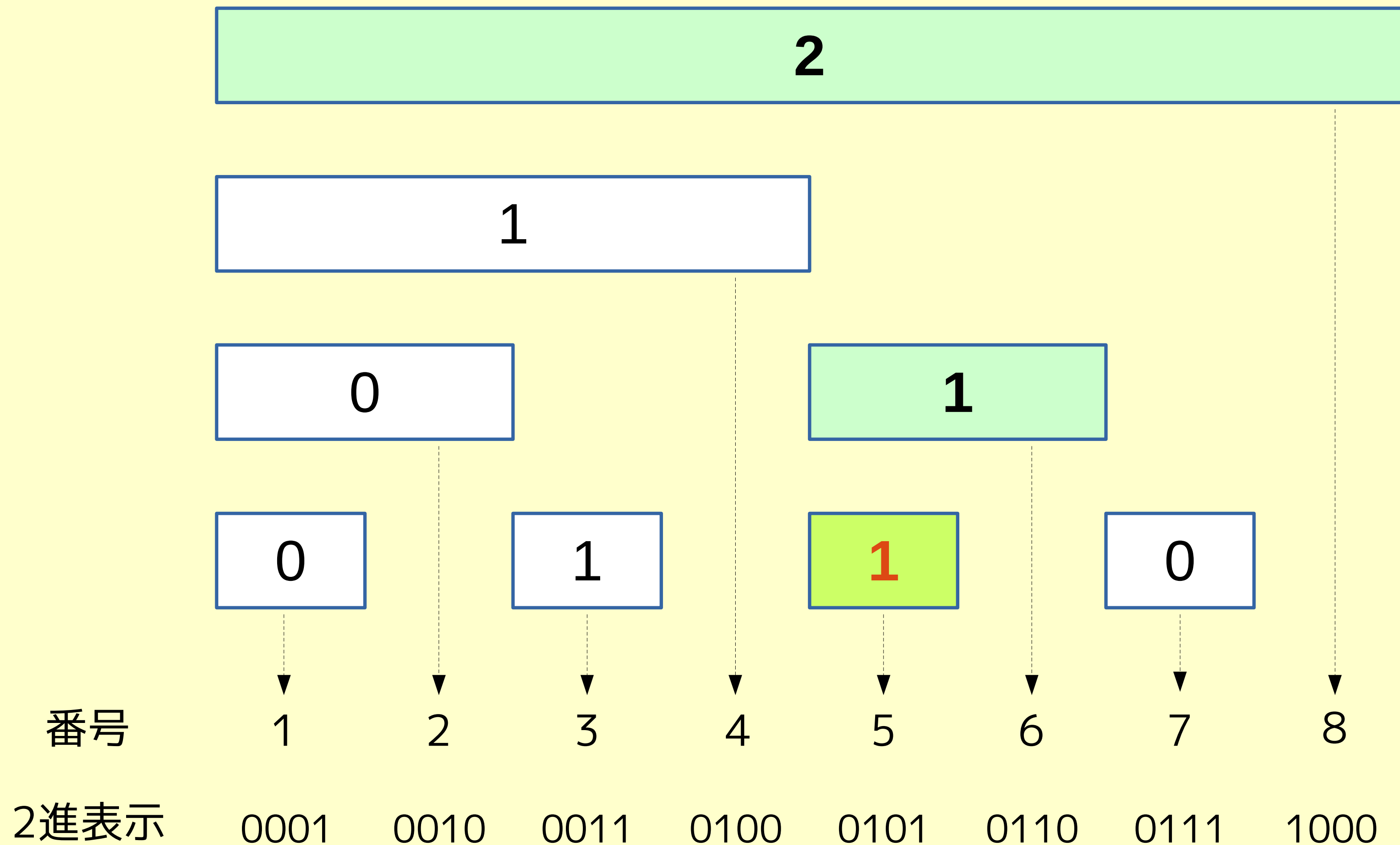
ans = 0



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 1
  - BIT の場所 a_1 = 5 に 1 を加える

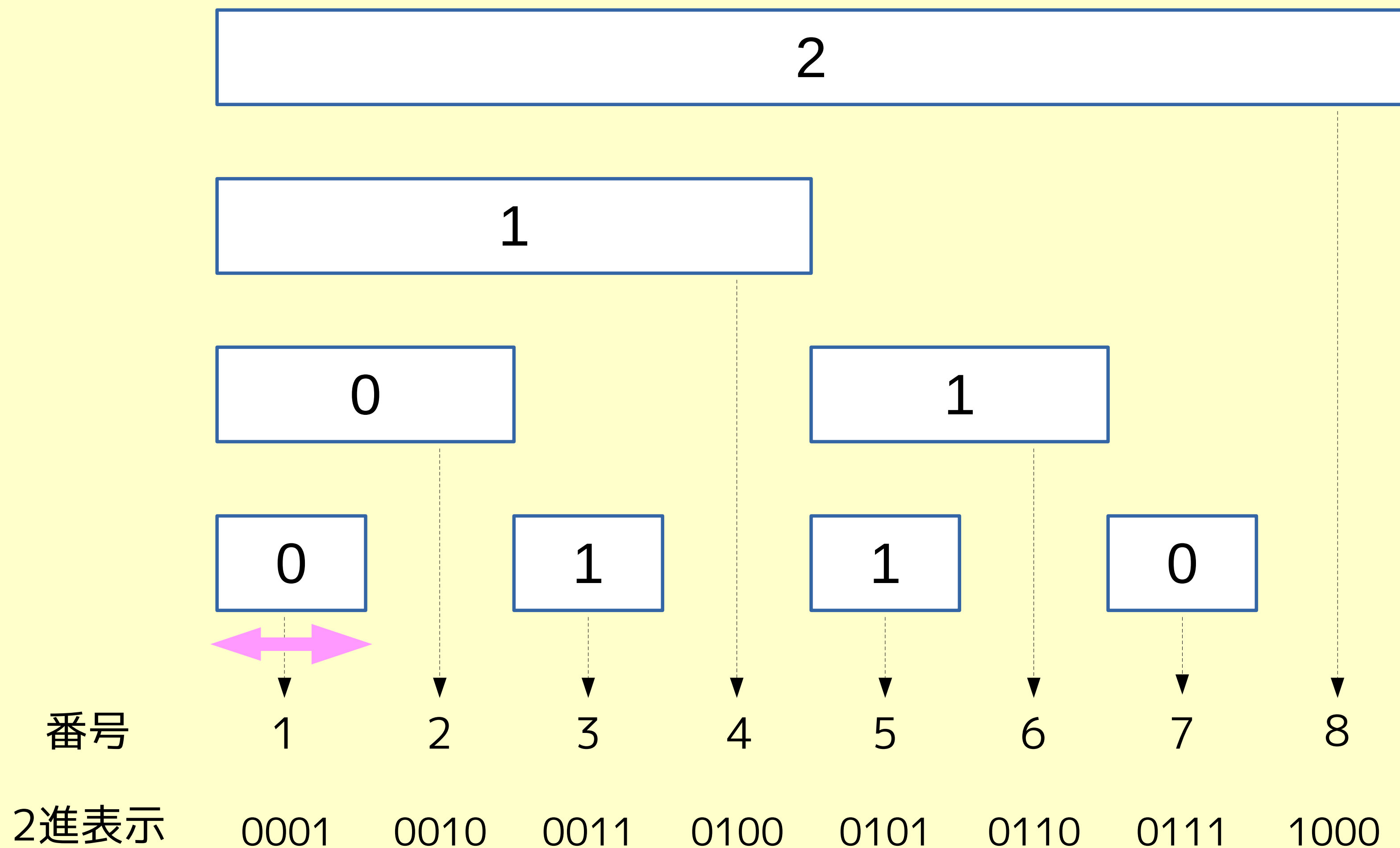
ans = 0



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 2
  - 2 – (BITのa_2 = 1 までの和 = 0) = 2 を 答えに加える

