Разбор задачи «Московские числа»

Автор: Николай Будин
Разработчики: Филипп Рухович, Дмитрий Саютин, Николай Будин

Если в заданной строке не было знаков вопроса, то значение числа найти легко — нужно лишь для каждой буквы в строке понять, есть ли справа от нее строго б**о**льшая буква, и прибавить или вычесть её номинал из ответа в зависимости от этого. Нахождение большей буквы перебором дает 6 баллов; однако если идти справа налево и хранить максимум на суффиксе, то можно набрать 15 баллов.

Третью группу тестов легко пройти, перебирая все возможные значения букв, которые можно найти под вопросиками, и выбирая оптимальный вариант. Отметим, что чтобы уложиться в ограничения по времени, необходимо вычислять значение каждого из получаемых чисел за линию.

Набрать же баллы по оставшимся двум группам помогает динамическое программирование. Действительно, пусть $dp[i][j]$, где $1 \leqslant i \leqslant n$, $0 \leqslant j < \Sigma$ — это максимально возможная стоимость i-го суффикса $S[i..n]$ строки S в предположении, что максимальной из букв на этом суффиксе является буква j. Будем вычислять такую динамику справа налево. При добавлении к строке слева i-го символа для определения знака его стоимости достаточно знать максимум среди символов от $i+1$ и дальше; в силу этого, любое $dp[i][j]$ может быть пересчитано за $O(\Sigma)$, где Σ — размер алфавита (в нашем случае 26). Таким образом, сложность такого решения $O(n\Sigma^2)$. Аккуратная реализация именно такого решения давала полный балл; однако отметим, что с помощью максимумов на префиксах и суффиксах массивов $dp[i][\ldots]$ длины 26 для разных i, решение можно было оптимизировать до $O(n\Sigma)$.

Разбор задачи «Гаджеты на дереве»

Автор: Максим Ахмедов
Разработчики: Арсений Кириллов, Азат Исмагилов

$m = 2n - 4$

Сначала обсудим решение в предположении, что $m = 2n - 4$. Заметим, что рёбра можно покрасить в чёрный и белый цвет таким образом, что в любом гаджете будет ровно одно чёрное и ровно одно белое ребро. Действительно, сначала покрасим в чёрные и белые цвета вершины леса так, чтобы у любого ребра концы имели разные цвета. После этого можно назначить ребру (u, v) тот же цвет, что у вершины u.

Заметим, что если бы в дереве были представлены все возможные $2n-2$ рёбер, ответ бы всегда существовал. Действительно, в таком ориентированном графе существует эйлеров цикл, который легко строится обходом в глубину. Этот цикл имеет чётную длину, значит он легко разбивается на $n-1$ гаджет.

Из этого наблюдения находится решения и для случая $m = 2n-4$. Построим тот же эйлеров цикл для полного дерева. Два удалённых ребра могут разбивать этот цикл либо на два отрезка чётной длины, либо на два отрезка нечётной длины. В первом случае также легко получить разбиение на гаджеты; в оставшемся случае легко видеть, что два удалённых ребра были одного цвета, а значит рёбер какого-то цвета среди оставшихся больше, чем другого; это доказывает, что ответ в таком случае – «невозможно».

Общий случай

Рассмотрим разбиение рёбер на пары (гаджеты) в ответе. Посмотрим произвольную вершину v и её поддерево; если наша вершина не является корнем, то рёбра между родителем p вершины v и самой v для удобства также включим в поддерево. Некоторые пары рёбер будут находиться целиком в поддереве, некоторые пары будут находиться целиком вне, а какие-то пары могут пересекать эту границу — одним концом в поддереве, другим снаружи. Заметим, что такие пары бывают только следующего вида: либо пара рёбер идущая вглубь нас $(u \to p, p \to v)$, либо пара рёбер идущая

наружу от нас ($v \to p, p \to u$), u обозначает какую-то вершину смежную с p, отличную от v. Обратите внимание, что u может быть как предком p, так и одной из детей p; кстати, в английском языке в такой ситуации используется удобный термин "sibling" (брат или сестра), вершина u является "sibling"-ом вершины v.

