Звездочкой (*) помечены разделы для старших

Частичные суммы и частичные разности

Для всякой конечной последовательности (массива) $A = a_1, a_2, \dots, a_n$ определим *последовательность частичных сумм (интегральную последовательность)* $\sigma(A) = p_1, p_2, \dots, p_n$, где $p_i = \sum_{k=1}^{i} a_k$ – сумма i-го префикса. Элементы интегральной последовательности также могут быть определены рекуррентно:

$$p_i = \begin{cases} a_i, & i = 1, \\ p_{i-1} + a_i, & i > 1. \end{cases}$$

С другой стороны,

$$a_i = \begin{cases} p_i, & i = 1, \\ p_i - p_{i-1}, & i > 1. \end{cases}$$

Таким образом, соответствие между последовательностями A и $\sigma(A)$ является взаимно-однозначным. Последовательность $\delta(A)$ будем называть последовательностью *частичных разностей* или *дифференциальной* по отношению к последовательности A, если

$$\delta(A) = a_1, (a_2 - a_1), (a_3 - a_2), \dots, (a_n - a_{n-1}).$$

Интегральная и дифференциальная последовательности связаны тождеством

$$\sigma\big(\delta(A)\big) = \delta\big(\sigma(A)\big) = A. \qquad (1)$$

В дальнейшем тождество (1) будем называть дельта-сигма тождеством. Оперируя элементами интегральной последовательности, можно вычислять сумму на любой непрерывной подпоследовательности (отрезке). Пусть $\sigma(a_1, a_2, \dots, a_n) = p_1, p_2, \dots, p_n$, а $s_{ij} = \sum_{k=i}^{j} a_k$ – сумма элементов с i по j тогда

$$s_{ij} = p_j - p_i + a_i, 1 \le i \le j \le n$$

Заметим, что

$$a_i - p_i = \begin{cases} 0, & i = 1, \\ -p_{i-1}, & i > 1. \end{cases}$$

Следовательно,

$$s_{ij} = \begin{cases} p_j, & i = 1, \\ p_j - p_{i-1}, & i > 1. \end{cases}$$

Бонус: в C++ есть стандартные функции расчета частичных сумм и частичных разностей: `std::partial_sum` *и* `std::adjacent_difference`.

Изменяемые последовательности

Рассмотрим, как изменения отдельных элементов последовательности A отражаются на элементах $\sigma(A)$ и $\delta(A)$. Прибавим некоторое число v к i-му элементу A, получив последовательность $A' = a_1, a_2, \dots, (a_i + v), \dots, a_n$.
Пусть $\delta(A) = d_1, d_2, \dots, d_n$, $\sigma(A) = p_1, p_2, \dots, p_n$, тогда

$$\delta(A') = d'_1, d'_2, \dots, d'_n$$

$$\sigma(A') = p'_1, p'_2, \dots, p'_n$$

$$p'_k = \begin{cases} p_k, & k < i, \\ p_k + v, & k \geq i. \end{cases}$$

$$d'_k = \begin{cases} d_k + v, & k = i, \\ d_k - v, & k = i + 1, \\ d_k, & \text{иначе.} \end{cases}$$

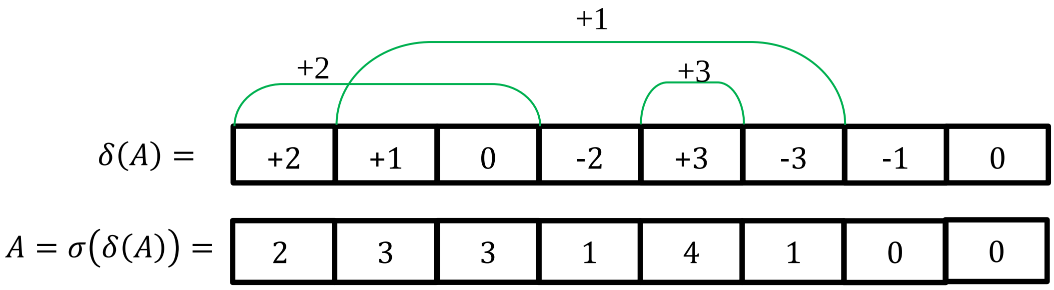
Таким образом, изменение i-го элемента последовательности влечет изменение $(n - i + 1)$ элементов интегральной последовательности и не более двух элементов дифференциальной последовательности.