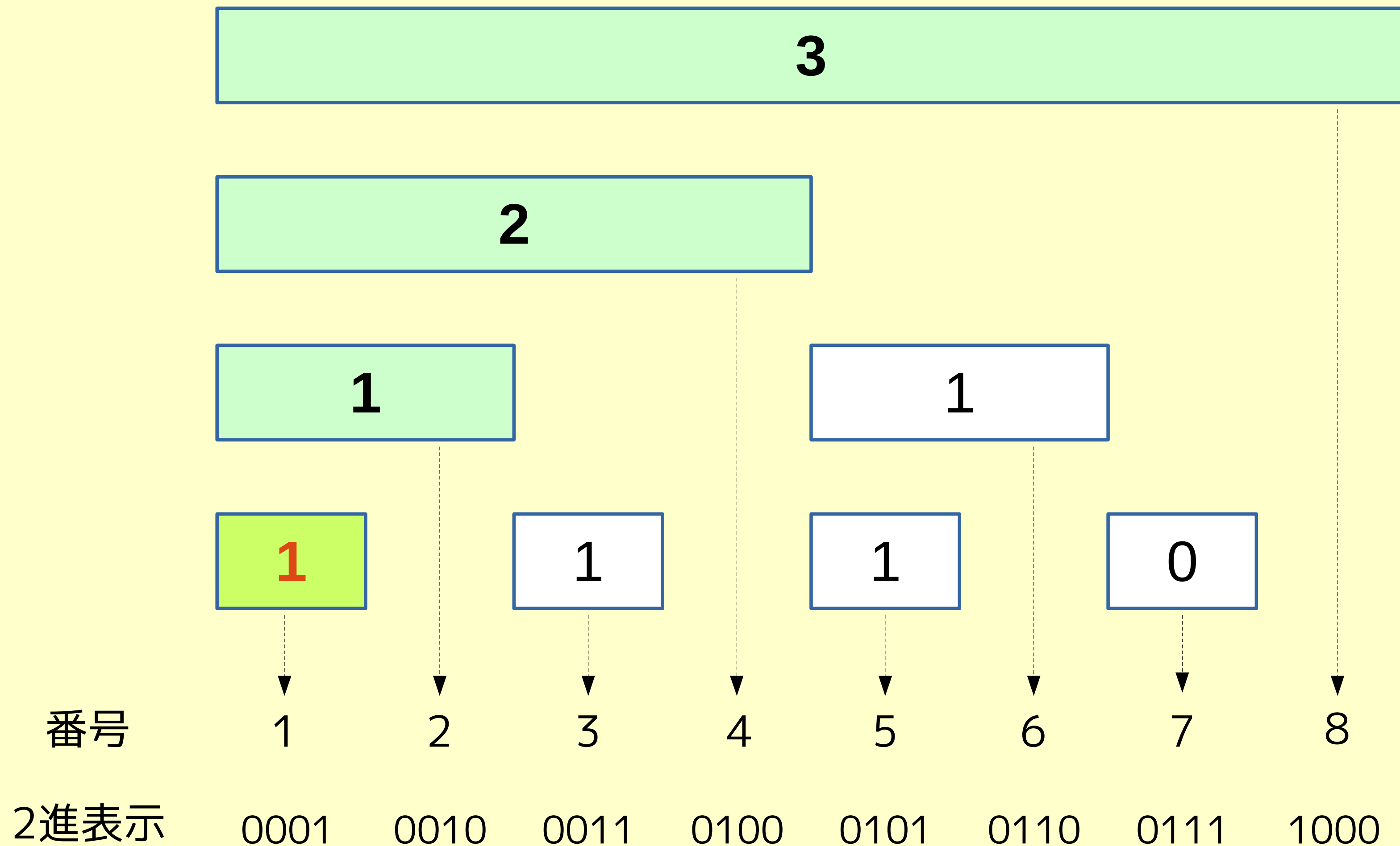
ans = 2



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 2
  - BIT の場所 a_2 = 1 に 1 を加える

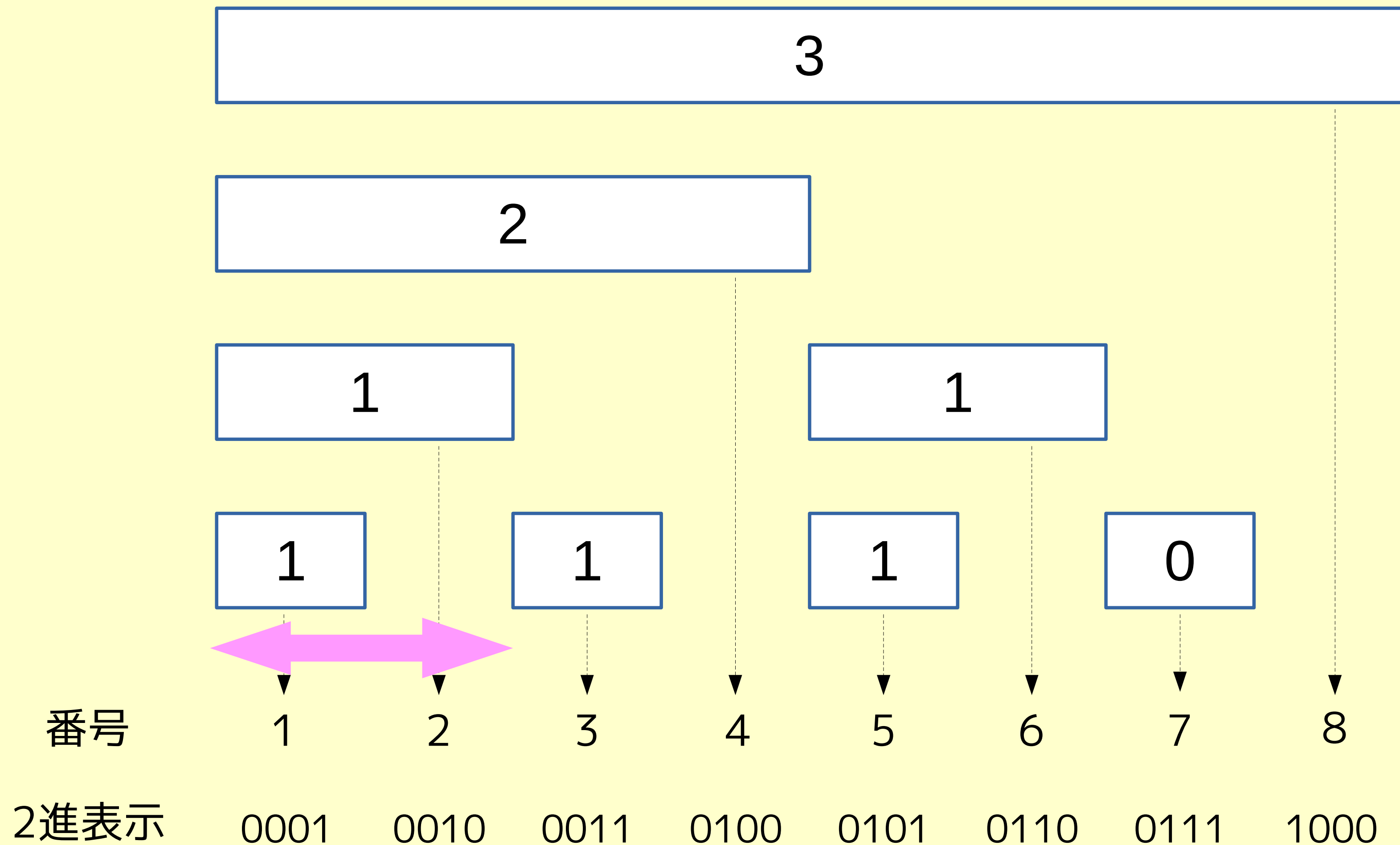
ans = 2



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 3
  - 3 – (BITのa_3 = 2 までの和 = 1) = 2 を 答えに加える

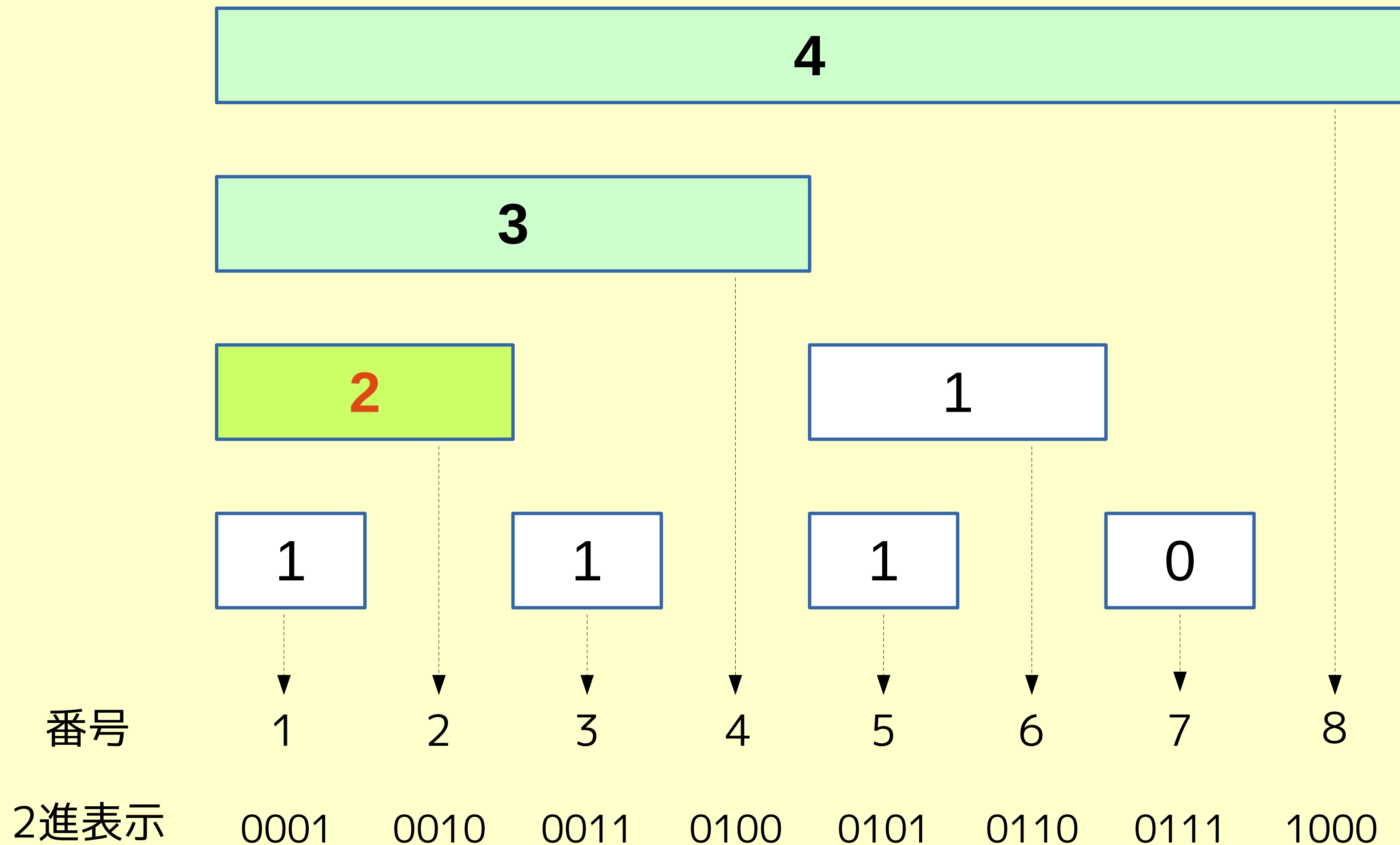
ans = 4



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 3
  - BIT の場所 a_3 = 2 に 1 を加える

