Разбор вступительной работы ЛКЛ 2017
Параллели C'–A

Если после разбора останутся вопросы, не стесняйтесь задавать их по любому из доступных контактов ЛКЛ. Обратите также внимание на исходные коды авторских решений, которые выложены отдельно.

Содержание

A. [C'] Чёткость

Автор задачи: Алексей Плешаков.
Автор разбора: Алексей Плешаков.

Задача была чисто реализационная: нужно было просто проверить каждое число на чётность/нечётность, запомнить их количества, а потом вывести это всё в правильном формате. Например, можно было написать такой проход:

```
int n, even, odd;
even = odd = 0;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
    cin >> a[i];
for (int i = 0; i < n; i++) {
    if (a[i] % 2 == 0)
        even++;
    else
        odd++;
}
cout << even << endl;
for (int i = 0; i < n; i++)
    if (a[i] % 2 == 0)
        cout << a[i] << ' ';
cout << endl << odd << endl;
for (int i = 0; i < n; i++)
    if (a[i] % 2 != 0)
        cout << a[i] << ' ';
cout << endl;
```

Код написан на C++, на остальных языках реализация отличается только синтаксисом. Решение работает за $O(n)$.

B. [C', C] Делимость на девять

Автор задачи: Алексей Плешаков.
Автор разбора: Алексей Плешаков.

Поскольку числа очень большие, невозможно записать их в какой-то стандартный числовой тип: самый большой из тех, что обычно есть в языке программирования, занимает 64 бита, то есть хранит порядка 19 десятичных цифр, а даны числа размером до миллиона цифр. Значит, просто перемножить и проверить произведение на делимость не получится.

Поймём, когда произведение делится на девять: тогда, когда хотя бы один из множителей делится на девять, либо когда оба множителя делятся на три.

Теперь нам надо проверить на делимость на три и девять непосредственно наши числа. Вспомним признаки делимости на три и девять: они зависят от суммы цифр числа. Заметим, что так как длина чисел не более 10^6, то сумма цифр одного из них может быть не более $9 \cdot 10^6$. Значит, мы можем посчитать эти суммы и проверить их на делимость трём и девяти — тогда и числа делятся на три и девять соответственно.

Чтобы посчитать сумму цифр, можно считать числа как строки и пройтись по ним посимвольно, например, так:

```
string a, b;
cin >> a >> b;
int asum, bsum;
asum = bsum = 0;
for (int i = 0; i < a.size(); i++)
    asum += int(a[i] - '0');
for (int i = 0; i < b.size(); i++)
    bsum += int(b[i] - '0');
```

После этого в `asum` и `bsum` записаны суммы цифр `a` и `b` соответственно.

Всё решение работает за $O(|a|+|b|)$, где $|a|$ — длина десятичной записи числа a.

C. [C', C] Удобные грядки

Автор задачи: Ирина Турова.
Автор разбора: Кирилл Симонов.

Главное наблюдение — достаточно проверять только соседей каждой клетки с морковкой. А именно, пусть какая-то клетка содержит морковку, тогда

1. 4 клетки по диагонали от неё не должны содержать морковок — если какая-то из них содержит, то это обязаны быть разные грядки, но тогда они будут соприкасаться по углу;

2. из 4 клеток, соседних с ней по стороне, либо ни одна не занята морковкой, либо только одна, либо только две противоположные — иначе возникает "уголок", который опять же не может быть одной грядкой, а значит будет соприкосновение разных грядок.

Как мы уже проверили, если для какой-то из клеток какое-то из условий выше не выполняется, то ответ точно "NO". На самом деле, этих условий и достаточно — если они для всех клеток с морковкой выполнены, то ответ "YES". Покажем, как можно разбить морковь на грядки в таком случае.

Выберем из клеток с морковкой самую левую, и из всех таких самую верхнюю. У неё может быть не более одного соседа с морковкой — их точно нет слева и сверху в силу выбора клетки, и не может быть одновременно снизу и справа в силу условия 2. Если такой сосед есть, перейдём в него и посмотрим на его соседей. В силу условий 1 и 2 их не более двух, но один из них — наша исходная клетка, а второй, если есть, расположен напротив неё — далее перейдём в него. Будем продолжать так идти, пока можем. Все пройденные клетки и будут грядкой. Из-за того, что новая клетка всегда была противоположна старой, мы получим прямоугольник ширины 1 либо высоты 1. При этом ни одна из оставшихся клеток с морковкой не соприкасается с этим прямоугольником, даже по углу, в силу того, что условия 1 и 2 выполнены для каждой из клеток прямоугольника.

