MIPT, сборы 2014, день #3

Тема: структуры данных корневая оптимизация, пересечение полуплокостей

14 ноября, Сергей Копелиович

---

1. Корневая оптимизация

Рассмотрим задачу про поиск подстрок $s_1, s_2, \ldots, s_k$ одинаковой длины l в строке t. Для каждого i нужно узнать, встречается ли s_i в t, как подстрока. Мы умеем решать такую задачу за линейное время, точнее говоря, за $\mathcal{O}(lk + |t|)$ с помощью хеш-таблицы полиномиальных хешей строк s. Здесь вы можете правильно заметить, что подобные задачи следует решать алгоритмом Ахо-Корасик, но давайте ненадолго представим, что мы его не знаем, зато про хеши уже слышали.

Пусть теперь даны строки произвольных длин. Как для новой задачи использовать наше решение с хешами? Идея корневой оптимизации в том, что если суммарная длина всех строк равна L, то строки длины до $\sqrt{L}$ мы можем перебрать за $\mathcal{O}(L+|t|\sqrt{L})$, а строк длины больше $\sqrt{L}$ всего лишь $\mathcal{O}(\sqrt{L})$, поэтому суммарное время работы решения "сгруппируем строки по длине, и для каждой длины решим за линейное время" получается также $\mathcal{O}(L+|t|\sqrt{L})$.

Такую идею можно использовать не только для строковых задач, но и, например, для задач про деревья. В дереве из n вершин может быть много вершин степени не более $\sqrt{n}$, а вершин степени хотя бы $\sqrt{n}$ не более $\sqrt{n}$.

2. Корневая оптимизация на массиве

Рассмотрим абстрактную задачу "у нас есть массив a, и мы хотим делать с ним много разных сложных запросов на отрезках". Начнем с простого. Запрос #1: сумма на отрезке. Запрос #2: поменять значение в точке. Будем за $[f(n), g(n)]$ обозначать то, что некая структура данных на запросы первого типа умеет отвечать за $f(n)$, а на запросы второго типа за $g(n)$. Наша задача решается деревьями отрезков и Фенвика за время $[\mathcal{O}(\log n), \mathcal{O}(\log n)]$. Мы сейчас решим ее за время $[\mathcal{O}(1), \mathcal{O}(\sqrt{n})]$ и за время $[\mathcal{O}(\sqrt{n}), \mathcal{O}(1)]$. Обозначим $k = \lfloor\sqrt{n}\rfloor$.

Решение #0. Заметим, что мы можем насчитать суммы на префиксах (частичные суммы) исходного массива `sum[i+1]=sum[i]+a[i]`, сумма на отрезке $[l, r]$ тогда равна `sum[r+1]-sum[l]`, а при каждом изменении массива будем пересчитывать весь массив `sum`. Кроме того мы можем ничего дополнительного не хранить, и сумму искать за линейное время. Это решения за $[\mathcal{O}(n), \mathcal{O}(1)]$ и $[\mathcal{O}(1), \mathcal{O}(n)]$ соответственно.

Решение #1 за $[\mathcal{O}(k), \mathcal{O}(1)]$. Будем поддерживать массив s, $s[i]$ равно сумме $a[j]$ по $j \in [ki..k(i+1))$. Запрос "сумма на отрезке": отрезок разбивается на "хвосты" и $\mathcal{O}(k)$ кусков с уже известной суммой. Запрос "поменять значение в точке":

```
set(i,x) { s[i/k] += x-a[i]; a[i] = x; }
```

Решение #2 за $[\mathcal{O}(1), \mathcal{O}(k)]$. Поддерживаем тот же массив s, а также частичные суммы массива s. Кроме того на каждом отрезке $[ki..k(i+1))$ насчитаем частичные суммы. Чтобы сделать изменение в точке, нужно целиком пересчитать два массива частичных сумм (суммы конкретного куска, суммы на s). Чтобы узнать сумму на отрезке, достаточно разбить его на два "хвоста" и все остальное. На каждой из трех частей мы умеем считать сумму за $\mathcal{O}(1)$.

