Алгоритм Мо

Попов Владимир Сергеевич,
Б01-411

Актуальность

Алгоритм Мо полезен при обработке большого количества запросов к отрезкам на неизменяемом массиве. Основное применение - анализ больших неизменяемых данных

- Олимпиадное программирование
- Обработка логов
- Анализ трафика

Математическая задача

- Множество индексов $I = \{1, 2, \ldots, n\}$.
- Функция $a : I \to A$, где A - множество значений (массив $[a_1, ..., a_n]$).
- Множество запросов $Q = \{q_1, q_2, \ldots, q_m\}$, где $q_i = (l_i, r_i) \in I \times I$, причём $l_i \leq r_i$.
- Функция $f : I \times I \to R$, где R - множество результатов.

Требуется вычислить $f(q_i), \forall i = \overline{1, m}$

Математическая задача

Пусть дано состояние $S \in \mathcal{S}$ соответствующее отрезку $[l, r]$, то есть $S = S(l, r)$.

Должна существовать функция $U_\delta : \mathcal{S} \times A \to \mathcal{S}$, где $\delta = \pm 1 (1 - remove, -1 - add)$

Должно выполняться:

$$S(L \pm 1, R) = U_{\pm 1}(S(L, R), a[L]) \quad \text{и} \quad S(L, R \pm 1) = U_{\pm 1}(S(L, R), a[R]).$$

Должна существовать функция $G : \mathcal{S} \to R : f(l, r) = G(S(l, r))$

$\forall S \in \mathcal{S}$ и $\forall a \in A$ должно выполняться:

$$U_{-1}(U_{+1}(S, a),\, a) = S \quad \text{и} \quad U_{+1}(U_{-1}(S, a),\, a) = S.$$

Как работает

- Работаем в оффлайн.
- Главная идея - переупорядочить запросы так, чтобы переход от прошлого к следующему был дешёвым. Сначала отсортируем по блокам исходя из левой границы, потом по правой
- Проходимся по всем запросам и “дошагиваем” с текущей позиции до запроса прибавлением и удалением элементов справа и слева

```
struct Query:
    int l, r, id

int K = sqrt(N)
int a = 1, b = 0

bool isLess(Query a, Query b):
    if a.l / K != b.l / K:
        return a.l < b.l
    return a.r < b.r

function process(Query[Q] q):
    sort(q, isLess)
    for i = 0 to Q - 1:
        while a > q[i].l:
            addLeft(a - 1)
            a -= 1
        while b < q[i].r:
            addRight(b + 1)
            b += 1
        while a < q[i].l:
            delLeft(a)
            a += 1
        while b > q[i].r:
            delRight(b)
            b -= 1
    result[q[i].id] = answer()
```

Псевдокод алгоритма Мо[1]

Асимптотика

Будем считать, что операции удаления и добавления слева и справа O(1)

- Правая граница во время обработки 1 группы будет только увеличиваться, значит O(n*n/K)

- Для запросов (l_i, r_i]) и (l_j, r_j) очевидно $|l_i - l_j| < K$, значит обработка левых границ одной группы - O(K^2), а всех - O(m*K)

- Также у нас есть сортировка O(m*logm)

- Суммируя получаем O(m*logm + n^2/K+m*K)
- Чтобы минимизировать сложность возьмём K = n/sqrt(m)

- Итог: O(m*logm + n*sqrt(m))
- Если m соизмеримо с n то можно взять K = sqrt(n), тогда O(m*logm + n*sqrt(n))

Пример задачи

D. Мощный массив

ограничение по времени на тест: 5 seconds
ограничение по памяти на тест: 256 megabytes

Имеется массив натуральных чисел $a_1, a_2, ..., a_n$. Рассмотрим некоторый его подмассив $a_l, a_{l+1}, ..., a_r$, где $1 \le l \le r \le n$, и для каждого натурального числа s обозначим через K_s число вхождений числа s в этот подмассив. Назовем *мощностью* подмассива сумму произведений $K_s \cdot K_s \cdot s$ по всем различным натуральным s. Так как количество различных чисел в массиве конечно, сумма содержит лишь конечное число ненулевых слагаемых.

Необходимо вычислить мощности каждого из t заданных подмассивов.

Входные данные

Первая строка содержит два целых числа n и t $(1 \le n, t \le 200000)$ — длина массива и количество запросов соответственно.

Вторая строка содержит n натуральных чисел a_i $(1 \le a_i \le 10^6)$ — элементы массива.

Следующие t строк содержат по два натуральных числа l и r $(1 \le l \le r \le n)$ — индексы левого и правого концов соответствующего подмассива.

Выходные данные

Выведите t строк, где i-ая строка содержит единственное натуральное число — мощность подмассива i-го запроса.

Пример задачи на алгоритм Мо[2]

Пример задачи. Сравнение решений

- **Решение брутфорсом (brutforce.cpp)**
  Каждый запрос будем проходить от left до right и записывать все частоты в unordered_map.