Однако заметим, что если у нас будет и пара внутрь, и пара наружу, то то можно было бы вместо этого организовать пару рёбер из двух нижних и отдельно пару из двух верхних. Поэтому можно запретить проводить две такие пары одновременно. Таким образом, получается что границу будет пересекать или ноль пар, или одна пара внутрь, или одна пара наружу. Причём заметим, что случай «ноль пар» от «одна пара в какую-то сторону» определяется просто чётностью количества рёбер в поддереве. На самом деле в процессе решения окажется, что ориентация пары (если она должна быть) тоже определяется однозначно.

Воспользуемся методом динамического программирования `dp[v] = 1/0` — можно ли разбить поддерево на пары. Если наша чётность подразумевает, что есть пара внутрь или наружу, разрешим себе «забрать» гипотетическое ребро вне поддерева в эту пару. На самом деле с учётом того, что `dp[v] = 0` для любой вершины означает ответ «невозможно» для всего теста, это удобнее делать рекурсивным поиском в глубину, который сразу обрывается, если в какой-то вершине разбивать не вышло. Этот поиск в глубину ходит по дереву, в графе, который был до удалений.

Как же осуществить разбиение, если мы сейчас находимся в вершине v? Во-первых рекурсивно разбиваем детей, и если там не вышло разбить (даже с учётом того, что мы разрешаем заимствовать внешнее ребро), то сразу выходим. Будем считать, что рекурсивный обход возвращает нам помимо значения динамического программирования также тип ребёнка — одно из чисел `0`, `+1`, и `-1`, реализующее описанный выше тип пересечения границы поддерева ребёнка.

Заметим, что у детей `0` всё уже хорошо, с ними ничего делать не надо. Рассмотрим мультимножество, в которое поместим все типы детей, отличные от `0`. Заметим, что их можно объединять в пары, компенсируя недостачу входящего или выходящего ребра. Также в мульимножество следует добавить `+1` или `-1` на все рёбра между p и v; они также могут участвовать в объединении; знак числа зависит от ориентации ребра (ведёт в v или из v).

После того, как мы образем максимальное количество пар (`+1`, `-1`), возможны следующие варианты:

- Все объекты знаков `+1` и `-1` объединились в пары. В таком случае мы являемся вершиной типа `0`.
- У нас остался один объект, предположим без ограничения общности, что его знак — `+1`. Заметим, что у нас не было никаких ограничений на то, кого с кем объединять в пары; значит, среди объектов `+1` можно оставить произвольный. Сделаем, так чтобы осталось как раз ребро в предка $v \to p$; тогда в рекурсивном процессе наш предок сможет его объединить в пару с кем-то ещё. Если если же такого ребра не было (удалено или мы корень), то ответ — «невозможно». Действительно, единственный способ добить это ребро до пары — это с помощью ребра вне нашего поддерева, но чтобы иметь возможность образования такой пары, нескомпенсированный `+1` обязан быть ребром в предка.
- У нас осталось более одного объекта одного знака. Тогда тоже ничего сделать нельзя.

Также заметим, что знак действительно определяется однозначно, если мы знаем знаки всех детей и ситуацию с существованием рёбер наверх.

Если неоптимально реализовать процедуру слияния, можно получить решение, которое пройдёт первые несколько групп. Эту же процедуру несложно реализовать и за линейное время от степени вершины; в таком случае решение пройдёт на полный балл.

Разбор задачи «Вырубка деревьев»

Автор и разработчик: Алексей Упирвицкий, Алексей Перевышин

Тривиальные подгруппы

Для начала давайте заметим, что если дерево можно повалить, то это всегда выгодно сделать, так как это не может помешать валить другие деревья. Таким образом, приходим к следующему решению: пока на отрезке есть дерево, которое в данный момент можно повалить, валим его и продолжаем алгоритм. Тривиальная реализация будет работать за $O(n^2)$ на запрос, более аккуратная реализация, которая поддерживает множество деревьев, которые сейчас можно повалить, будет работать за $O(n)$ на запрос.