Решение #3 за $[\mathcal{O}(k), \mathcal{O}(k)]$. Будем поддерживать суммы на префиксах `sum[i+1]=sum[i]+a[i]` и массив изменений, произошедших с массивом с тех пор, как мы считали суммы на префиксах, – *changes*. Одно изменение – пара $\langle i, x\rangle$, обозначающая операцию `a[i]+=x`. Будем поддерживать свойство $|changes| \le k$. Запрос "сумма на отрезке": сумма на отрезке $[l, r]$ в исходном массиве была равна `sum[r+1]-sum[l]`, за $\mathcal{O}(|changes|)$ мы можем посчитать, насколько она с тех пор изменилась. Запрос "поменять значение в точке": чтобы в точку i записать новое значение x, добавляем в *changes* пару $\langle i, x - a[i]\rangle$, в $a[i]$ записываем x. Если после этого $|changes| > k$, за $\mathcal{O}(n)$ строим суммы на префиксах текущего массива a и очищаем список *changes*. Эту операцию назовем `rebuild`. Заметим, что `rebuild` мы будем вызывать не чаще, чем один раз за k запросов, поэтому амортизированное время обработки одного запроса равно $\mathcal{O}(\frac{n}{k}) = \mathcal{O}(\sqrt{n})$.

Последнюю технику будем называть "отложенные операции". Для данной задачи такая техника не имеет никаких плюсов, но пригодится нам в дальнейшем.

3. Корневая оптимизация на массиве: split & rebuild

Решим теперь более сложную задачу: операции с массивом будем делать такие

1. `Insert(i, x)` – вставить x на позицию i.
2. `Erase(i)` – удалить i-й элемент массива.
3. `Sum(l,r,x)` – посчитать на отрезке $[l, r]$ сумму элементов больших x.
4. `Reverse(l,r)` – перевернуть отрезок $[l, r]$.

Чтобы решить такую задачу, для начала научимся отвечать на запрос `Sum(l,r,x)` (единственный запрос с осмысленным ответом), применный ко всему массиву, то есть на запрос `Sum(0,n-1,x)`. Пусть пока других типов запросов нет, есть только один запрос `Sum`. Тогда достаточно поддерживать отсортированную версию исходного массива и частиные суммы на ней. Ответ на запрос – бинарный поиск по массиву и обращение к частичным суммам. Время построения структуры данных $\mathcal{O}(n \log n)$, время ответа на один запрос $\mathcal{O}(\log n)$.

Теперь решим полную версию задачи. Есть массив $a[0..n)$. Будем также в каждый момент времени хранить разбиение массива на отрезки $T = [A_1, A_2, \dots, A_m]$. Для каждого отрезка у нас будут храниться две версии – исходная и отсортированная с частичными суммами. Постараемся поддерживать два свойства: $\forall i\colon |A_i| \le \sqrt{n}$ и $m < 3\sqrt{n}$. Изначально разобьем массив на $k = \sqrt{n}$ отрезков длины $\sqrt{n}$. Для каждого из k отрезков за $\sqrt{n}\log n$ вызовем операцию `build`, которая построит отсортированный массив и частичные суммы. Теперь напишем основную операцию `split(i)`, которая возвращает такое j, что i-й элемент – начало j-го отрезка. Если точка i не является началом ни одного из отрезков, найдем такой $A_j = [l, r)$, что $l < i < r$, и разобьем его на два отрезка $B = [l, i)$ и $C = [i, r)$. Для отрезков B и C запустим `build`, получим новое разбиение массива на отрезки: $T' = [A_1, A_2, \dots, A_{j-1}, B, C, A_{j+1}, \dots, A_m]$. Время работы операции `split(i)` равно $\mathcal{O}(k)+\mathcal{O}(\texttt{build}(\frac{n}{k}))$, то есть при $k = \sqrt{n}$ получается $\sqrt{n}\log n$. Здесь мы пользуемся тем, что внутри T хранятся не сами отрезки, а ссылки на них, поэтому "копирование" отрезков происходит за $\mathcal{O}(1)$. Теперь у нас получится через `split(i)` просто выразить все остальные операции.

```
vector&lt;Segment*&gt; T;
int split( int i ) { ... }
void Insert(i, x) {
  a[n++] = x;
  int j = split(i);
  T.insert(T.begin() + j, new Segment(n-1, n));
}
void Erase(i) {
  int j = split(i);
  split(i + 1);
  T.erase(T.begin() + j);
}
int Sum(l, r, x) { // [l, r]
  l = split(l), r = split(r + 1); // [l, r)
  int res = 0;
  while (l <= r)
    res += T[l++].get(x); // бинарный поиск и обращение к частичным суммам
  return res;
}
```

Чтобы обрабатывать запросы типа `Reverse`, нам нужно для каждого отрезка хранить дополнительный флаг "развернут ли отрезок", код операции `split(i)` несколько усложнится, остальная часть не изменится.

```
void Reverse(l, r) {
  l = split(l), r = split(r + 1);
  reverse(T + l, T + r)
  while (l <= r)
    T[l++].reversed ^= 1;
}
```

Сейчас у нас есть решение, которое начинает с $k = \sqrt{n}$ отрезков, и выполнение каждого запроса увеличивает k не более чем на 2. Через k запросов может произойти ситуация, что $k \geq 3\sqrt{n}$ (стало в три раза больше), в этот момент мы за $\mathcal{O}(n \log n)$ перестроим всю структуру (сделаем `rebuild`). Амортизированное время выполнение операции `rebuild` равно $\frac{n \log n}{k} = \sqrt{n}\log n$. Итого наше решение обрабатывает все запросы за амортизированное время $\mathcal{O}(\sqrt{n}\log n)$.