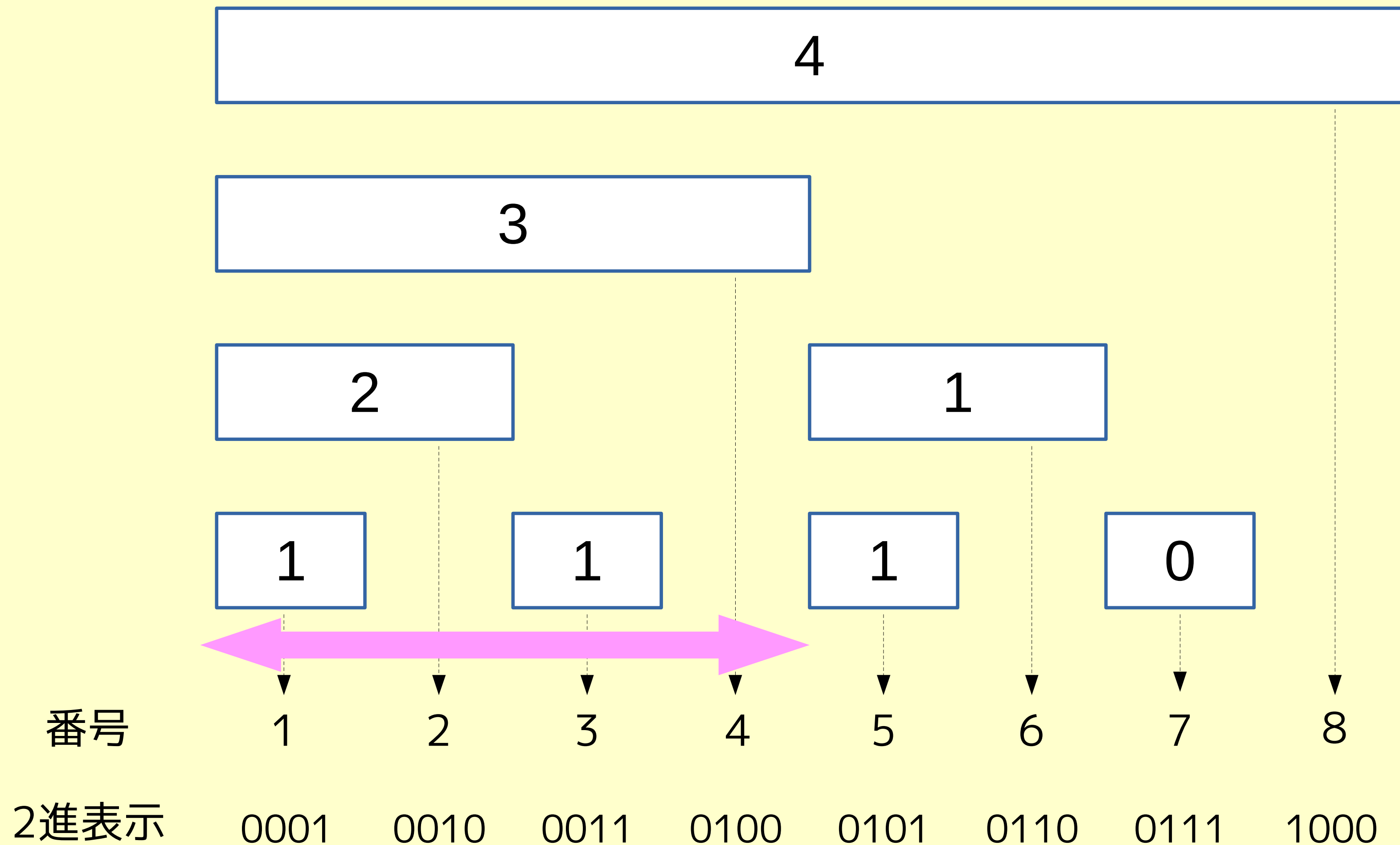
ans = 4



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 4
  - 4 – (BITのa_4 = 4 までの和 = 3) = 1 を 答えに加える

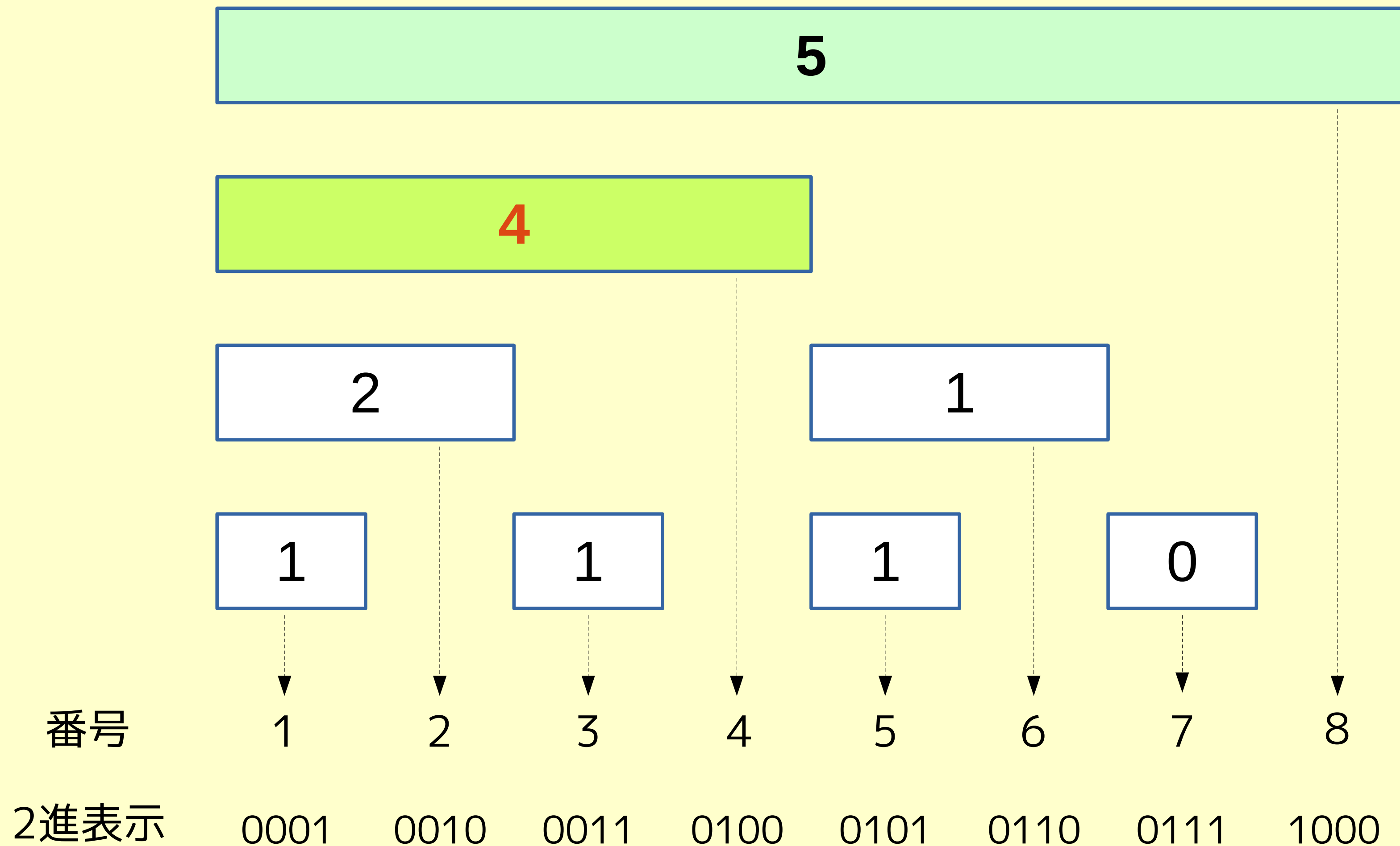
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 4
  - BIT の場所 a_4 = 4 に 1 を加える

