**Sqrt-декомпозиция**

Sqrt-декомпозиция — это метод, или структура данных, которая позволяет выполнять некоторые типичные операции (суммирование элементов подмассива, нахождение минимума/максимума и т.д.) за O(), что значительно быстрее, чем O(n) для тривиального алгоритма.

**Структура данных на основе sqrt-декомпозиции**

**Поставим задачу**. Дан массив a[0 \ldots n-1]. Требуется реализовать такую структуру данных, которая сможет находить сумму элементов a[l \ldots r] для произвольных l и r за O(\sqrt{n}) операций.

**Описание**

Основная идея sqrt-декомпозиции заключается в том, что сделаем следующий **предпосчёт**: разделим массив a на блоки длины примерно \sqrt{n}, и в каждом блоке i заранее предпосчитаем сумму b[i] элементов в нём.

Можно считать, что длина одного блока и количество блоков равны одному и тому же числу — корню из n, округлённому вверх:

 s = \left\lceil \sqrt{n} \right\rceil, 

тогда массив a[] разбивается на блоки примерно таким образом:

 \underbrace{ a[0] ~ a[1] ~ \ldots ~ a[s-1] }_{b[0[...]

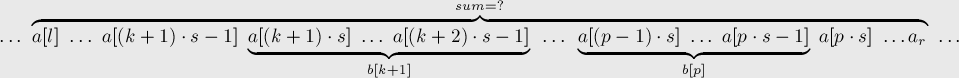
Хотя последний блок может содержать меньше, чем s, элементов (если n не делится на s), — это не принципиально.

Таким образом, для каждого блока k мы знаем сумму на нём b[k]:

 b[k] = \sum_{i=k \cdot s}^{\min (n-1, (k+1) \cdot[...]

Итак, пусть эти значения b_k предварительно подсчитаны (для этого надо, очевидно, O(n) операций). Что они могут дать при вычислении ответа на очередной запрос (l,r)? Заметим, что если отрезок [l;r] длинный, то в нём будут содержаться несколько блоков целиком, и на такие блоки мы можем узнать сумму на них за одну операцию. В итоге от всего отрезка [l;r] останется лишь два блока, попадающие в него лишь частично, и на этих кусках нам придётся произвести суммирование тривиальным алгоритмом.

Иллюстрация (здесь через k обозначен номер блока, в котором лежит l, а через p — номер блока, в котором лежит r):



На этом рисунке видно, что для того чтобы посчитать сумму в отрезке [l \ldots r], надо просуммировать элементы только в двух "хвостах": [l \ldots (k+1) \cdot s-1] и [p \cdot s \ldots r], и просуммировать значения b[i] во всех блоках, начиная с k+1 и заканчивая p-1:

 \sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1) \cdot s-1[...]

(примечание: эта формула неверна, когда k=p: в таком случае некоторые элементы будут просуммированы дважды; в этом случае надо просто просуммировать элементы с l по r)

Тем самым мы экономим значительное количество операций. Действительно, размер каждого из "хвостов", очевидно, не превосходит длины блока s, и количество блоков также не превосходит s. Поскольку s мы выбирали \approx \sqrt{n}, то всего для вычисления суммы на отрезке [l \ldots r] нам понадобится лишь O(\sqrt{n}) операций.

**Реализация**

Приведём сначала простейшую реализацию:

// входные данные

int n;

vector<int> a (n);

// предпосчёт

int len = (int) sqrt (n + .0) + 1; // и размер блока, и количество блоков

vector<int> b (len);

for (int i=0; i<n; ++i)

b[i / len] += a[i];

// ответ на запросы

for (;;) {

int l, r; // считываем входные данные - очередной запрос

int sum = 0;

for (int i=l; i<=r; )

if (i % len == 0 && i + len - 1 <= r) {

// если i указывает на начало блока, целиком лежащего в [l;r]

sum += b[i / len];

i += len;

}

else {

sum += a[i];

++i;

}

}

Недостатком этой реализации является то, что в ней неоправданно много операций деления (которые, как известно, выполняются значительно медленнее других операций). Вместо этого можно посчитать номера блоков c_l и c_r, в которых лежат границы l и r соответственно, и затем сделать цикл по блокам с c_l+1 по c_r-1, отдельно обработав "хвосты" в блоках c_l и c_r. Кроме того, при такой реализации случай c_l = c_r становится особым и требует отдельной обработки:

int sum = 0;

int c\_l = l / len, c\_r = r / len;

if (c\_l == c\_r)

for (int i=l; i<=r; ++i)

sum += a[i];

else {

for (int i=l, end=(c\_l+1)\*len-1; i<=end; ++i)

sum += a[i];

for (int i=c\_l+1; i<=c\_r-1; ++i)

sum += b[i];

for (int i=c\_r\*len; i<=r; ++i)

sum += a[i];

}