Можно показать, что прибавление числа даже к нескольким последовательным элементам $a_i, a_{i+1}, \dots, a_j$ меняет не более двух частичных разностей. Пусть $A' = a_1, a_2, \dots, (a_i + v), (a_{i+1} + v), \dots, (a_j + v), \dots, a_n$, тогда

$$\delta(A')_k = \begin{cases} \delta(A)_k + v, & k = i, \\ \delta(A)_k - v, & k = i + 1, \\ \delta(A)_k, & \text{иначе.} \end{cases} \tag{2}$$

Используя это свойство, можно поддерживать массив частичных разностей, выполняя операции прибавления к отрезку за $O(1)$ и производя восстановление самого массива за $O(N)$.

Рассмотрим пример. Дан массив из 8 изначально нулевых элементов. Добавим 2 к элементам с 1-го по 3-й, 1 к элементам со 2-го по 6-й и 3 к 5-му элементу. Посчитаем частичные разности, используя правило (2), просуммируем их префиксы и получим значения элементов массива после всех изменений (см. рисунок ниже).



Обобщение на другие размерности

Понятия интегральной и дифференциальной последовательности естественным образом обобщаются на другие размерности. В качестве примера рассмотрим двумерный случай. Пусть $A = (a_{ij})$ – матрица размера $n \times m$, тогда интегральной матрицей по отношению к A будем называть такую матрицу $\sigma^2(A)$, что

$$\sigma^2(A) = \begin{bmatrix} p_{11} & \cdots & p_{1m} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nm} \end{bmatrix},$$

где $p_{ij} = \sum_{r=1}^{i} \sum_{c=1}^{j} a_{rc}$.

По аналогии с одномерным случаем p_{ij} может быть определено рекуррентно:

$$p_{ij} = \begin{cases} a_{11}, \ \ i = 1, j = 1, \\ p_{ij-1} + a_{ij}, i = 1, \\ p_{i-1j} + a_{ij}, j = 1, \\ p_{i-1j} + p_{ij-1} - p_{i-1j-1} + a_{ij}, i > 1, j > 1. \end{cases}$$

Нетрудно понять, что количество частных случаев в определениях $\sigma^k(A)$ и $\delta^k(A)$ будет расти экспоненциально с ростом k. Вводя величину

$$F(X, i, j) = \begin{cases} x_{ij}, i > 0, j > 0, \\ 0, i \leq 0 \vee j \leq 0. \end{cases}$$

где $X = (x_{ij})$ – матрица, мы избавляемся от необходимости описывать все частные случаи явно. Пусть $I = \sigma^2(A)$, тогда элементы матрицы A могут быть выражены следующим образом:

$$a_{ij} = F(I, i, j) - F(I, i-1, j) - F(I, i, j-1) + F(I, i-1, j-1). \qquad (3)$$

Определение интегральной матрицы кажется очевидным, но как определить дифференциальную матрицу? Это можно сделать несколькими способами, мы же определим дифференциальную матрицу исходя из дельта-сигма тождества. Итак, если $\delta^2(A)$ – дифференциальная матрица матрицы A, то

$$\delta^2\big(\sigma^2(A)\big) = \sigma^2\big(\delta^2(A)\big) = A.$$

Пусть

$$\delta^2(A) = \begin{bmatrix} d_{11} & \cdots & d_{1m} \\ \vdots & \ddots & \vdots \\ d_{n1} & \cdots & d_{nm} \end{bmatrix},$$