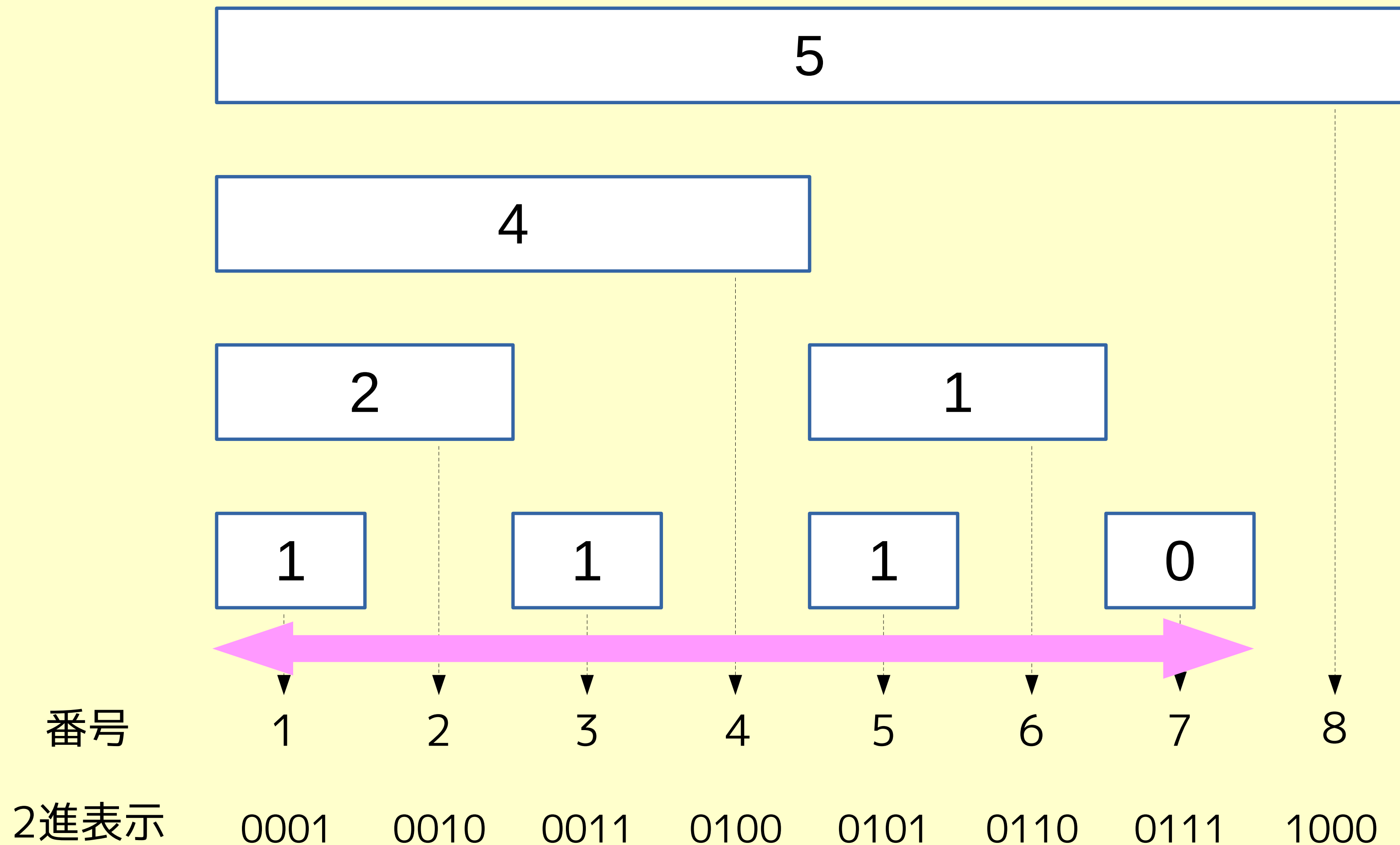
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 5
  - 5 – (BITのa_5 = 7 までの和 = 5) = 0 を 答えに加える

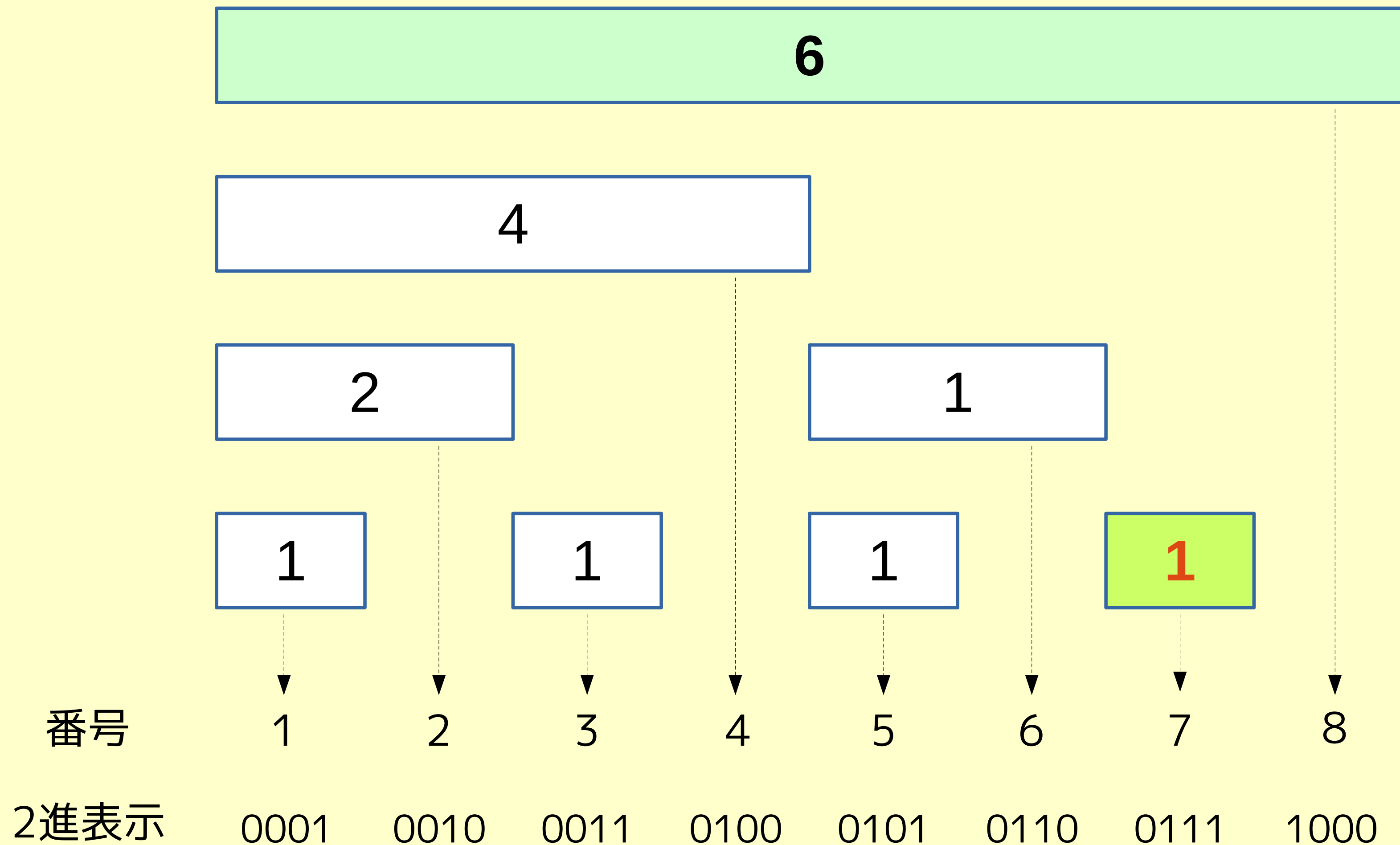
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 5
  - BIT の場所 a_5 = 7 に 1 を加える