Альтернативное решение за линейное время

Чтобы получить более быстрое решение, давайте для начала попробуем пересчитывать ответ при добавлении одного дерева к текущему отрезку. Пусть у нас есть некоторый отрезок $l..r$, и мы повалили все деревья, которые смогли. Теперь разрешим валить также дерево $r+1$. Если это дерево нельзя повалить, то ответ остается таким же. Если его можно повалить, то валим его и далее жадно валим все, что можно. Очевидно, что нужно проверять только самое правое из оставшихся деревьев, поэтому это можно делать быстро, храня стек неповаленных деревьев; легко видеть, что средняя амортизационная сложность добавления одного дерева справа составляет $O(1)$, так как каждое дерево будет добавлено и удалено не более одного раза.

Используя описанный метод, можно построить еще одно решение, которое работает за $O(n)$ на запрос, однако его также можно использовать для построения более быстрых решений.

Алгоритм МО

Чтобы решить задачу за время $O((n+q)\sqrt{n})$ воспользуемся алгоритмом МО.

Рассмотрим все запросы, чья левая граница попадает в полуинтервал $[l_i, r_i)$. Заметим, что мы не можем тривиально удалять дерево из рассматриваемого отрезка. Поэтому сделаем следующее: исходно поставим $l = r = r_i$ и будем двигать правый указатель до нужной точки, поддерживая дек неповаленных деревьев. Теперь, когда нужно обработать запрос и правая рассматриваемая граница совпадает с его правой границей, мы должны по 1 добавлять деревья слева. Заметим, что делать это наивно нельзя; каждое добавляемое слева дерево может привести к падению вплоть до всех деревьев, а аргумент про амортизационную стоимость более не применим, т.к. отдельное «дорогое» для добавления дерево может быть многократно обработано в ходе работы алгоритма МО. Для эффективного добавления дерева слева нужно придумать некий способ сжатого представления деревьев.

Что это значит? Когда мы добавляем дерево слева, старые деревья могут упасть только налево. Так как деревьев в блоке всего $O(\sqrt{n})$ - их можно обрабатывать за линию. Теперь мы хотим узнать, сколько деревьев из отрезка $[r_i, r]$ упадут. Заметим, что дерево x упадет если ему будет достаточно расстояния и упадет предыдущее неповаленное. Таким образом, для каждого для каждого дерева можно предпосчитать, какое самое правое дерево слева должно упасть, чтобы мы могло упасть налево. Когда мы идем стеком нужно заменять это значение на минимум из него и значения для предыдущего элемента в стеке. Таким образом в стеке будет невозрастающая последовательность - какое дерево должно упасть чтобы мы упали. Тогда при обработке левого блока можно делать одно из двух: бинарный поиск по этим значениям, что добавит еще $O(\log(n))$ в ассимптотику, или хранить эти значения сжато.

Действительно, нас интересует лишь то, сколько деревьев с каким значением. Можно завести один массив, где добавление дерева это $+1$, а удаление -1. Тогда мы можем просто посмотреть в соответствующую ячейку и узнать, сколько деревьев упадет, если уронить все деревья с полуинтервала $[x, r_i)$.

Divide and Conquer

Чтобы решить задачу за время $O(\log n)$ на запрос, воспользуемся следующей идеей.

Рассмотрим некоторое дерево. Если его можно повалить, то его можно повалить либо налево, либо направо. Пусть его можно повалить налево. Тогда деревья справа от него этому не мешают.

Кроме этого, добавление новых деревьев слева также этому не помешает. Таким образом, есть некоторое число a_i такое, что дерево i можно повалить налево если и только если левая граница отрезка $l \leqslant a_i$. Аналогично, есть некоторое число b_i такое, что дерево i можно повалить направо если и только если правая граница отрезка $r \geqslant b_i$.