тогда, согласно выражению (3)

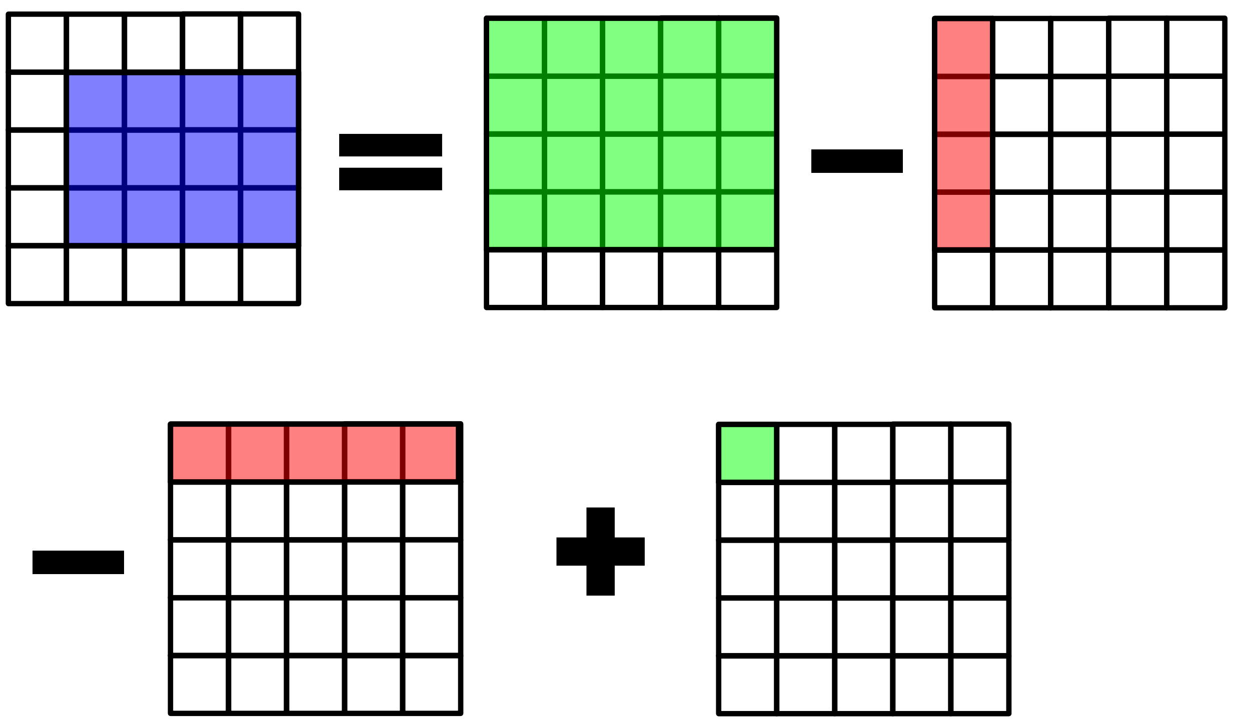
$$d_{ij} = F(A, i, j) - F(A, i-1, j) - F(A, i, j-1) + F(A, i-1, j-1).$$

С помощью элементов интегральной матрицы можно вычислять сумму на любой подматрице матрицы A. Пусть $I = \sigma^2(A)$ и

$$s(r_1, c_1, r_2, c_2) = \sum_{i=r_1}^{r_2} \sum_{j=c_1}^{c_2} a_{ij}$$

тогда выразим $s(\dots)$ через частичные суммы:

$$s(r_1, c_1, r_2, c_2) \\ = F(I, r_2, c_2) - F(I, r_2, c_1 - 1) - F(I, r_1 - 1, c_2) + F(I, r_1 - 1, c_1 - 1)$$



Нетрудно заметить, что формула для суммы подматрицы построена по принципу *включений-исключений* (ШТА?[1]). Аналогично выражаются суммы для размерностей больше 2.