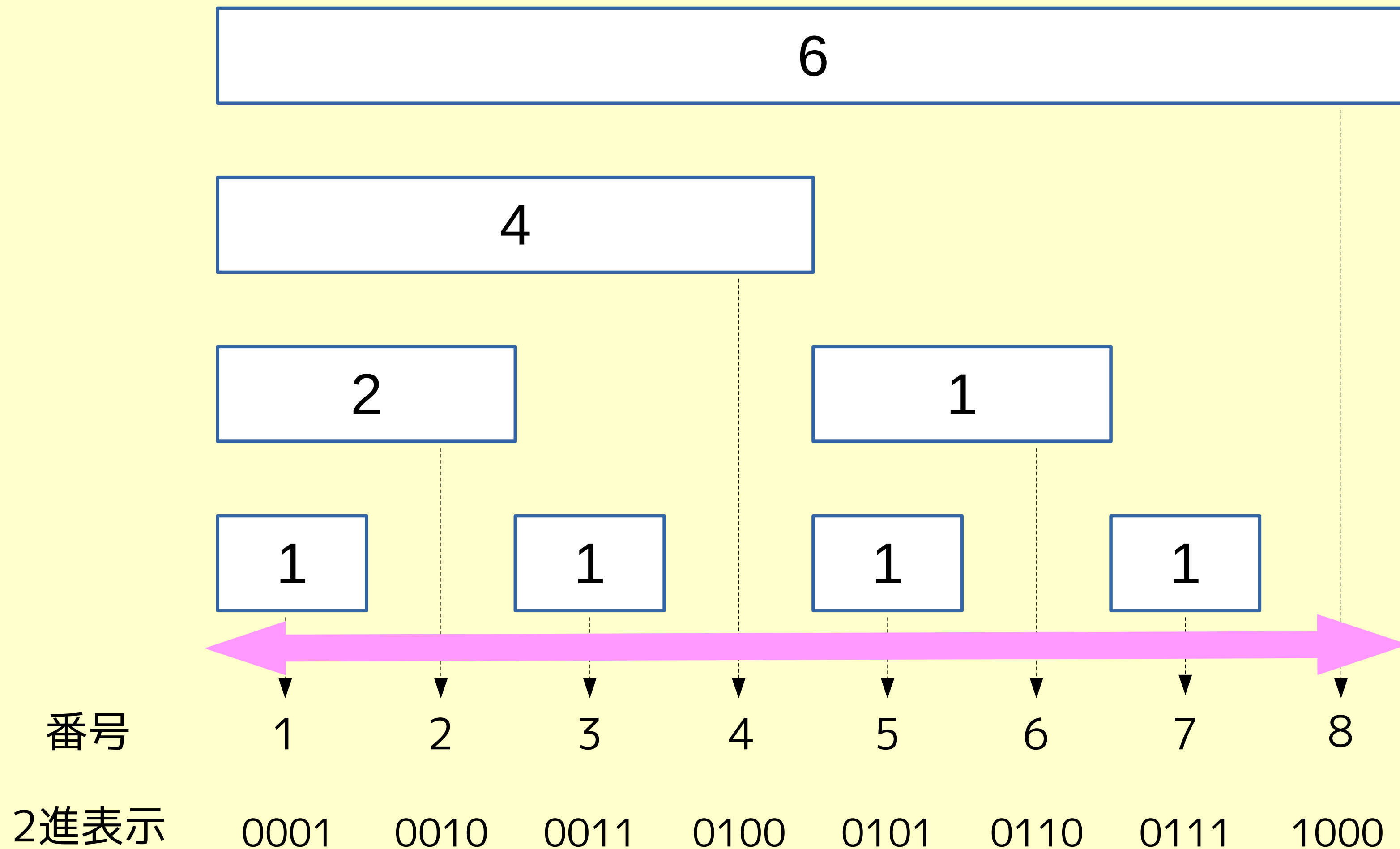
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 6
  - 6 – (BITのa_6 = 8までの和 = 6) = 0 を 答えに加える

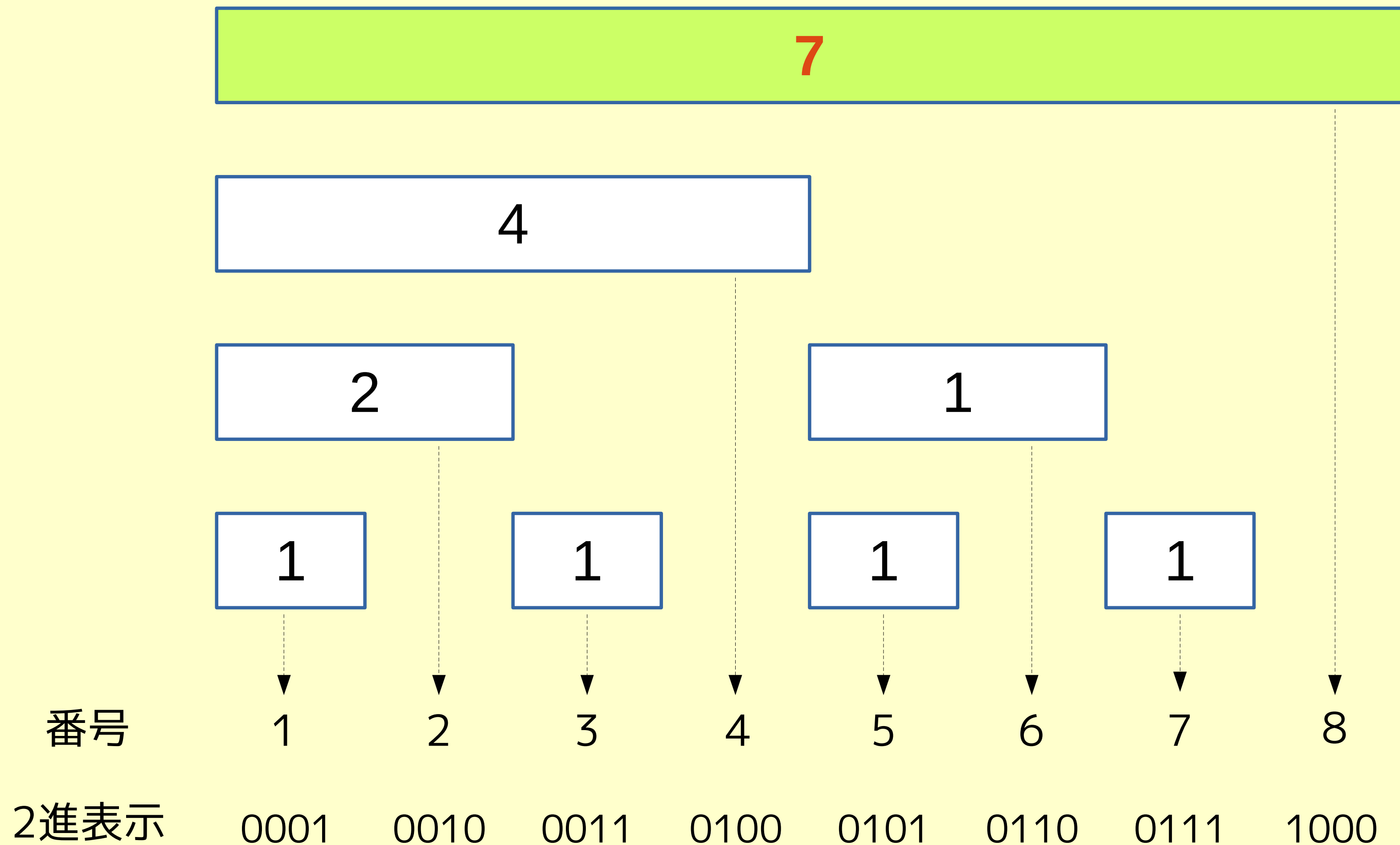
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 6
  - BIT の場所 a_6 = 8 に 1 を加える

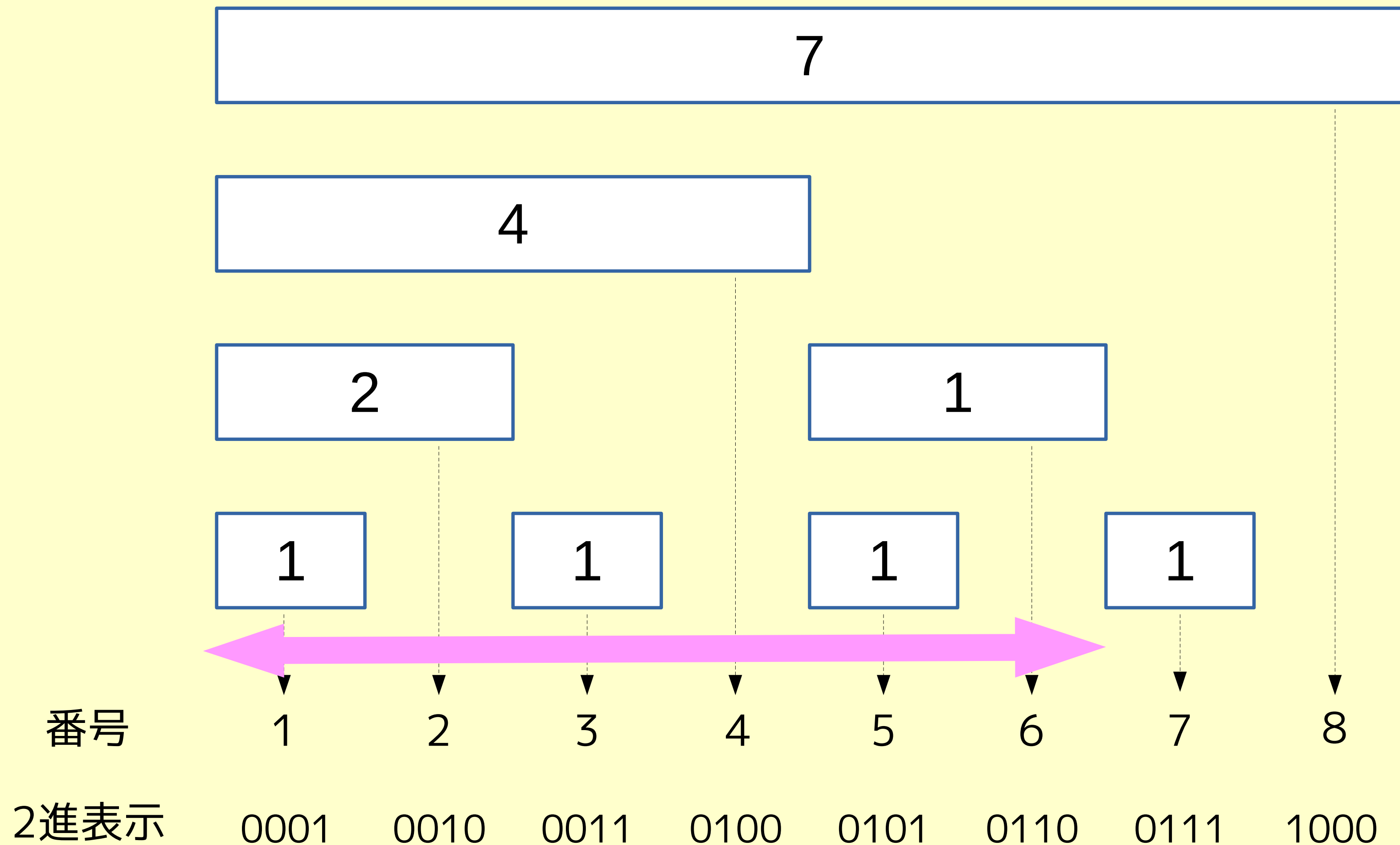
ans = 5



バブルソートの交換回数

- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 7
  - 7 – (BITのa_7 = 6 までの和 = 5) = 2 を 答えに加える