Можно ускорить. Для этого заметим, что `split(i)` можно делать за линейное время, то есть за $\mathcal{O}(k + \frac{n}{k})$, а операцию `rebuild` можно делать за время $\mathcal{O}(n)$. Тогда если выбрать количество отрезков k равным не $\sqrt{n}$, а $\sqrt{n/\log n}$, амортизированное время ответа на один запрос типа `Sum` будет $\mathcal{O}(S+G+R)$, где S – время `split(i)`, равно $\mathcal{O}(\frac{n}{k}+k)$, G – время внутреннего цикла функции `Sum`, равно $\mathcal{O}(k\log n)$, B – амортизированное время на один `rebuild`, равно $\mathcal{O}(\frac{n}{k})$. Получаем в сумме $\mathcal{O}(\sqrt{nlogn})$ на запрос.

4. Пересечение полуплоскостей

Решим одну сложную задачу acm.timus.ru:1390. Условие такое: мы стоим на плоскости в точке $(0,0)$. Время от времени на плоскости появляются стенки – отрезки, не проходящие через $(0,0)$. Стенки могут пересекаться. Время от времени мы выпускаем в произвольных направлениях пули. Пуля летит по прямой, пока не каснется стены. Ответ на запрос типа "пуля" – расстояние, которое она пролетит (возможно, бесконечность).

В итоге мы научимся n запросов произвольного вида обрабатывать за время $\mathcal{O}(n\log^2 n)$. Но для начала попробуем понять, как вообще подступиться к задаче. Задача сложная. Запросы разных типов вперемешку, стены пересекаются... Попробуем задачу упростить. Пусть уже какие-то стены есть, а новые не появляются, более того, пусть все стены – прямые.

Задача теперь имеет следующий вид: даны прямые, которые высекают на плоскости выпуклый многоугольник, содержащий точку $(0,0)$, нужно уметь быстро по лучу из точки $(0,0)$ находить точку пересечения с границей многоугольника. Решение задачи: пересечем полуплоскости за $\mathcal{O}(n\log n)$, получим многоугольник. Затем каждый запрос мы обработаем бинарным поиском за время $\mathcal{O}(\log n)$. Обсудим решение подробнее, начнем с бинарного поиска. Возьмем угол α любой из вершин многоугольника. Для каждой другой вершины возьмем ее угол из промежутка $[\alpha,\alpha+2\pi)$, выпишем эти числа в возрастающем порядке, получим массив углов a. Чтобы понять, куда попадет пуля, пущенная в направлении (x,y), посчитаем угол $\beta\in[\alpha,\alpha+2\pi)$, найдем бинарным поиском минимальное $i\colon a[i]\le\beta<a[i+1]$ и пересечем отрезок, образованный точками i, $i+1$ с лучом, по которому летит пуля. Время $\mathcal{O}(\log n)$. Теперь пересечение полуплоскостей. Во-первых, чтобы они точно в пересечении давали конечный многоугольник, добавим bounding box, полуплоскости $-M\le x, x\le M, -M\le y, y\le M$, где M выбирается из ограничений на координаты исходных точек. Теперь отсортируем все прямые по углу нормали (угол от 0 до 2π). Из прямых с одинаковым углом оставим лишь ту, что ближе к $(0,0)$. Затем будем добавлять прямые в таком порядке в стек. Перед добавлением очередной прямой, несколько верхних прямых со стека нужно снять. Пусть сейчас верхние прямые на стеке – l_1, l_2. Верхнюю прямую нужно снять со стека, если точка $p=intersect(l_1,l_2)$ лежит по другую сторону чем $(0,0)$ от новой прямой. В стеке наращивается многоугольник-ответ. Чтобы многоугольник замкнулся в цикл, добавим второй раз все прямые в таком же порядке. Теперь содержимое стека = хвост + ответ + хвост. Чтобы отрезать хвосты, рассмотрим точки пересечения соседних прямых на стеке и возьмем ту, которая встречается два раза. Между этими двумя позициями в стеке лежит многоугольник-ответ. Время построения $\mathcal{O}(sort)+\mathcal{O}(n)$.