Обобщение на произвольные операции

Пусть $(M,\ \circ)$ – множество M с определённой на нём бинарной операцией $\circ$, обладающей свойствами коммутативности и ассоциативности:

$$x \circ y = y \circ x, \forall x, y \in M$$

$$(x \circ y) \circ z = x \circ (y \circ z), \forall x, y, z \in M.$$

Также, во множестве M содержится нейтральный относительно операции $\circ$ элемент $\boldsymbol{e}$: $x \circ \boldsymbol{e} = \boldsymbol{e} \circ x = x, \forall x \in M$, и каждому $x \in M$ сопоставлен обратный элемент $x^{-1} \in M$ такой, что $x \circ x^{-1} = \boldsymbol{e}$**.** Иными словами $(M,\ \circ)$ – коммутативная группа. Тогда все рассуждения выше справедливы для множества M с операцией $\circ$, например:

- Исключающее «ИЛИ» (XOR):

$$M = \mathbb{N}, \boldsymbol{e} = 0, \circ = \mathrm{xor}, x^{-1} = x;$$

- Сумма по модулю N:

$$M = \{0,1,2, \dots, N-1\}, \boldsymbol{e} = 0, x \circ y = (x+y) \bmod N, x^{-1} = (N-x) \bmod N$$

- Произведение положительных чисел:

$$M = \mathbb{R}^{+}, \boldsymbol{e} = 1, x \circ y = xy, x^{-1} = \frac{1}{x}$$

- Произведение обратимых матриц 2×2:

$$M = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \middle| ad \neq bc;\ a, b, c, d \in \mathbb{R} \right\}$$

$$\boldsymbol{e} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\circ - \text{матричное произведение}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

[1] https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D1%83%D0%BB%D0%B0_%D0%B2%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9-%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9

В некоторых случаях необратимая операция может быть представлена группой обратимых. Ярким примером таких операций являются битовые AND и OR. Битовые операции выполняются поразрядно, следовательно, можно поддерживать частичные суммы для каждого разряда отдельно и находить значение каждого бита независимо. Это возможно благодаря тому, что результат операций AND и OR зависит только от количества единиц в каждом разряде, а именно:

– AND. i-ый бит результата равен 1, если i-ый бит всех операндов равен единице, в остальных случаях бит равен 0.
– OR. i-ый бит результата равен 0, если i-ый бит всех операндов равен 0, в остальных случаях бит равен 1.

Рассмотрим пример. Дан массив $A = \{1,2,3,4,5,6\}$ и мы хотим вычислить AND элементов с 4-го по 6-й и OR элементов с 4-го по 5-й.

A						Отрезок [4;6]		Отрезок [4; 5]	
a_1	a_2	a_3	a_4	a_5	a_6	Кол-во единиц	AND	Кол-во единиц	OR
0	0	0	1	1	1	**3**	1	2	1
0	1	1	0	0	1	1	0	**0**	0
1	0	1	0	1	0	1	0	1	1

Получаем, что $a_4\ \&\ a_5\ \&\ a_6 = 4$ и a_4 or $a_5 = 5$

Структуры данных

Массив частичных сумм

Применим в задачах, где необходим многократный расчёт сумм на отрезке некоторого массива, элементы которого не изменяются.

[StaticRangeSumQuery]:

Дан массив n чисел и даны m запросов вида «найти сумму на отрезке $[l; r]$»

```
//рассчитаем частичные суммы и будем отвечать на запрос за O(1)
int p[100500];
int a[100500];

inline int sum(int l, int r) {
	return l > 0 ? p[r] - p[l - 1] : p[r];
```

```
}

int main() {
	int n, m;
	cin >> n >> m;
	for (int i = 0; i < n; ++i)
		cin >> a[i];
	partial_sum(a, a + n, p);
	for (int i = 0; i < m; ++i) {
		int l, r;
		cin >> l >> r;
		cout << sum(l - 1, r - 1) << '\n';
	}
}
```

Массив частичных разностей

Используется в задачах, где многократно происходит прибавление числа к отрезкам массива, а восстановить массив в явном виде нужно только после всех прибавлений.