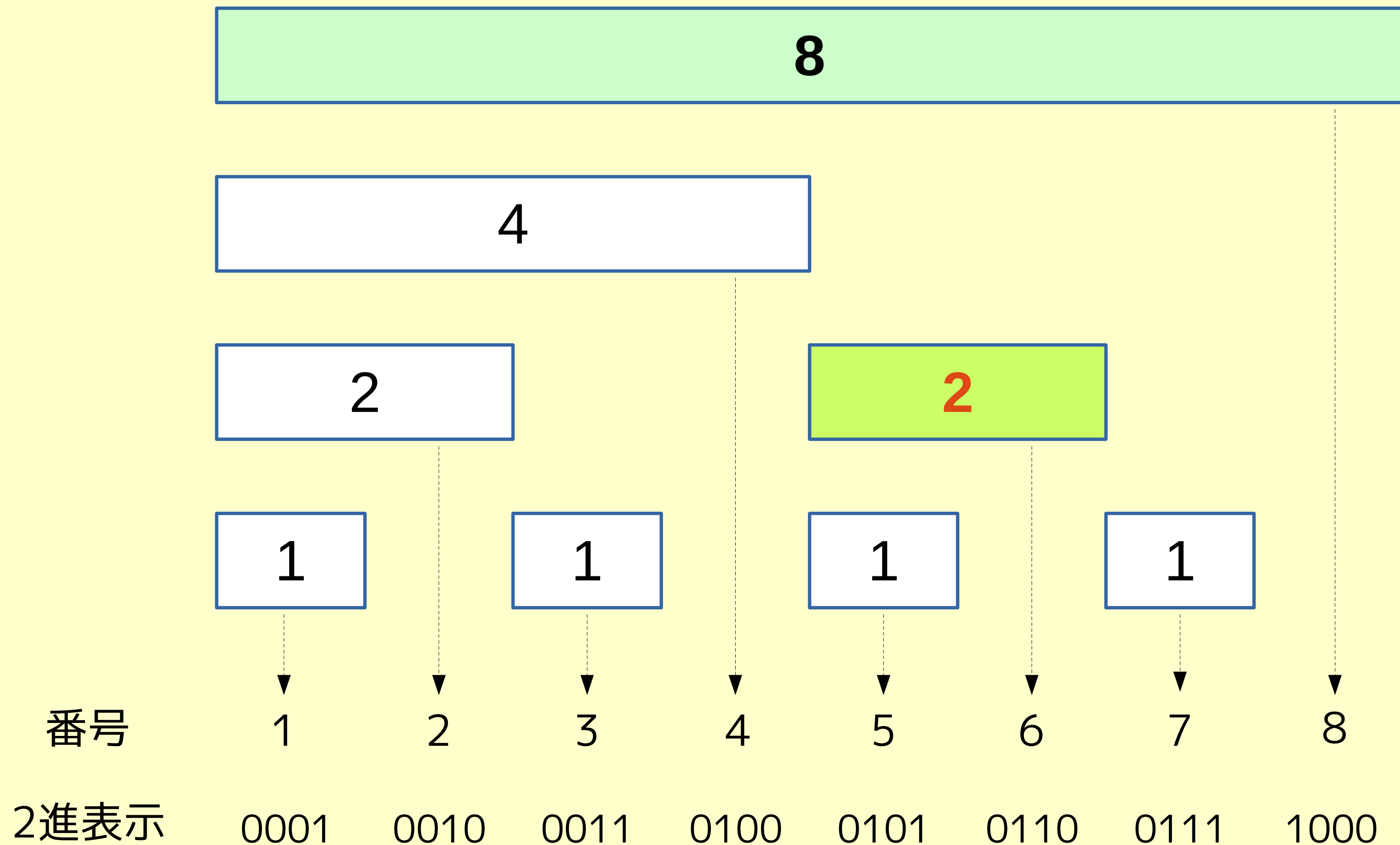
ans = 7



バブルソートの交換回数

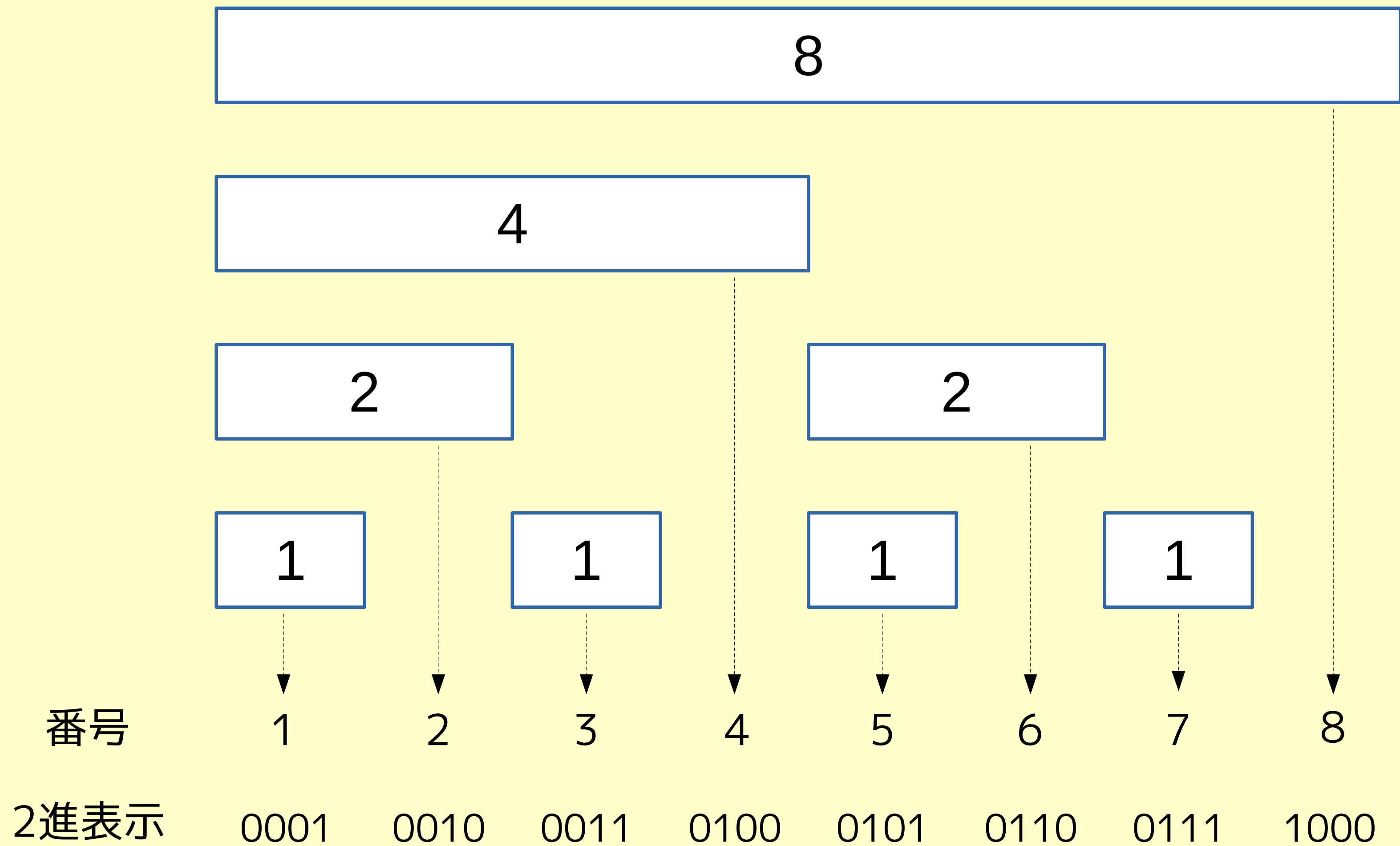
- 例 → {3, 5, 1, 2, 4, 7, 8, 6} , j = 7
  - BIT の場所 a_7 = 6 に 1 を加える

ans = 7



バブルソートの交換回数

- end
  - **ans = 7**



バブルソートの交換回数

- 実装は簡単です

```cpp
// BIT 略
// Verified: ALDS1_5_D (反転数)
signed main() {
    int n; cin >> n;
    BIT<int> b(n);
    vector<int> a(n);
    map<int, int> m;
    rep(i,0,n) {
        cin >> a[i];
        m[a[i]];
    }

    int newnum = 1;
    for(auto &x : m) x.second = newnum++;
    rep(i,0,n) a[i] = m[a[i]];

    int ans = 0;
    rep(j,0,n) {
        ans += j - b.sum(a[j]);
        b.add(a[j], 1);
    }
    cout << ans << endl;
    return 0;
}
```