Предположим, что мы вычислили числа a_i и b_i. Тогда задача сводится к следующей: для каждого запроса (l, r) требуется найти число деревьев, для которых $l \leqslant a_i$ и $r \geqslant i$ или $l \leqslant i$ и $r \geqslant b_i$. Это можно сделать с помощью стандартной техники с помощью сканирующей прямой и дерева отрезков.

Осталось научиться вычислять числа a_i и b_i. Покажем, как вычислить числа a_i, числа b_i вычисляются аналогично. Будем вычислять числа a_i с помощью двоичного поиска.

В каждый момент времени у нас будет некоторый полуинтервал $[L..R)$ и множество деревьев, и мы знаем, что для всех деревьев в этом множестве число a_i находится в полуинтервале $[L..R)$. Выберем значение M, среднее между L и R. Запустим алгоритм со стеком, начиная с позиции M. Если дерево не удалось повалить налево, то для него $a_i < M$, если удалось, то $a_i \geqslant M$. Отнесем такие деревья к множествам A и B, соответственно.

Рассмотрим деревья из множества A. Для них нужно запустить двоичный поиск на полуинтервале $[L..M)$, при этом заметим, что деревья из множества B для любого значения из этого полуинтервала точно упадут, поэтому они не будут мешать деревьям из множества A. Таким образом, в этом рекурсивном вызове можно считать, что деревьев из множества B не существует.

Аналогично для деревьев из множества B нужно запустить двоичный поиск на полуинтервале $[M..R)$, при этом деревья из множества A не могут помешать им упасть, так как они не помешали им упасть для значения M. Таким образом, в этом рекурсивном вызове можно считать, что деревьев из множества A не существует.

Итого мы получили рекурсивный алгоритм, который каждый раз делит диапазон значений пополам, а деревья на два непересекающихся множества. Время работы такого алгоритма $O(n \log n)$, так как глубина рекурсии равна $\log n$, и общее число элементов на каждом слое рекурсии равно n.

Итоговое время работы алгоритма $O((n + q) \log n)$.

Рассмотрим альтернативный способ вычисления чисел a_i. Будем вычислять их в порядке возрастания i. Для фиксированного i изначально установим значение $a_i = i$ и будем повторять следующие действия:

1. Если дерево $a_i - 1$ не мешает дереву i упасть влево, значение a_i найдено правильно, выходим. В противном случае значение a_i точно нужно уменьшить.

2. Если дерево $a_i - 1$ может упасть вправо, не задев дерево i, уменьшим значение a_i на единицу и вернёмся к пункту 1.

3. Иначе заметим, что дерево $a_i - 1$ придётся повалить влево, а вместе с ним и все деревья от a_{a_i-1} до $a_i - 2$. В таком случае установим $a_i = a_{a_i-1}$ и также вернёмся к пункту 1.

Чтобы оценить асимптотику этого способа вычисления a_i, рассмотрим некоторое дерево j. Пусть $i_1, i_2, \ldots, i_k$ $(j < i_1 < i_2 < \ldots < i_k)$ — это номера деревьев, для которых в некоторый момент значение a_i было равно $j + 1$, но в итоге оказалось меньше.

Заметим, что если в некоторый момент $a_i = j + 1$, то все деревья от $j + 1$ до $i - 1$ включительно можно повалить либо влево, либо вправо, не задевая деревья j и i.

Рассмотрим любой индекс t $(1 \leqslant t < k)$. Из определения i_t мы знаем, что дерево i_t нельзя повалить влево, не задевая дерево j. Следовательно, его должно быть можно повалить вправо, не задевая дерево i_{t+1}.

Значит, расстояние между деревьями j и i_t не превосходит расстояния между деревьями i_t и i_{t+1}. Или, что эквивалентно, расстояние между деревьями j и i_t хотя бы вдвое меньше расстояния между деревьями j и i_{t+1}. Значит, $k = O(\log C)$, где C — максимальная координата дерева, а приведённый алгоритм вычисления a_i работает за $O(n \log C)$.

Разбор задачи «Блогеры-путешественники»

Автор: Ильдар Гайнуллин
Разработчик: Иван Сафонов

В первой подзадаче данный граф был деревом. Поэтому его можно обойти с помощью dfs и посчитать ответ для каждой вершины (так как путь от вершины 1 до нее единственный).