- **Решение через корневую декомпозицию (sqrt_decompose.cpp)**
  Поделим запросы числа на “частые” (freq > sqrt(n)) и “редкие” (иначе). Частые будем считать через префиксные сумма за O(1), а редкие - искать в их массиве позиций бинарным поиском за O(log(sqrt(n))) в среднем.

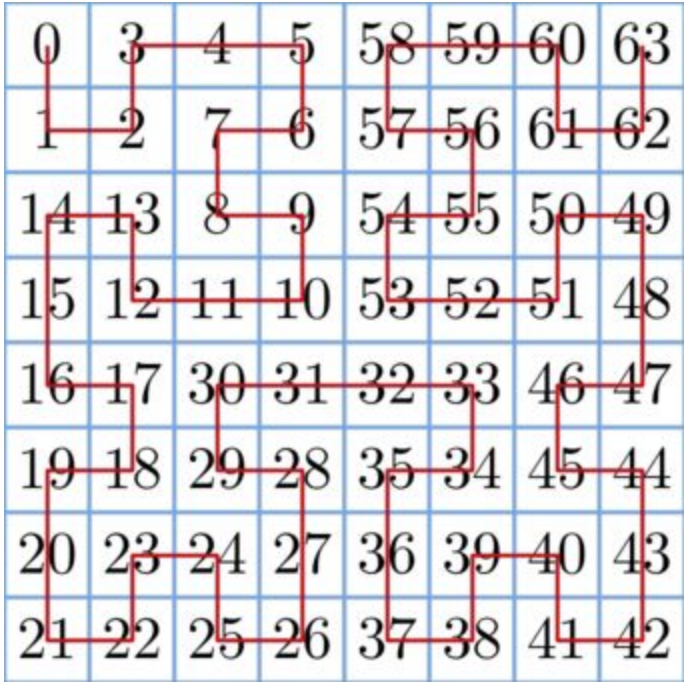
- **Решение через алгоритм Мо (mo.cpp)**
  Отсортируем запросы, как описывалось ранее. Заводим частотный словарь и в пересчетах обновляем его.

Решение \ Харак-ика	Худшее по времени	Среднее по времени	Память
Брутфорс	$O(t*n^2)$	O(t*n) амортизировано	O(n^2)
Корнячка	O(t*n*log(n))	O(t*sqrt(n)*log(sqrt(n)))	O(n*sqrt(n) + maxA)
Мо	O(t*logt + n*sqrt(n))	O(t*logt + n*sqrt(n))	O(n + t + maxA)

Константные оптимизации

- Каждый чётный блок сортировать right по убыванию, а каждый нечетный - по возрастанию.

- Как ещё меньше “прыгать” - кривая Гильберта



Визуализация кривой Гильберта[3].

```
unsigned long long HilbertOrder(int x, int y, int pow, int rotate) {
    if (pow == 0) {
        return 0ULL;
    }

    const int height = 1 << (pow - 1);

    int seg = (x < height ? 0 : 1) * 2 + (y < height ? 0 : 1);
    seg = (seg + rotate) & 3;

    static constexpr const int rotateDelta[4] = {3, 0, 0, 1};

    int newX = x & (height - 1);
    int newY = y & (height - 1);
    int newRotate = (rotate + rotateDelta[seg]) & 3;

    unsigned long long lowBits = HilbertOrder(newX, newY, pow - 1, newRotate);

    unsigned long long highBits = (unsigned long long)seg << (2 * (pow - 1));

    return highBits | lowBits;
}
```

Расчёт значения кривой гильберта[3].

Константные оптимизации

n	q	$\frac{n}{q}$	Standard Mo time	Mo+Hilbert time
400000	400000	1	2730 ms	2698 ms
1000000	1000000	1	13602 ms	10841 ms
500000	250000	2	3369 ms	2730 ms
1000000	500000	2	10077 ms	7644 ms
600000	200000	3	4134 ms	2901 ms
1000000	333333	3	8767 ms	6240 ms
600000	150000	4	4851 ms	2496 ms
1000000	250000	4	8672 ms	5553 ms
700000	140000	5	6255 ms	2854 ms

n	q	$\frac{n}{q}$	Standard Mo time	Mo+Hilbert time
1000000	200000	5	8423 ms	5100 ms
750000	100000	7.5	5116 ms	2667 ms
1000000	333333	7.5	7924 ms	4009 ms
1000000	100000	10	7425 ms	3977 ms
1000000	40000	25	9671 ms	2355 ms
1000000	20000	50	9016 ms	1590 ms
1000000	10000	100	6879 ms	1185 ms
1000000	5000	200	5802 ms	857 ms
1000000	2500	400	4897 ms	639 ms

Сравнение бенчмарков стандартной реализации Мо и с оптимизацией Гильбертом[3].

Список литературы

- [1] Общее описание алгоритма
https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9C%D0%BE&mobileaction=toggle_view_desktop

- [2] Задача “Мощный массив”
https://codeforces.com/problemset/problem/86/D

- [3] Оптимизация кривой Гильберта
https://codeforces.com/blog/entry/61203

Полезные ссылки

- Гитхаб с докладом и кодом
https://github.com/kzueirf12345/mo

- Мой телеграмм канал
https://t.me/+DSOn7YUQQN9mODFi