Дальше будем повторять то же самое для оставшихся клеток — выберем из них самую левую, из них самую верхнюю, и выделим тем же способом её прямоугольник-грядку, потом снова, и так, пока клетки с морковкой не кончатся. Так как условия 1 и 2 выполнены для каждой клетки с морковкой, никакие две грядки не будут соприкасаться — в месте соприкосновения одно из условий обязательно бы нарушилось.

Таким образом, в решении не обязательно строить грядки явно, а достаточно только проверить условия 1 и 2 для каждой клетки с морковкой. Более того, условие 2 на самом деле излишне — если для всех клеток с морковкой верно условие 1, то и условие 2 автоматически верно: там, где оно неверно, возникает "уголок", а в "уголке" обязательно есть две соседние по углу клетки. На самом деле, достаточно проверять только следующее условие для каждой клетки с морковкой:

1. клетки слева-сверху и слева-снизу от неё свободны от морковок.

Ведь если для одной клетки другая оказалась соседней по диагонали не слева, а справа, то для той клетки наша является соседней как раз слева по диагонали, и условие всё равно нарушится.

Ещё один тонкий момент: проверяя клетки слева-сверху и слева-снизу от данной, нужно не вылезти за границы массива, в котором хранится поле. Нужно либо при обращении к соответствующему элементу проверять, что его координаты корректны, либо с самого начала добавить "каёмку" — расширить поле на 1 в каждую сторону на клетки, свободные от морковок.

Описанное решение работает за $O(NM)$.

D. [C, B'] Ленивая подготовка к ЛКЛ

Автор задачи: Ирина Турова.
Автор разбора: Кирилл Симонов.

Для строк a и b введём обозначение: $a \prec b$ тогда и только тогда, когда поручение a предпочтительнее поручения b в описанном в условии смысле. Можно заметить, что это будет корректное отношение линейного порядка. То есть, для данного набора поручений существует упорядочивание такое, что для любых a и b, если a идёт раньше b, то $a \prec b$. Это упорядочивание можно получить, просто запустив любой алгоритм сортировки, который будет сравнивать два поручения ровно так, как описано в условии. Заметим, что так как по условию наборы заданий во всех поручениях различны, то для любых двух a, b либо $a \prec b$, либо $b \prec a$, и итоговый порядок определён однозначно.

Формально, нужно написать функцию `compare`, которая принимает два аргумента a, и b, и возвращает `true` тогда и только тогда, когда $a \prec b$ — а это проверяется просто моделированием условия: возьмём в каждом из поручений наименьшую букву и сравним, если они оказались равны, то возьмём следующую, и так далее пока не нашлось различие, либо одно из слов не кончилось. Теперь, если вы пишете свой алгоритм сортировки, то там, где вы бы в обычном случае сравнивали числа, вы вместо этого будете вызывать функцию `compare` для соответствующих двух поручений. Если же вы пользуетесь стандартным алгоритмом сортировки, вы просто передадите ему функцию `compare` в качестве компаратора.

Пусть L — максимальная длина строки, тогда сравнение двух строк производится за $O(L^2)$. Всего сравнений будет $O(N^2)$ для квадратичной сортировки и $O(N \log N)$ для быстрой. Общее время работы будет соотвественно $O(L^2N^2)$ или $O(L^2N \log N)$.

Альтернативное решение.

Определение. Строка a лексикографически меньше строки b, если в первой слева позиции, в которой они отличаются, соответствующий символ a меньше соответствующего символа b, либо если строка a короче b и является её началом.

Можно заметить, что на самом деле $a \prec b$ тогда и только тогда, когда отсортированная по буквам строка a лексикографически меньше отсортированной строки b — действительно, в обоих случаях сначала сравнивается наименьшая буква, затем следующая, и так далее. Из этого сразу понятно, что определённый таким образом порядок “хороший”, но можно получить и более простой алгоритм — вместо того, чтобы явно проделывать процедуру из условия в каждом сравнении, можно сразу отсортировать данные строки по буквам, а дальше сортировать полученные строки уже стандартной операцией $<$, которая как раз и соответствует лексикографическому сравнению строк.