Усложним задачу: пусть теперь прямые добавляются. Можно применить технику отложенных операций и обрабатывать каждый запрос типа "пуля" за $O(\sqrt{n})$. Как только отложенных добавлений прямых станет больше чем $\sqrt{n}$, вызываем процедуру `rebuild`, которая отработает за $\mathcal{O}(n)$, так как старые прямые уже отсортированы, а новые $\sqrt{n}$ штук мы можем примерджить за $\mathcal{O}(\sqrt{n}\log n+n)$. Итого, время обработки n запросов будет $\mathcal{O}(n\sqrt{n})$.

Можно поступить по-другому и хранить вершины многоугольника не в отсортированном по углу массиве, а в сбалансированном дереве поиска. Тогда, чтобы добавить новую прямую с направляющим вектором $\vec{v}$ и нормалью $\vec{n}$ (направление от $(0,0)$), нужно научиться находить вершину многоугольника $\vec{p}$: скалярное произведение $\langle \vec{n}, \vec{p} \rangle$ максимально. Это можно сделать бинарным поиском по сортированному массиву. Или, теперь у нас дерево поиска, спуском вниз по дереву поиска. Если точка $\vec{p}$ лежит, с той же стороны от прямой, что и $(0,0)$, ничего делать не нужно, иначе нужно удалить $\vec{p}$ и, возможно, еще несколько смежных с ней вершин многоугольника. Вместо них добавятся две точки. Удалять будем в цикле, пока очередная точка не будет с той же стороны, что и $(0,0)$ от прямой. В конце добавим точку пересечения стороны и отрезка, на котором мы остановили процесс удаления. Время добавления новой прямой получилось $\mathcal{O}(\log n + k \log n)$, где k – количество удаленных точек. Каждая точка удалится только один раз, поэтому суммарное время обработки n запросов – $\mathcal{O}(n \log n)$. С точки зрения скорости работы предыдущее решение гораздо хуже. С точки зрения простоты реализации предыдущее решение в разы проще.

И еще раз усложним задачу: теперь у нас не прямые, а отрезки. Сперва решим задачу в offline (все запросы известны заранее). Тогда мы знаем множество всех возможных углов точек концов отрезков. Отсортируем их и построим на массиве углов дерево отрезков. Каждой вершине дерева отрезков соответствует некоторый диапазон углов. Когда мы добавляем стену, деревом отрезков она разбивается на $\mathcal{O}(\log n)$ вершин дерева отрезков, в каждой из которых стена идет ровно от левой границы диапазона углов до правой, то есть ведет себя как прямая. Итого: в каждой вершине дерева отрезков поддерживаем пересечение полуплоскостей. Заметим, чтобы построить пересечение полуплоскостей внутри вершины дерева отрезков, нужно строить не многоугольник (замкнутую ломаную), а ломаную от левого луча до правого (можно считать, что разница углов не более $\frac{\pi}{4}$). Поэтому процедура построения пересечения полуплоскостей значительно упрощается – достаточно одного прохода со стеком, каждую прямую добавить один раз. По второму разу добавлять прямые, а затем выделять цикл не нужно. Итак, у нас есть дерево отрезков, в каждой вершине которого лежит пересечение полуплоскостей. Обработка запроса "пуля": угол пули попал в $\log n$ вершин дерева отрезков, в каждой из этих вершин сделаем за $\mathcal{O}(\log n)$ запрос к структуре на полуплоскостях. Суммарное время работы $\mathcal{O}(\log^2 n)$. Обработка запроса "добавить стену": разбили ее на $\log n$ вершин дерева отрезков и в каждую вершину добавили "прямую" в структуру полуплоскостей. Суммарное время работы $\mathcal{O}(\log^2 n)$. Используемая память – $O(n \log n)$.

Теперь, чтобы получить решение в online, достаточно от обычного дерева отрезков перейти к динамическому. Изначально дерево отрезков содержит только корень. Затем по ходу запроса к дереву отрезков, если мы пытаемся пойти в не существующую вершину, создаем ее. Процесс спуска вниз по дереву отрезков останавливается, когда длина отрезка меньше ε.

Можно было тоже самое сделать, используя отложенные операции, тогда время обработки запроса было бы $\mathcal{O}(k + \log^2 n)$. Где k – количество отложенных операций. Структуру мы строим за время $\mathcal{O}(n \log n)$ – один раз отсортировали стены-прямые, затем в каждой вершине дерева отрезков за линейное время построили пересечение соответствующих полуплоскостей. Оптимально максимальное k выбрать равным $\sqrt{n \log n}$. Итого суммарное время обработки n запросов равно $(n \log n)^{3/2}$.