[OfflineRangeUpdate]:

Есть n пустых ящиков и m запросов вида «положить v шариков в ящики с номерами с l по r». Вывести итоговое количество шариков в каждом ящике после выполнения всех запросов.

Решение: будем хранить массив частичных разностей, а в конце восстановим исходный массив, посчитав частичные суммы (вспоминаем про $\sigma\big(\delta(A)\big) = A$).

```
int d[100500];

int main() {
	int n, m;
	cin >> n >> m;
	for (int i = 0; i < m; ++i) {
		int l, r, v;
		cin >> l >> r >> v;
		--l; --r;
		d[l] += v;
		if (r != n - 1)
			d[r + 1] -= v;
	}
	int cur = 0;
	for (int i = 0; i < n; ++i) {
		cur += d[i];
		cout << cur << ' ';
	}
```

```
}
```

*Дерево Фенвика

Дерево Фенвика – способ поддержания частичных сумм в онлайне, т.е. допустимы изменения отдельных элементов массива. Базовые операции: прибавить v к элементу с индексом i и запросить сумму префикса j. И модификация и запрос суммы выполняются за $O(\log n)$. Неплохое описание дерева Фенвика есть на топкодере[2], ИТМО[3]. Мы традиционно используем вариант, как на топкодере, т.е. индексируем с единицы, что позволяет описать обновление и запрос суммы единообразно:

```
void add(int pos, int v) {
	for (int i = pos; i <= n; i += i & -i) {
		t[i] += v;
	}
}

int sum(int pos) {
	int res = 0;
	for (int i = pos; i > 0; i -= i & -i) {
		res += t[i];
	}
	return res;
}
```

Главное преимущество дерева Фенвика – элементарное (с точки зрения реализации) обобщение на многомерные случаи. Например, 2D дерево:

```
inline void add(int x, int y, int v) {
	for (int i = x; x <= n; i += i & -i) {
		for (int j = y; j <= m; j += j & -j) {
			t[i][j] += v;
		}
	}
}

inline int sum(int x, int y) {
	int res = 0;
	for (int i = x; i > 0; i -= i & -i) {
		for (int j = y; j > 0; j -= j & -j) {
			res += t[i][j];
		}
```

[2] https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

[3] https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%A4%D0%B5%D0%BD%D0%B2%D0%B8%D0%BA%D0%B0

```
    }
    return res;
}

//сумма в прямоугольнике (x1, y1), (x2, y2), x1 <= x2, y1 <= y2
inline int rectSum(int x1, int y1, int x2, int y2) {
    return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y - 1);
}
```

[DynamicRangeSumQuery], вариант 1

Дан массив n чисел и даны m запросов 2-х типов: «найти сумму на отрезке $[l; r]$» и «прибавить число v к i-му элементу».

Решение: будем поддерживать дерево Фенвика для массива

```
int n;
int t[100500];

inline void add(int pos, int v) {
    for (int i = pos; i <= n; i += i & -i) {
        t[i] += v;
    }
}

inline int sum(int pos) {
    int res = 0;
    for (int i = pos; i > 0; i -= i & -i) {
        res += t[i];
    }
    return res;
}

inline int sum(int l, int r) {
    return sum(r) - sum(l - 1); //нет проверки на равенство l единице,
т.к. sum(0) гарантированно вернет 0
}

int main() {
    int m;
    cin >> n >> m;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        add(i + 1, x);
    }
    for (int i = 0; i < m; ++i) {
        int t, x, y;
        cin >> t >> x >> y;
        if (t == 1) { //запрос прибавления
```

```
            add(x, y);
        } else { //запрос суммы
            cout << sum(x, y) << '\n';
        }
    }
    return 0;
}
```

[DynamicRangeSumQuery], вариант 2

То же, что и первый вариант, только вместо прибавления – присваивание значения v i-му элементу массива.