В этом решении нужно позаботиться ещё о восстановлении ответа — ведь после сортировки по буквам строки уже не совпадают с исходными. Для этого можно либо сортировать пары из отсортированной строки и неотсортированной, либо в конце для каждой из исходных строк определить какой из отсортированных она соответствует — это можно однозначно сделать, так как наборы букв во всех строках различны.

Сортировка всех строк по буквам будет производиться за $O(NL^2)$ или $O(NL \log N)$ в зависимости от выбора алгоритма сортировки, а сортировка отсортированных строк — за $O(LN^2)$ или $O(LN \log N)$ Итоговое время работы будет $O(LN(L+N))$ для квадратичной сортировки и $O(LN(\log L + \log N))$ для быстрой.

E. [C, B’] Сладенькая игра

Автор задачи: Кирилл Симонов.
Автор разбора: Алексей Плешаков.

Давайте представим, что какой-то из игроков может забрать все сладенькие конфеты из кучки противкника. Так как каждый из игроков за ход может забрать только количество конфет, кратное k, то и общее количество конфет в кучке противника должно делиться на k. Но тогда этот игрок мог взять все конфеты из кучки противника за один ход.

Значит, если A делится на k, то первый игрок либо выигрывает, либо игра оканчивается вничью — в том случае, когда второй игрок может закончить игру за один ход так же, как и первый. То есть, ничья случается тогда, когда k является делителем и A, и B, а если k делит только A, но не B, то выигрывает первый игрок. Соответственно, второй игрок выигрывает тогда, когда B делится на k, но A не делится.

Значит, можно просто перебрать все делители каждого из чисел A, B и проверить, какие из них являются общими.

Асимптотика решения — $O(\sqrt{A} + \sqrt{B})$.

F. [B', B] Выбор подарков

Автор задачи: Алексей Плешаков.
Автор разбора: Алексей Плешаков.

Поскольку задача дана во вступительной, значит, она решается каким-нибудь общеизвестным приёмом. Заметим, что если мы зафиксировали одну вершину из пары, то мы точно знаем, чем обновлять ответ: максимальным значением вершины из тех, которые не лежат в поддереве зафиксированной вершины. В общем, это может натолкнуть вас на мысль написать DFS :)

Пусть мы находимся в вершине `v`. Тогда, чтобы обновлять ответ, будем передавать в DFS величину `out`, которая будет равна максимальному значению стоимости вершины не из поддерева. Когда мы запускаемся из корня, `out = -INF` (очень большое по модулю отрицательное число, чтобы `(вес вершины 1) + out` не обновило ответ сразу).

Когда мы переходим из вершины в сына, необходимо обновить `out` теми вершинами, которые были в поддереве родителя, но не находятся в поддереве сына. Это все вершины в поддеревьях других сыновей. Поскольку нужно только максимальное значение, достаточно знать максимум в поддереве каждого из сыновей, и среди всех этих значений только два наибольших — для каждой вершины нужен лишь максимальный из сыновей, если это не она сама, а в противном случае — максимальный из остальных.

Посчитаем `mx[v]` — максимальный вес в поддереве вершины `v`. Это можно сделать простеньким DFS'ом:

```
calc(v) {
    mx[v]=(вес вершины v);
    for (по сыновьям w вершины v) {
        calc(w); //запускаем calc от сына v
        mx[v] = max(mx[v], mx[w]);
    }
}
```

DFS, решающий задачу, можно написать следующим образом:

```
dfs(v, out) {
    ans = max(ans, mx[v] + out);
    pair <int, int> maxes;
```

```
    for (по сыновьям w вершины v) {
        запомним в пару maxes две вершины с наибольшими значениями mx[w]
    }
    for (по сыновьям w вершины v) {
        if (maxes.first != w)
            dfs(w, max(out, mx[maxes.first]));
        else
            dfs(w, max(out, mx[maxes.second]));
    }
}
```

Альтернативное решение.

