Применения корневой декомпозиции

Равилов Игорь Б05-325

Май 2024

1 Введение

Корневая декомпозиция — это процесс разделения структуры размером $O(N)$ на $O(\sqrt{N})$ блоков размером $O(\sqrt{N})$ каждый, что облегчает манипуляцию всей структурой. Корневая декомпозиция чрезвычайно универсальна. Некоторые из её наиболее известных применений включают:

- ответы на запросы по статическому массиву с помощью методов, таких как алгоритм Мо или предвычисления блоков
- "ленивые" модификации методом перебора, где легко запросить весь блок, но неочевидно как обновлять флаги или изменения внутри блока (так как необходимо учитывать их влияние на весь массив)
- разделение объектов (например элементов массива) на основе порогового значения, когда существует как $O(x)$, так и $O(n/x)$ алгоритмы.

Рассмотрим несколько типов корневой декомпозиции.

2 Ответы на запросы на подотрезках массива (алгоритм Мо), офлайн

Рассмотрим задачу, где необходимо найти ответ для определённых отрезков $[l, r]$. Мы не можем быстро вычислить ответ для произвольного отрезка, но знаем, как перейти к $[l, r\pm1]$ и $[l\pm1, r]$, имея некоторую информацию о состоянии, оставшуюся после вычисления ответа для $[l, r]$. Количество обновлений (переходов), которое нам нужно сделать, чтобы перейти от $[l_1, r_1]$ к $[l_2, r_2]$, равно $|l_1 - l_2| + |r_1 - r_2|$.

Если есть только два отрезка, нам не нужно большое количество переходов. Однако, если отрезков много, выбор хороших переходов значительно сократит общее время. Поиск лучшего пути перехода является NP-трудной задачей, поэтому мы сосредоточимся на оценке *хорошего* пути.

Рассмотрим следующую схему для **задачи коммивояжера** с Манхэттенской метрикой **на декартовой плоскости**: мы делаем корневую декомпозицию по координате x, разделяя сетку на вертикальные блоки по n ячеек в высоту и B ячеек в ширину, таких блоков n/B.

Для произвольного блока подсчитаем количество точек, которые нам нужно посетить в нём. Чтобы посетить все точки, мы можем посетить их сверху вниз: нам потребуется не более $O(BC)$ переходов влево и вправо и $O(n)$ переходов вниз.

Если для каждого блока нам нужно $O(BC + n)$ переходов, в сумме нам потребуется $O(\sum(BC + n)) = O(Bq + n^2/B)$ переходов. Выбор размера блока $B = O(n/\sqrt{q})$ даст нам общее количество переходов $O(qn/\sqrt{q} + n\sqrt{q}) = O(n\sqrt{q})$.

Реализация сортировки этих переходов осуществляется следующим образом:

```
int B = ???;
struct query {
    int l, r, id;
    const bool operator<(const query& o) const {
        if (l / B == o.l / B) {
            // они в одном блоке, сортируем по убыванию
            return r < o.r;
        } else {
            // они не в одном блоке, сортируем по номеру блока
            return l < o.l;
        }
    }
};
```

Листинг 1: Пример реализации сортировки переходов

Обратите внимание, что между каждым блоком нам нужно пробегать от нижней части "сетки"до самой верхней. Мы можем избежать этого, используя зигзагообразный паттерн, получая небольшую оптимизацию:

```
int B = ???;
struct query {
    int l, r, id;
    const bool operator<(const query& o) const {
        if (l / B == o.l / B) {
            if ((l / B) % 2 == 0) return r < o.r;
            else return r > o.r;
        } else {
            return l < o.l;
        }
    }
};
```

Листинг 2: Пример оптимизации с зигзагообразным паттерном

3 Запрос уникальных элементов на отрезке (SPOJ DQUERY)

Условие: Дан статический массив и запросы вида $[l, r]$. Необходимо посчитать количество различных значений, встречающихся в отрезке $[l, r]$.

Наивная реализация интуитивна: мы поддерживаем таблицу v, где $v[x]$ – это количество раз, когда x появлялся в данном отрезке. При добавлении элемента x такой, что $v[x] = 0$ – увеличиваем ответ. При удалении элемента x такой, что $v[x] = 1$ – уменьшаем ответ.

Однако, это требует много прыжков по памяти для доступа к ячейкам таблицы, что неэффективно с точки зрения кэширования и приводит к огромной константе. Рассмотрим более эффективную реализацию.