Решение: если i-ый элемент имеет значение w, а нужно присвоить v, то это равносильно прибавлению $(v - w)$. <u>Нетрудно заметить, что такой приём уместен для любой обратимой операции.</u> Для удобства можно поддерживать не только дерево Фенвика, но и сам массив (да, можно брать текущее значение из дерева Фенвика, но отдельным массивом – быстрее и удобнее).

```
//...
int a[100500];
//...
add(i + 1, x);
a[i] = x;
//...
if (t == 1) {
    add(x, (y - a[i]));
    a[i] = y;
}
//...
```

[OnlineRangeUpdate]

Дан массив n элементов, есть два типа запросов: «прибавить число v к отрезку $[l; r]$» и «вывести значение i-<u>го элемента</u>».

Решение: Аналогично [OfflineRangeUpdate] будем поддерживать дерево Фенвика на массиве частичных разностей. Тогда значение i-го элемента массива равно сумме i-го префикса.

```
int n;
int t[100500];

inline void add(int pos, int v) {
```

```
        for (int i = pos; i <= n; i += i & -i) {
            t[i] += v;
        }
}

inline int sum(int pos) {
        int res = 0;
        for (int i = pos; i > 0; i -= i & -i) {
            res += t[i];
        }
        return res;
}

int main() {
        int m;
        cin >> n >> m;
        for (int i = 0; i < n; ++i) {
            int x;
            cin >> x;
            add(i + 1, x);
            add(i + 2, -x);
        }
        for (int i = 0; i < m; ++i) {
            int t, x, y, v;
            cin >> t;
            if (t == 1) { //запрос прибавления к отрезку [x; y]
                cin >> x >> y >> v;
                add(x, v);
                add(y + 1, -v);
            } else { //запрос элемента
                cin >> x;
                cout << sum(x) << '\n';
            }
        }
}
```

[RangeUpdate+RangeSumQuery]

Прибавление на отрезке + запрос суммы на отрезке – типичная задача для дерева отрезков. Но есть трюк, позволяющий решать её деревом Фенвика, точнее, двумя. О нем можно почитать здесь[4] и тут[5]. В этом случае также используется трюк с поддержанием частичных разностей, но вместо значений в дереве хранятся функции от индекса $k \cdot i + b$ (одно дерево для k, другое дерево

[4] http://petr-mitrichev.blogspot.com/2013/05/fenwick-tree-range-updates.html
[5] https://kartikkukreja.wordpress.com/2013/12/02/range-updates-with-bit-fenwick-tree/

для b). Дерево Фенвика для RangeUpdate+RSQ быстрее дерева отрезков (хотя, дерево отрезков можно по-разному написать).

Дерево Фенвика vs. Дерево отрезков

Помимо RangeUpdate+RSQ дерево Фенвика может конкурировать с деревом отрезков и на необратимых операциях. Мы уже разобрали, как реализовать AND или OR (И или ИЛИ, **ИИЛИИЛИ**) через частичные суммы по каждому разряду, но в этом случае дерево Фенвика, очевидно, уступает дереву отрезков по времени, так как запрос будет выполняться за $O(\log X_{max} \cdot \log N)$. На самом деле, обратимость операции sнеобходима только для вычисления операции на отрезке и реализации присваивания. Если необходимо вычислять оператор только на префиксе и обновления имеют вид $a_i = a_i \circ v$ (прибавление $a_i = a_i + v$, улучшение максимума/минимума $a_i = \max(a_i, v)$, $a_i = \min(a_i, v)$, обновление НОД $a_i = \gcd(a_i, v)$), то дерево Фенвика применимо и заметно выигрывает у ДО по всем параметрам: время реализации, время работы, память.