A Simple Problem with Integers

数列 $A_1, A_2, \cdots, A_n$ と、 Q 個のクエリが与えられます。クエリを順次処理してください。クエリは次の 2 種類です。

- l, r, x が与えられるので、$A_l, A_{l+1}, \cdots A_r$ に x を加える
- l, r が与えられるので、$A_l, A_{l+1}, \cdots A_r$ の和を求める
- $1 \leqq N, Q \leqq 100000$

A Simple Problem with Integers

お、これ BIT で解けるんじゃね … ？ (適当)

- l, r, x が与えられるので、A_l, A_{l+1}, … A_r に x を加える
- l, r が与えられるので、A_l, A_{l+1}, … A_r の和を求める
- 1 ≦ N, Q ≦ 100000

A Simple Problem with Integers

この部分が無理

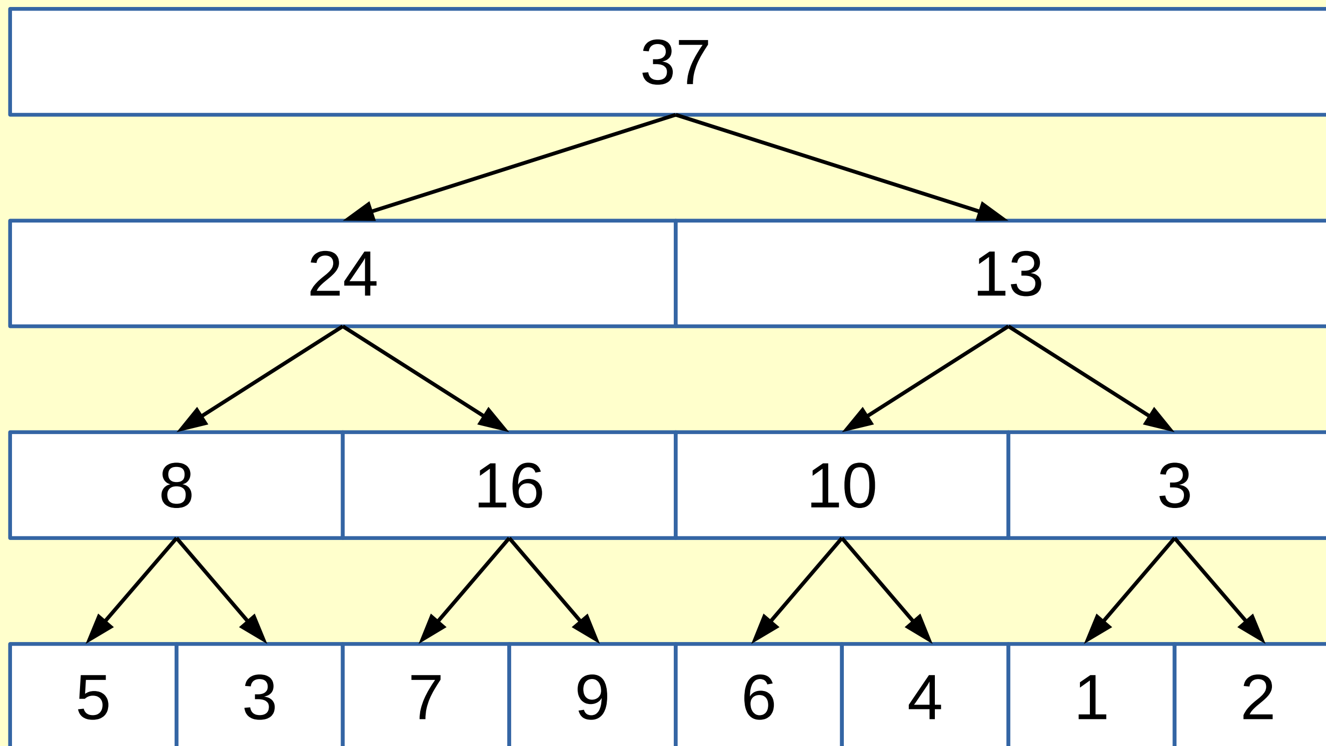
- l, r, x が与えられるので、**A_l, A_{l+1}, … A_r に x を加える**
- l, r が与えられるので、A_l, A_{l+1}, … A_r の和を求める
- 1 ≦ N, Q ≦ 100000

A Simple Problem with Integers

- 区間内の全ての要素に x を加えるクエリに対応するにはどうすればよいか？
  → BIT を 2 つ用意すれば対応できる！
- なぜそれが言えるかを、セグメント木の場合から考えていこう

A Simple Problem with Integers

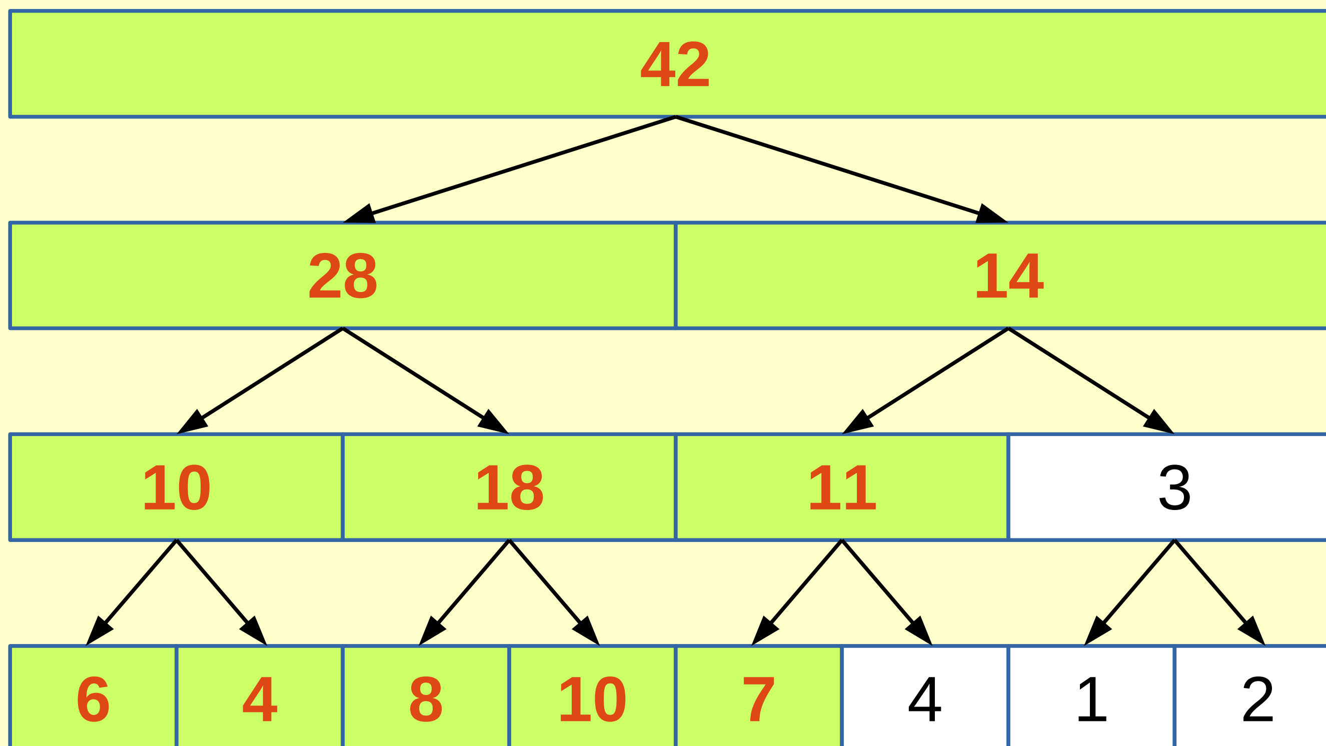
- セグメント木 (再掲)