Создадим массивы $pre[i]$ и $nxt[i]$, которые равны последнему положению элемента i и следующему положению элемента i соответственно (они равны $+INF$, если элемента не существует). Мы можем использовать эти массивы для быстрой проверки наличия элемента в отрезке.

- Случай 1: $[l, r] \to [l, r+1]$. Мы добавляем 1 тогда и только тогда, когда $r+1 \notin [l, r]$, или, что эквивалентно, $pre[r+1] < l$.
- Случай 2: $[l, r] \to [l, r-1]$. Мы вычитаем 1 тогда и только тогда, когда $r \notin [l, r-1]$.
- Случай 3: $[l, r] \to [l-1, r]$. Мы добавляем 1 тогда и только тогда, когда $l-1 \notin [l, r]$, или, что эквивалентно, $r < nxt[l-1]$.
- Случай 4: $[l, r] \to [l+1, r]$. Мы вычитаем 1 тогда и только тогда, когда $l \notin [l+1, r]$.

Это приводит к гораздо более быстрому решению, которое работает достаточно быстро даже для $n = 10^6$, даже если мы реализуем классический алгоритм Мо.

4 Запрос инверсий на отрезке (наивная реализация)

Условие: Дан статический массив и запросы вида $[l, r]$. Необходимо посчитать количество пар (i, j) таких, что $a[i] > a[j]$.

Каждый раз, когда мы добавляем или удаляем элемент, мы учитываем, сколько этот элемент вносит в ответ. Если мы будем поддерживать дерево Фенвика для пересчета текущего отрезка, мы можем быстро определить, сколько инверсий включает определенное значение. Запросы выполняюися за $O(\log n)$; поэтому в целом это $O(n \log n \sqrt{n})$.

5 Продвинутый уровень: Sweepline Мо, офлайн

Это аналог задачи 617E - XOR и любимое число (будет разобрано отдельно дальше), за исключением того, что здесь имеется несколько *любимых чисел*.

Условие: Дан статический массив a размера n и статическое множество S размером C и запросы вида $[l, r]$, посчитать количество пар $(i, j) \in [l, r]$ таких, что $a[i] \oplus a[j] \in S$. Время $O(nC + n\sqrt{n})$. Память $O(n)$. Пусть $q = O(n)$.

Заметьте, что $a[i] \oplus a[j] \in S$ эквивалентно $\exists v \in S : a[i] \oplus v = a[j]$. Мы можем рассмотреть каждое v отдельно и запустить C итераций алгоритма Мо, но это будет $O(nC\sqrt{n})$. Рассмотрим переходы: мы добавляем элемент j и хотим узнать количество i таких, что $a[i] \oplus a[j] \in S$ и $i \in [l, r]$.
Назовем переходную функцию $f(l, r, p)$, где p – элемент, который мы добавляем. Удаление же можно выполнить, вычитая $f(l, r-1, p)$ или $f(l+1, r, p)$ в зависимости от направления. Sweepline Мо можно использовать, когда эта переходная функция может быть представлена как разность двух значений функции, каждое из которых зависит только от одной границы отрезка: то есть она удовлетворяет равенству $f(l, r, p) = f(1, r, p) - f(1, l-1, p)$.

Основная идея заключается в том, чтобы сначала запустить первую *предварительную* итерацию алгоритма Мо и найти все $f(1, x, p)$, которые нам нужно вычислить. Для этого мы воспользуемся сканирующей (заметающей) прямой по x, обновляя соответствующим образом структуру данных и вычисляя $f(1, x, p)$, которые включают x. В общем случае "вычислительная" часть должна быть $O(1)$, как у обычного алгоритма Мо, но мы оставляем больше места для манёвра в части обновления. Стоит отметить, что так как мы добавляем всего n элементов, то сложность операции добавления может быть $O(\sqrt{n})$.

Однако, если мы будем хранить все (x, p), то получим $O(n\sqrt{n})$ памяти и огромную константу. Обратим внимание, что из-за порядка переходов в алгоритме Мо мы можем разделить все p на несколько типов, учитывая фиксированную x:

1. $p = x$.
2. $p = x + 1$
3. $p \in [A, B]$

Рассмотрим реализацию:
Для первого и второго типов мы вычисляем $ansPrev[i]$ и $ansThere[i]$ (см. пример) и не обновляем ничего, оставляя эти массивы для использования второй *основной* итерацией алгоритма Мо.