Ни одна из вершин пары, идущей в ответ, не лежит в поддереве другой. Значит, корень дерева не может входить в ответ. Значит, корень дерева является предком для обеих вершин ответа. Значит, для любых двух вершин, которые могут входить в ответ, существует вершина такая, что обе эти вершины лежат в её поддереве. Тогда можно перебрать все вершины и обновить ответ максимальными двумя значениями mx[] её сыновей (максимумами поддеревьев).

Время работы обоих решений — $O(n)$.

G. [B', B] Это не шутки

Автор задачи: Кирилл Симонов.
Автор разбора: Алексей Плешаков.

Сразу оценить количество каких-то длинных слов с большим количеством ограничений (запрещённые подстроки, право на "ошибку") сложно. Но, если мы уже знаем количество таких слов длины `k`, то посчитать количество слов длины `k + 1` проще — достаточно попробовать дописать каждую из букв и посмотреть, когда ограничения выполняются.

Заметим, что для того, чтобы узнать, допишем ли мы запрещённую подстроку при дописывании буквы, достаточно знать последнюю букву текущего слова. А чтобы узнать, можно ли совершить ошибку, нужно знать, была она уже в этом слове или нет.

Воспользуемся методом динамического программирования. Пусть `dp[len][last][was]` — количество слов длины `len` с последней буквой `last`. Если `was = 0`, то в таких словах ещё не было запрещённой подстроки, если `was = 1`, то была.

Начальными значениями динамики будут `dp[1][c][0] = 1`, где `c` — любая из допустимых букв, ведь каждая однобуквенная строка не содержит никакой запрещённой.

Переход от длины `len` к длине `len + 1`: будем пробовать дописать символ `c` в конце строки. Тогда, если строка из символов `last` и `c` — запрещённая и `was = 0`, то к значению `dp[len + 1][c][was + 1]` добавим `dp[len][last][was]`, иначе к `dp[len + 1][c][was]` добавим `dp[len][last][was]`.

Для того, чтобы быстро определять, является ли строка из двух символов запрещённой, можно было представить набор запрещённых строк в виде двумерного массива, каждое измерение которого индексируется буквой алфавита. В ячейке с индексами `a` и `b` будет лежать 1, если строка `ab` запрещённая, и 0 иначе.

Тогда ответом на задачу является сумма `dp[n][c][was]` по всем допустимым буквам `c` и `was = 0, 1`.

Асимптотика решения — $O(nk^2)$.

Альтернативное решение.

Можно было посчитать только количество слов, вообще не содержащих запрещённых подстрок, для каждой длины от 1 до n и для каждой последней буквы. Сделать это можно такой же динамикой, только без последнего параметра.

Количество ответов без ошибок — просто сумма `dp[n][c]` по всем буквам `c`. Чтобы посчитать число ответов ровно с одной ошибкой, можно перебрать саму запрещённую строку, позицию, где она встретилась, и тогда часть слова перед ней не содержит ошибок и кончается на определённую букву, и то же самое с частью после неё. Формально, если строка `ab` запрещена и её первый символ стоит на позиции `i`, то способов выбрать начало будет `dp[i][a]`, а способов выбрать конец — `dp[n - i][b]`. Заметим, что каждый ответ будет посчитан только один раз, поскольку иначе это была бы строка, в которой запрещённая подстрока встречается два раза в разных местах — но мы считали только те, в которой запрещённая встречается ровно один раз.

Асимптотика этого решения также $O(nk^2)$.

H. [B, A’] День дружбы

Автор задачи: Кирилл Симонов.
Автор разбора: Кирилл Симонов.

Формально условие звучит так: даны N отрезков на прямой, нужно расположить минимальное число точек так, чтобы каждый отрезок оказался покрыт — то есть, содержал хотя бы одну из выбранных точек.

Часто подобные задачи решаются жадно, так происходит и здесь.

Будем считать, что отрезки упорядочены по правому краю: $r_1 \le r_2 \le \dots \le r_N$ — с самого начала отсортируем их так. Построим наш ответ — набор точек $x_1, x_2, \dots, x_k$ такой, что любой из отрезков покрыт какой-то из точек, будем считать, что $x_1 \le x_2 \le \dots \le x_k$.