Для того, чтобы решить четвертую и пятую подзадачи, нужно организовать перебор всех путей из вершины 1. В четвертой подзадаче достаточно перебрать все перестановки ребер и проверить все префиксы этих перестановок. Такое решение работает за $O(m!m)$. Для того, чтобы пройти пятую подзадачу, нужно было реализовать любой перебор всех реберно-простых путей с помощью функции перебора. Утверждается, что если в графе **нет кратных ребер** и количества вершин и ребер $\leqslant 20$, то количество реберно-простых путей крайне мало.

В шестой подзадаче можно заметить, что любой интересный реберно-простой путь будет выглядеть так: [путь от 1 до j] + [путь от j до i] ($i \leqslant j$). Первая часть пути при этом проходит по минимальным ребрам между парами вершин p и $p+1$, вторая часть пути проходит по вторым минимальным ребрам между парами вершин p и $p+1$ (чтобы путь был, они должны существовать для всех $i \leqslant p < j$). Можно перебрать все $O(n^2)$ этих путей и обновить ответы для всех вершин.

Во второй подзадаче надо заметить, что цена минимального ребра для любого пути из вершины 1 будет равна 0. Значит нам нужно найти путь с минимально возможной максимальной ценой ребра на пути. Это известная задача, ее решение состоит в том, что надо найти минимальное остовное дерево и взять пути из него.

В третьей подзадаче надо заметить, что цена максимального ребра для любого пути из вершины 1 будет равна 10^9. Значит нам нужно найти путь с минимально возможной минимальной ценой ребра на пути. Построим компоненты вершинной двусвязности и дерево на них. Известно, что ребра этого дерева это мосты графа. Любой реберно-простой путь из вершины 1 в вершину p будет выглядеть как путь внутри этого дерева из компоненты вершины 1 в компоненту вершины p, а внутри компонент будет выбран какой-то путь между нужными концами. Можно доказать, что в любой компоненте вершинной двусвязности можно выбрать реберно-простой путь между заданной парой вершин, проходящий по заданному ребру. Значит внутри одной компоненты можно будет выбрать реберно-простой путь, проходящий по минимальному ребру внутри этой компоненты. Поэтому найдем в каждой компоненте минимальную цену ребра. Ответ для вершины p будет равен минимальной цене ребра по всем компонентам и ребрам на пути от компоненты вершины 1 до компоненты вершины p.

Решение третьей подзадачи очень поможет нам решить задачу в общем случае. Зафиксируем вес максимального ребра и скажем, что он будет $\leqslant t$. Тогда оставим только ребра i, такие что $c_i \leqslant t$. Среди таких ребер нам нужно для каждой вершины p найти минимально возможную цену минимального ребра на пути — f_p. Мы можем сделать это за $O(m)$ (как в шестой подзадаче). Далее обновим для каждой вершины p ответ числом $t + f_p$. Легко понять, что для каждой вершины мы найдем верный ответ, потому что каждый путь будет учтен в тот момент, когда мы рассмотрим t равным максимальной цене ребра на этом пути.

В седьмой подзадаче нужно перебрать значение t равное каждой цене ребра. Время работы решения $O(m^2)$.

В восьмой подзадаче надо заметить, что нужно оставить только такие ребра, добавление которых меняет компоненты вершинной двусвязности (если мы будем добавлять ребра в порядке возрастания цены). Легко понять, что количество таких ребер $\leqslant 2n$ (не нужно рассматривать ребро, которое при добавлении соединяет две вершины из одной компоненты вершинной двусвязности). Если оставить только такие ребра, то предыдущее решение будет работать за $O(n^2 + m \log m)$.

Далее опишем полное решение задачи.

Построим минимальное остовное дерево T. Будем идти по ребрам в порядке возрастания цены. Ограничение t на максимальную цену ребра на пути будет равно цене текущего ребра. Нам нужно научиться как-то быстро обновлять ответы для вершин числами $t + f_p$.