Для третьего типа мы находим все (p, A, B), которые получаются из первой итерации алгоритма Мо. Таких отрезков не более $4(q-1)$, поскольку перемещение любого конца отрезка в алгоритме Мо создает ровно один отрезок. Затем, для каждого фиксированного x, мы храним соответствующие отрезки (p, A, B) и итерируемся по ним, добавляя найденное значение перехода к соответствующему запросу.

```
sort(qs, qs + q);
cl = 1, cr = 0;

for(int i = 0; i < q; i++) {
    if (cr < qs[i].r) {
        iv[cl - 1].push_back({cr + 1, qs[i].r, i, -1});
        cr = qs[i].r;
    }
    if (qs[i].l < cl) {
        iv[cr].push_back({qs[i].l, cl - 1, i, 1});
        cl = qs[i].l;
    }
    if (qs[i].r < cr) {
        iv[cl - 1].push_back({qs[i].r + 1, cr, i, 1});
        cr = qs[i].r;
    }
    if (cl < qs[i].l) {
        iv[cr].push_back({cl, qs[i].l - 1, i, -1});
        cl = qs[i].l;
    }
}
```

```
26
27 for(int i = 1; i <= n; i++) {
28     ansPrev[i] = ansPrev[i - 1] + cgc[ar[i]];
29     add(ar[i]);
30     for (auto [l, r, i, c] : iv[i]) {
31         ll g = 0;
32         for(int p = l; p <= r; p++)
33             g += cgc[ar[p]];
34         ans[i] += c * g;
35     }
36     ansThere[i] = ansThere[i - 1] + cgc[ar[i]];
37 }
38
39 for(int i = 0; i < q; i++) {
40     curans += ans[i];
41     fans[qs[i].i] = curans + ansPrev[qs[i].r] + ansThere[qs[i].l - 1];
42 }
43
44 for(int i = 0; i < q; i++) {
45     print(fans[i]);
46     pc('\n');
47 }
```

Листинг 3: Пример реализации

Особенности реализации

- Функция `add` просто обновляет массив `cgc`, который содержит количество элементов в текущем префиксе, xor-нутых с элементами в множестве. Это можно делать за $O(C)$ времени, просто добавляя элементы в префикс напрямую.
- `iv` – это отрезки для каждого x, вместе с запросом и коэффициентом вклада.
- Если мы посчитаем префиксную сумму на `ansPrev` и `ansThere`, нам даже не нужно будет использовать цикл `while` во второй итерации алгоритма Мо.

6 Запрос инверсий на отрезке за $O(n\sqrt{n})$

Рассмотрим алгоритм sweepline Мо. В общем случае, сканирующая прямая используется для уменьшения количества изменений массива для алгоритма Мо до $O(n)$, не влияя на количество запросов $O(n\sqrt{n})$. Это означает, что изменения должны выполняться за $O(\sqrt{n})$, а запросы должны вычисляться за $O(1)$.

Для этой задачи нам нужно поддерживать **инкремент/декремент в точке** за $O(\sqrt{n})$ и **запрос префиксной суммы** за $O(1)$. Создадим эквивалентную задачу: **инкремент / декремент на суффиксе** за $O(\sqrt{n})$ и **запрос в точке** за $O(1)$.

Разделим вспомогательный массив на блоки размера $O(\sqrt{n})$. Для каждого блока будем поддерживать флаг lazy. Когда мы выполняем инкремент/декремент суффикса, для блоков, пересекающихся с этим суффиксом, но не полностью покрытых им, изменяем элементы напрямую в соответствующих блоках. В противном случае для блоков, которые полностью покрываются суффиксом, мы не проходим по каждому элементу, а вместо этого

изменяем флаг lazy для блока. Чтобы выполнить запрос позиции, добавляем (перебором) значение с соответствующим флагом lazy. Это означает, что для уменьшения количества непосредственных изменений используются отложенные операции.

Для тех, кто считает, что между $O(n\sqrt{n})$ и $O(n \log n\sqrt{n})$ нет большой разницы: при $n = 100000$, первый выполняется за 300ms, в то время как второй выполняется за 3000ms. Обычно лимит времени составляет 1000ms.