各節点は、対応する区間の和を持っている

A Simple Problem with Integers

- 1 ～ 5 番目に 1 を加えた時に更新が必要な節点



多すぎ！

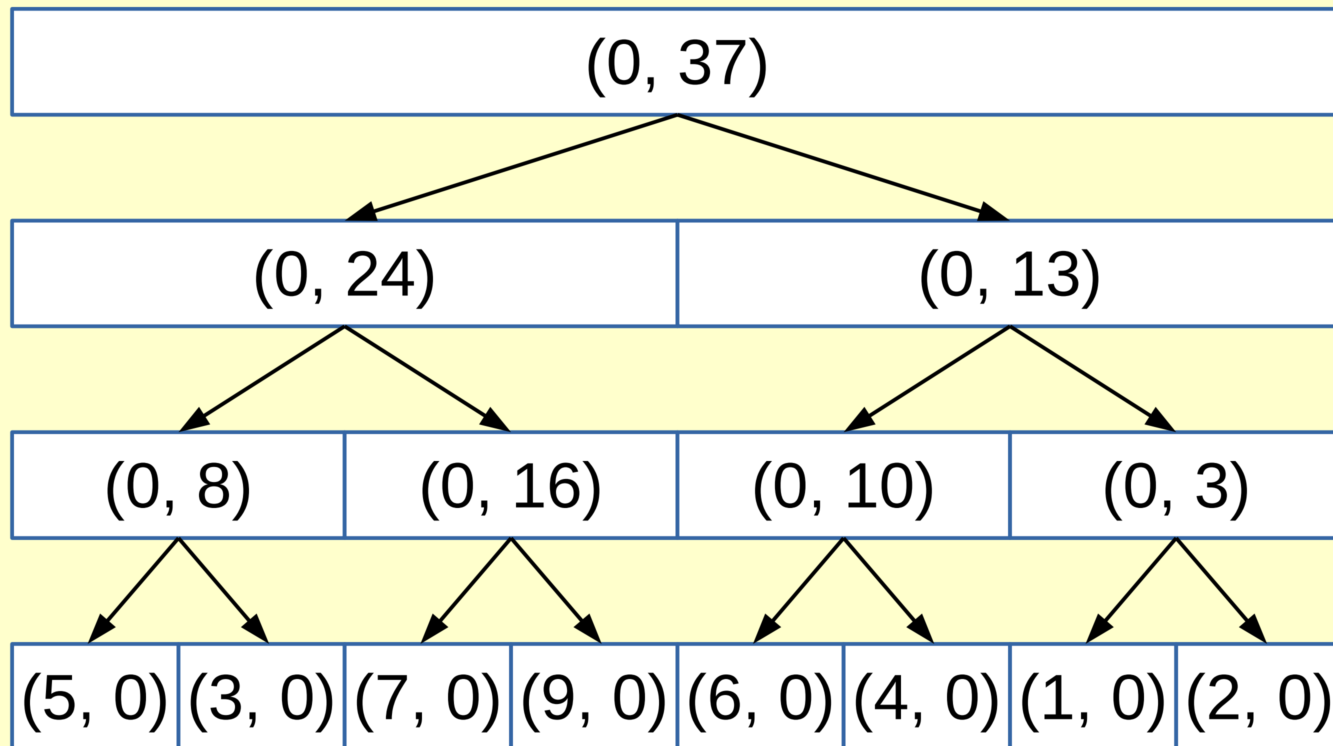
A Simple Problem with Integers

各節点が、次の2つの情報を持つように改良してみる

- その節点の区間全体に一様に加えられた値
- その節点の区間に一様でなく加えられた値の和

A Simple Problem with Integers

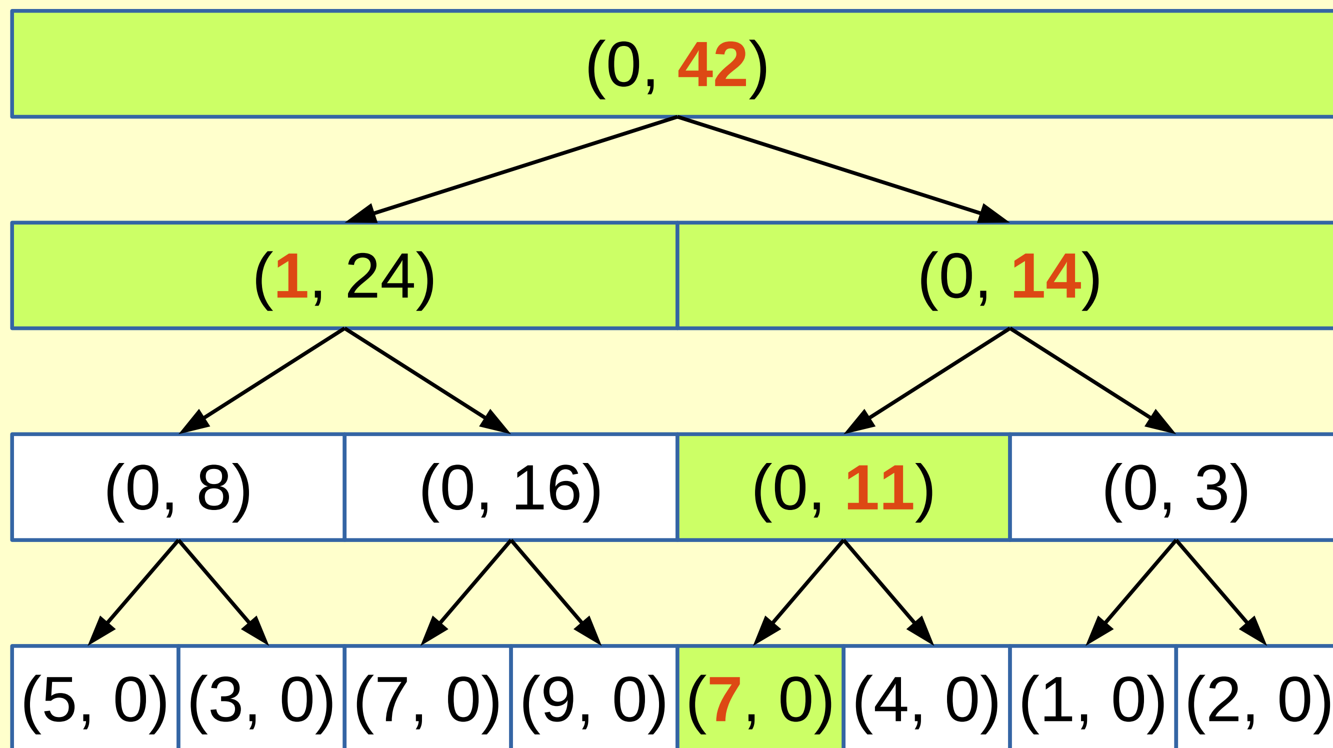
- 左: その節点の区間全体に一様に加えられた値
- 右: その節点の区間に一様でなく加えられた値の和



各節点は、対応する区間の和を持っている

A Simple Problem with Integers

- 1 ～ 5 番目に 1 を加えた時に更新が必要な節点



親の節点で一様に加えられたら、子に加算しない
→更新すべき節点数が抑えられる！

A Simple Problem with Integers

(参考) 先述の機能を実装したセグメント木は
蟻本 P.164に載っています

- では、BIT で表現するには
  どのような BIT を持てば良いか？

A Simple Problem with Integers

区間 [l, r] に x を加えると、各点での和はどう変化するかを考察しよう

- $s(i) := x$ を加える前の $a_1 + a_2 + \cdots + a_i$
- $s'(i) := x$ を加えた後の $a_1 + a_2 + \cdots + a_i$

A Simple Problem with Integers

3つに場合分けする

```
• i < l        →  s'(i) = s(i)
• l <= i <= r  →  s'(i) = s(i) + x*(i-l+1)
                        = s(i) + x*i - x*(l-1)
• r < i        →  s'(i) = s(i) + x*(r-l+1)
                        = s(i) + x*r - x*(l-1)
```

A Simple Problem with Integers

差分を考える

- i < l → s'(i) = s(i)
- l <= i <= r → s'(i) = s(i) + x*i - x*(l-1)
- r < i → s'(i) = s(i) + - x*(l-1) + x*r

A Simple Problem with Integers

差分を考える

- i < l → s'(i) = s(i)
- l <= i <= r → s'(i) = s(i) + x*i - x*(l-1)
- r < i → s'(i) = s(i) + - x*(l-1) + x*r

A Simple Problem with Integers

sum(bit, i) := BITの i までの累積和 とし、
s(i) = sum(bit1, i) × i + sum(bit0, i)
で表すこととすると、次のようにして更新が実現できる

- i < l → s'(i) = s(i)
- l <= i <= r → s'(i) = s(i) + x*i - x*(l-1)
- r < i → s'(i) = s(i) + - x*(l-1) + x*r

② + x*i
① - x*(l-1)
④ - x*i
③ + x*r

① bit0 の l に -x*(l-1) を加える
② bit1 の l に x を加える
③ bit0 の r+1 に x*r を加える
④ bit1 の r+1 に -x を加える

A Simple Problem with Integers

以上のことから、BIT を 2 つ持てば …

- 区間内の全ての要素に x を加算するクエリ
- 累積和を計算するクエリ

どちらも **O(log n)** で実現可能である！

- 実装してみましょう

A Simple Problem with Integers

- これを導くまでの考察のほうが大変です (実装は楽)

```cpp
// Verified: POJ 3468 (A Simple Problem with Integers)
// BIT 略

int N, Q;
int A[100010];
char T[100010];
int L[100010], R[100010], X[100010];

signed main() {
    cin >> N >> Q;
    BIT<int> bit0(N), bit1(N);
    repq(i,1,N) {
        cin >> A[i];
        bit0.add(i, A[i]);
    }
    rep(i,0,Q) {
        cin >> T[i] >> L[i] >> R[i];
        if(T[i] == 'C') {
            cin >> X[i];
            bit0.add(L[i]    , -X[i] * (L[i] - 1));
            bit1.add(L[i]    ,  X[i]            );
            bit0.add(R[i] + 1,  X[i] * R[i]     );
            bit1.add(R[i] + 1, -X[i]            );
        }
        else {
            int res = 0;
            res += bit0.sum(R[i]    ) + bit1.sum(R[i]    ) *  R[i];
            res -= bit0.sum(L[i] - 1) + bit1.sum(L[i] - 1) * (L[i] - 1);
            printf("%lld\n", res);
        }
    }
    return 0;
}
```

まとめ

- BIT は累積和 $a_1 + a_2 + \cdots a_i$ を求めるクエリと a_i に x を足すクエリを高速に処理できる
- セグメント木から、和の計算時の無駄を取り除いたような形をしている
- クエリの処理ではビット演算を使う
- 多次元への拡張が容易 (多重ループ)
- 工夫すれば他のクエリにも答えられる