Будем поддерживать в СНМ на множестве вершин текущие компоненты реберной двусвязности. Заметим, что каждая компонента реберной двусвязности всегда будет являться связным поддеревом T.

Когда мы добавляем новое ребро (s, f) есть несколько случаев:

1. s и f лежат в одной компоненте реберной двусвязности. В этом случае это ребро можно пропустить, так как оно не обновит ответы.

2. Ребро (s, f) лежит в T. В этом случае компоненты реберной двусвязности не поменяются. Единственное, что может поменяться — некоторые новые вершины могут стать достижимы из вершины 1.

3. Ребро (s, f) не лежит в T, но s и f из разных компонент реберной двусвязности. В этом случае компоненты реберной двусвязности на пути между s и f объединяются в одну.

Поймем, как в случаях 2 и 3 нужно обновлять ответы. Пусть, не умаляя общности, глубина вершины s не больше глубины вершины f.

В случае 2 обойдем вершины, которые станут достижимы из вершины 1 (а до этого не были). Это можно сделать обходом в глубину из вершины f (проходя по ребрам T вес которых $\leqslant t$). Для каждой из этих вершин ответ нужно установить равным f_p+t, где f_p равно текущему минимальному ребру на пути от 1 до p (включая ребра внутри компонент реберной двусвязности).

В случае 3 объединим в СНМ компоненты реберной двусвязности на пути между s и f. Это можно сделать за $O(\alpha(n)\ell)$, где ℓ — длина этого пути. Далее для всех вершин p, таких что на пути от 1 до p лежит новая компонента, нужно обновить ответ числом w_s+t, где w_s — цена минимального ребра внутри этой компоненты.

У нас есть несколько нерешенных вопросов:

1. Как поддеживать числа w_s для всех компонент реберной двусвязности?

2. Как поддерживать числа f_p для всех вершин?

3. Как обновлять ответы быстро?

Для решения первого вопроса просто будем поддерживать эти числа для каждой компоненты в СНМ и обновлять при объединении компонент. Также для каждой компоненты будем поддерживать самую высокую вершину из этой компоненты (вершину с минимальным расстоянием до 1 в дереве T).

Для решения второго вопроса будем поддерживать дерево отрезков на массиве вершин, расположенных в порядке эйлерова обхода дерева T. Его значениями будут f_p. Тогда:

- В случае 2 нужно обновить числа в поддереве T для вершины f числом t. Поскольку вершины поддерева это отрезок эйлерова обхода, нам нужно сделать групповую операцию $min =$ на отрезке.

- В случае 3 рассмотрим самую высокую вершину v новой объединенной компоненты. Тогда нам нужно обновить числа в поддереве T для вершины v числом t (потому что если эта компонента лежит на пути от 1 до p, то вершина v тоже на нем лежит). Для этого аналогично нужно сделать групповую операцию $min =$ на отрезке.

Для решения третьего вопроса аналогично будем поддерживать другое дерево отрезков на массиве вершин, расположенных в порядке эйлерова обхода дерева T. Его значениями будут ответы для вершин. Тогда:

- В случае 2 нужно в этом ДО выставлять конкретное значение $f_p + t$ в позиции вершины p (не смотря на посчитанный ответ до этого).

- В случае 3 рассмотрим самую высокую вершину v новой объединенной компоненты. Тогда нам нужно обновить ответ в поддереве T для вершины v числом $w_s + t$. Для этого нужно сделать групповую операцию $min =$ на отрезке. Заметим, что мы можем обновить ответ для некоторых вершин, которые сейчас не достижимы из 1, но в этом случае мы эти ответы все равно забудем, когда будем выставлять правильное значение в тот момент, когда эта вершина станет достижима.

Получаем, что для решения задачи нам нужно дерево отрезков с операцией «$min = x$» (уменьшить значение до x, если оно больше x) на отрезке, выставление конкретного значения в точке, получение конкретного значения в точке.

В итоге мы получаем решение за $O(m \log n)$.