7 Ответы на онлайн запросы по статическому массиву

В отличие от оффлайн, когда мы работаем с онлайн запросами на статическом массиве, мы не можем сортировать их при каждом запросе, поэтому можно сделать следующий трюк: можно сначала выполнить декомпозицию на блоки размера B, а затем для каждого интервала блоков построить структуру данных, которая поможет отвечать на запросы (как в Мо), что приведет к созданию $O((\frac{N}{B})^2)$ массивов. Таким образом нам не будет требоваться знать все запросы заранее. Рассмотрим подробнее далее.

Предположим, что время запроса для этой структуры данных $O(T)$, тогда запрос для произвольного интервала можно выполнить за $O(TB)$.

Обычно инициализация каждой структуры составляет $O(B)$, а запрос (переход между структурами) за $O(1)$, поэтому лучше выбрать размер блока $O(\sqrt{n})$, чтобы получить предподсчёт за $O(n\sqrt{n})$ и запросы $O(\sqrt{n})$.

522D - Ближайшие равные

Условие: Дан статический массив и запросы вида $[l, r]$, для каждого запроса вычислить $\min(|i - j| : a[i] = a[j] \wedge i, j \in [l, r])$. $n \leq 500000$.

Реализация Мо для этой задачи выглядит сложной, поскольку добавление элементов выполняется легко с использованием массивов pre и nxt, но кажется невозможным удалить значения и восстановить минимальное значение, т.е. ответ. Поэтому рассмотрим декомпозицию на блоки как в предыдущей задаче, которая требует только добавления новых элементов во вспомогательные массивы.

Разобьем этот массив на блоки размера $O(B)$. Для каждого интервала блоков вычислим ответ, что в сумме дает предподсчёт за $O(\frac{N}{B}N)$. Разница в предподсчёте с прошлой задачей возникает из-за того, что когда мы хотим вычислить структуру для интервала с L по R, то интервал L по $R-1$ уже вычислен и нужно добавить элементы по одному из блока R, т.е. суммарная сложность создания структур $O(\frac{N}{B}\frac{N}{B}B) = O(\frac{N}{B}N)$.

Заметьте, что если мы хотим найти минимальное расстояние и фиксируем i, то единственные возможные значения j, которые могут внести вклад в ответ, это $pre[i]$ и $nxt[i]$. Таким образом, при расширении предподсчитанного блока мы проверяем, находятся ли $pre[i]$ и $nxt[i]$ в отрезке запроса на который нам нужно ответить, и если да, пытаемся обновить минимум, что дает ответ на запрос за $O(B)$.

Общая асимптотика составляет $O(\frac{n^2}{B} + qB)$, при выборе $B = O(\sqrt{n})$ получаем общую временную сложность $O((n+q)\sqrt{n})$. Заметьте, что ввод и вывод для этой задачи составляют большую часть от константы, т.к. мы выполняем небольшое количество арифметических операций, а удалений вовсе не делаем.

617E - XOR и любимое число, онлайн версия

Условие: Дан статический массив, число k и запросы вида $[l, r]$, для каждого запроса вычислить количество пар $(i, j) \in [l.r]$ таких, что $a[i] \oplus a[j] = k$.

1) Построение структуры для всех пар блоков:
Заметим, что $a[i] \oplus a[j] = k \Leftrightarrow a[i] \oplus k = a[j]$. Будем для каждого начала интервала блоков перебирать последовательно слева направо элементы i, добавлять к текущему ответу количество ранее встреченных $a[i] \oplus k$, а также отмечать в структуре, что встретили элемент $a[i]$.

Каждый элемент мы рассмотрим $\frac{n}{B}$ раз, для сохранения ответов нам потребуется дополнительная память порядка $O((\frac{n}{B})^2)$.

2) Вычисление ответов на запрос:
Рассматриваемый интервал может включать как элементы в целых блоках текущего интервала, так и максимум $O(B)$ элементов в крайних блоках, которые не попадают целиком внутрь запроса, так что мы можем просто поддерживать массив этих $O(B)$ элементов. Обратите внимание, что для интервала блоков, количество элементов со значением x в блоках от l до r можно вычислить путем вычитания $(cnt[1..r] - cnt[1..l-1])$. Значит мы предподсчитаем количество элементов x в блоках от 1 до r, чтобы при обработке запроса мы быстро получали ответ. Предподсчет займёт $O(\frac{n}{B} \max A)$ памяти и $O(n + \frac{n}{B} \max A)$ времени, ведь нам нужно скопировать структуру после подсчета каждого из блоков. Тут $\max A$ – максимальное значение элемента в массиве A. Заметьте, что существует не более $O(n)$ возможных значений для $a[i]$ и $a[i] \oplus k$ – получаем $O(\frac{n}{B} n)$ памяти и такую же оценку по времени на эту часть предподсчета.