Какой может быть x_1? Ясно, что самая левая из выбранных точек не может лежать правее r_1, иначе первый отрезок точно непокрыт. С другой стороны, этой точке “незачем” (ниже строго докажем, почему так можно делать) лежать левее r_1, ведь никаких отрезков раньше r_1 не кончается, зато к моменту r_1 могли начаться ещё какие-то отрезки. Поэтому в качестве x_1 выберем просто r_1.

Вместе с отрезком номер 1 точкой x_1 возможно оказались покрыты и какие-то другие отрезки. Забудем про всех них, и повторим то же самое с оставшимися — возьмём из них отрезок, кончающийся раньше всех, и за x_2 примем его правый конец. Снова забудем про все покрытые отрезки, и будем делать так, пока ещё остаются непокрытые. Другими словами, всякий раз будем выбирать самую правую точку из тех, что ещё могут покрыть очередной отрезок — в этом и состоит жадность.

Нужно понять, как реализовать это быстро — ведь если наивно искать и удалять все отрезки, содержащие очередную точку, время работы получится квадратичным. На самом деле, явно ничего удалять не нужно — достаточно идти по отрезкам в порядке увеличения правого конца, и поддерживать последнюю взятую в ответ точку `last`. Если очередной отрезок начинается раньше `last` — то он точно покрыт этой точкой, ведь `last` совпадает с правым концом какого-то из предудущих отрезков, а значит правый конец нашего отрезка находится не левее точки `last` в силу порядка на отрезках. В этом случае отрезок уже покрыт, и точно можно его пропустить.

Если же левый конец очередного отрезка находится правее `last` — то этот отрезок не покрыт точкой `last` и, более того, не покрыт и никакой другой точкой из тех, что мы уже взяли в ответ — ведь `last` самая правая из них. Значит, этот отрезок ещё не покрыт, и очередной точкой ответа будет его правый конец (`last` тоже нужно обновить этим значением).

Время работы такого решения — $O(N \log N)$.

Поймём теперь, почему такое решение действительно оптимально. Пусть $x_1 \le x_2 \le \cdots \le x_k$ — полученный нашим решением ответ. По построению, каждая из этих точек — правый конец некоторого отрезка, то есть для каждого i от 1 до k существует s_i от 1 до N, что $x_i = r_{s_i}$, при этом $s_1 < s_2 < \cdots < s_k$, если считать отрезки упорядоченными по правому концу.

Оказывается, что наше решение в некотором смысле максимально правое. А именно, пусть $y_1 \le y_2 \le \cdots \le y_m$ — любое другое решение, то есть набор точек, покрывающих все отрезки. Тогда обязательно $y_1 \le x_1$, $y_2 \le x_2, \ldots, y_{\min(m,k)} \le x_{\min(m,k)}$. Предположим, что это не так, и рассмотрим первое такое l, что $x_l < y_l$, и $x_i \ge y_i$ для всех предыдущих номеров i. Отрезок s_l, соответствующий x_l, не будет покрыт точками из y — ведь раз мы взяли его правый конец в ответ в качестве x_l, то он не был покрыт предыдущими x_i, а значит и не был покрыт предыдущими y_i, поскольку они ещё левее. Но он не покрыт и y_l — по предположению эта точка строго правее x_l — правого конца этого отрезка. И оставшимися точками из y он не покрыт — они лежат ещё правее y_l. Таким образом, один из отрезков не покрыт, а значит y — не решение, противоречие.

Рассмотрим теперь оптимальное решение $y_1 \le y_2 \le \cdots \le y_m$, пусть в нём меньше точек, чем в нашем, то есть $m < k$. По только что доказанному, $y_i \le x_i$ для любого i от 1 до m. Но тогда отрезок s_{m+1} не покрыт точками из y — раз точка x_{m+1} вошла в ответ, предыдущие точки из x были строго левее этого отрезка, а значит и точки из y его не покрывают, так как находятся ещё левее точек из x. То есть, y — не решение, и мы получили противоречие. Значит, не может быть решения, точек в котором меньше чем в нашем, и наше решение оптимально.

I. [B, A', A] Умножение и деление

Автор задачи: Кирилл Симонов.
Автор разбора: Кирилл Симонов.