Ответы на сами запросы вычисляются за $O(B)$, ведь нам нужно обрабатывать все элементы в крайних блоках, а также взять один ответ для отрезка блоков, целиком попадающих в запрос. При $B = \sqrt{n}$ общая асимптотика будет $O(\frac{n}{B} n + \frac{n}{B} n + qB) = O(n\sqrt{n} + q\sqrt{n})$ времени и $O((\frac{n}{B})^2 + \frac{n}{B} n) = O(n\sqrt{n})$ памяти.

Запрос уникальных элементов на отрезке, онлайн

Условие: Запрос уникальных элементов на отрезке, но онлайн :)

Очевидно, что предподсчёт и использование массивов $pre[i]$ и $nxt[i]$ приводят к времени выполнения $O((n + q)\sqrt{n})$.

Заметьте, что предподсчёт осуществляется за $O(\frac{n}{B} n)$ (как в задаче 552D), потому что нам нужно вычислить ответы для всех интервалов блоков, которые мы сохраним используя $O((\frac{n}{B})^2)$ значений. С учётом этого, асимптотика будет лучше, если мы будем использовать методы декомпозиции, так как можно подобрать такое B, чтобы улучшить и итоговую сложность. Даже лучше, чем $O((n + q)\sqrt{n})$.

Рассмотрим определение ответа для произвольного отрезка $[l, r]$:

$$A_{[l,r]} = \sum_{i=l}^{r} [\mathrm{pre}[i] + 1 \leq l]$$

Разместим точки $(pre[i]+1, i)$ на декартовой плоскости. Определим функцию $p(x, y)$ как количество точек, таких, что их x-координата меньше x и y-координата меньше y, т.е. как двумерную префиксную сумму. Теперь наш ответ:

$$A_{l,r} = p(l, r)$$

Но вычисление $p(x, y)$ занимает $O(n^2)$. Рассмотрим преобразование выражения A с учетом декомпозиции на блоки размера B:

$$A_{[l,r]} = \sum_{i=l}^{r} [\frac{\text{pre}[i]}{B} + 1 \leq \frac{l}{B}]$$

Теперь это префиксная сумма по точкам $(pre[i]/B + 1, i/B)$ с округлением вниз. Это, очевидно, можно вычислить за $O((n/B)^2)$.

Рассмотрим оптимальный выбор B. Временная сложность:

$$O\left(\frac{n^2}{B^2} + qB\right)$$

Выбирая $B = O(\sqrt[3]{n})$, получаем общую временную сложность

$$O((n+q)\sqrt[3]{n})$$

Этот метод работает быстрее, чем самый лучший известный метод с использованием персистентного дерева отрезков из-за маленькой константы для $n = 10^6$. Памяти тоже нужно гораздо меньше.

8 Более сложные примеры

Оптимизация перебора: 911G - Запросы на массовое обновление

Условие: Дан массив и операции вида l, r, x, y. Задача установить $a[i] = y$ для всех i, таких что $i \in [l, r]$ и $a[i] = x$. Здесь $1 \leq a[i], x, y \leq 10^5$.

Если $l = 1$ и $r = n$, мы можем выполнять эти операции довольно быстро, сохраняя индексы каждого значения в векторе, и выполняя слияние путем добавления меньшего множества к большему.

В общем случае, разобьем массив на блоки размера B. Для блоков, которые полностью содержатся в отрезке одной операции, выполняем слияние от меньшего к большему. Иначе, восстанавливаем этот блок и изменяем его прямым перебором.

Автор не знает (п. п.: и я тоже), как доказать асимптотику строго, но мы можем оценить её сверху. Оценим сложность, если мы не выполняем перебор, то есть используем только приливайки в блоках. Для каждого блока потребуется в общей сложности $O(B \log B)$ времени, потому что каждый раз, когда элемент перемещается, размер массива, в котором он находится, как минимум удваивается. Общее время в таком случае будет $O(Q\frac{N}{B} + \frac{N}{B} B \log B) = O(Q\frac{N}{B} + N \log B)$, ведь для каждого запроса нужно пройтись

по всем блокам, а дальше суммарно по всем запросам для каждого блока будет произведено $O(B \log B)$ операций.

На то, чтобы восстанавить блок и сделать изменения напрямую, потребуется $O(B)$ операций. При этом, мы нарушили потенциал этого блока, поэтому в будущем нам снова может потребоваться $B \log B$ операций для приливайки, а значит мы можем заложить, что операция требует $O(B \log B)$ времени. Отсюда общая сложность $O(Q\frac{N}{B} + N \log B + QB \log B)$.

Для упрощения, предположим, что $Q = O(N)$, тогда асимптотика будет $O(\frac{N^2}{B} + N \log B + NB \log B) = O(\frac{N^2}{B} + NB \log B)$. Выбирая $B = \sqrt{\frac{N}{\log N}}$, получаем:

$$O\left(\frac{N^2}{\sqrt{\frac{N}{\log N}}} + N\sqrt{\frac{N}{\log N}} \log \sqrt{\frac{N}{\log N}}\right) \leq O(N\sqrt{N \log N})$$

Заметьте, что логарифм находится под корнем.

Запрос расстояния между значениями

Условие: Дан статический массив и запросы вида x, y, найти $\min(|i - j| : a[i] = x, a[j] = y)$.

Существует не более $O(\sqrt{n})$ значений, которые встречаются чаще, чем $O(\sqrt{n})$ раз. Рассмотрим пару x, y так, что хотя бы одно из x, y встречается чаще, чем $O(\sqrt{n})$ раз. Мы пытаемся предподсчитать ответы для всех таких пар x, y, так как их не более $O(n\sqrt{n})$ пар.

Для всех x, которые встречаются чаще, чем $O(\sqrt{n})$ раз, мы обходим массив (используя x) один раз вперед и один раз назад. Мы храним последнее местоположение, где x встречался в направлении, в котором мы обходим, и затем пытаемся обновить ответ для x и $a[i]$ (где i - индекс указателя на текущий элемент).

Кроме того, для каждого значения мы храним отсортированный вектор индексов, где данное значение встречается.

Когда мы отвечаем на запросы, если ответы уже предподсчитанны, мы используем их напрямую; В противном случае мы запускаем два указателя на соответствующие векторы, которые мы сохранили. Общая временная сложность $O((n + q)\sqrt{n})$.

Ynoi2008 - rplexq

Условие: Дано корневое дерево и запросы вида (l, r, x), для каждого запроса найти количество пар i, j таких, что $l \leq i < j \leq r$ и LCA (наименьший общий предок) i и j равен x.

Прежде всего, отвечаем на эти запросы оффлайн и присваиваем значения $[l, r]$ соответствующим вершинам x. Создадим СНМ (система непересекающихся множеств) на дереве, а также будем поддерживать неявное дерево отрезков на индексах вершин, которые принадлежат поддереву, которые впоследствии будем сливать вместе путем добавления меньшей КС к большей.

Если мы не учитываем отрезки $[l, r]$, то количество пар, которые имеют LCA в поддереве вершины x, равно

$$C_(^2 sz_x) - \sum C_(^2 sz_j)$$

где j - ребенок x.

Когда у вершины небольшое количество детей, мы можем ответить на запросы с помощью перебора, так как запрос префиксной суммы в неявном дереве отрезков работает быстро. Но когда у вершины много детей **и** к ней много запросов, это слишком медленно. Будем называть вершину *лёгкой*, если у неё мало детей, иначе – *тяжелой*.

Сначала мы найдём самые тяжелые вершины, чтобы гарантировать, что количество вершин, по которым мы разбиваем граф около $O(n \log n)$. Затем берем легких детей и конденсируем их (сжимаем их в одну вершину). Теперь можем запустить алгоритм Мо на этих индексах, где мы храним

$$\sum C_(^2 colors)$$

Из-за большого времени на создание структур для алгоритма Мо, может быть лучше использовать жадный перебор, когда количество запросов или детей маленькое.

Заметьте, что сложность для алгоритма Мо с размером массива n и количеством запросов m равна $O(n\sqrt{m})$. Из-за оценки

$$\sum n_i \sqrt{m_i} \leq (\sum n_i)\sqrt{\max m}$$

общая временная сложность ограничена сверху $O(n \log n \sqrt{m})$.

Источники

- Оригинал статьи
- алгоритм Мо
- задачи коммивояжера
- DSU/СНП/Система непересекающихся множеств