Посмотрим на выписывание произведения как на процесс: мы начинаем с пустого произведения, и всякий раз либо умножаем, либо делим на какое-то число, и при этом после каждой операции значение произведения остаётся в отрезке целых чисел от 1 до N. И за каждое действия умножения или деления мы платим какую-то стоимость, которые складываются по всем переходам.

Описанное хорошо формализуется в терминах теории графов: пусть наше состояние (вершина) — текущее значение произведения, переход (ребро) — умножение на очередной множитель. Формально, вершины — целые числа от 1 до N, и если b делится на a, то из a в b ведёт ребро стоимости $x_{b/a}$, и из b в a — ребро стоимости $y_{b/a}$. Тогда, действительно, любое корректное по условию произведение со значением a — это некоторый путь в таком графе от вершины 1 до вершины a, и каждый путь из 1 даёт корректное произведение. Стоимость произведения равна обычной стоимости пути — сумме стоимостей рёбер.

Таким образом, мы свели задачу к нахождению кратчайшего расстояния от вершины 1 до всех остальных в некотором графе. Вершин в графе N, а рёбер, на самом деле, $O(N \log N)$ — их не больше чем $2 \cdot (N + N/2 + N/3 + \dots)$, ведь умножать на k можно только числа, не превосходящие N/k.

Для решения задачи поиска кратчайших расстояний существуют стандартные алгоритмы — наиболее известны алгоритм Дейкстры и алгоритм Форда–Беллмана. В случае алгоритма Дейкстры достаточно обычной реализации с выбором ближайшей вершины циклом, тогда общее время работы будет $O(N^2)$, что прекрасно укладывается в ограничение по времени. Алгоритм Форда–Беллмана в обычной реализации двумя циклами, даже с учётом хорошей оценки на число рёбер, будет работать порядка $N^2 \log N$ времени, что несколько долго. Однако, поскольку граф не произвольный, кратчайшие пути в нём обычно состоят из небольшого числа рёбер, и варианты алгоритма Форда–Беллмана с очередью или просто отсечением по отсутствию релаксаций на очередной итерации работают быстро.

J. [A', A] Скобочки

Автор задачи: Алексей Гордеев.
Автор разбора: Алексей Плешаков.

Давайте поймём, какие именно скобки нужно дописать к какой-то скобочной последовательности, чтобы она стала правильной. Посчитаем баланс на этой последовательности; посмотрим на минимальное значение этого баланса при подсчёте (пусть оно равно p) и на баланс в конце строки (пусть он равен q).

Нетрудно проверить, что скобочная последовательность является правильной тогда и только тогда, когда её баланс всюду неотрицателен, а в конце равен 0. Тогда становится ясно, что нужно добавить хотя бы $-p$ открывающих скобок в последовательность, чтобы минимальное значение баланса стало неотрицательным. Баланс в конце станет равен $q - p$, и нужно будет добавить ещё хотя бы $q - p$ закрывающих скобок, чтобы баланс в конце стал нулём. Всего нужно хотя бы $q - 2p$ скобок. Но это количество окажется и достаточным — действительно, если в начало поставить $-p$ открывающих скобок, минимум баланса станет равным нулю, и с ещё $q - p$ закрывающими скобками в конце суммарный баланс тоже станет равен нулю.

Мы умеем считать величины p и q для последовательности за её длину. Заметим, что если мы знаем значения p и q для двух скобочных последовательностей, то мы знаем эти значения и для их конкатенации. В самом деле, пусть минимум баланса первой последовательности — p_1, а баланс в конце — q_1, и аналогично для второй p_2 и q_2. Тогда минимум баланса конкатенации будет $\min(p_1, q_1 + p_2)$ — он либо достигся на первой части, для которой ничего не изменилось, либо на второй части, в которой значение баланса в каждой точке увеличилось на суммарный баланс первой последовательности. А суммарный баланс конкатенации будет просто суммой балансов — $q_1 + q_2$.

Построим на данной последовательности скобок дерево отрезков, вершина которого будет соответсвовать скобочной последовательности на соответствующем отрезке. В вершине ДО будем поддерживать значения p и q для этой последовательности. В листьях дерева расположены последовательности длины 1, для которых эти значения можно посчитать явно — это будет $(0, 1)$ для открывающей скобки, и $(-1, -1)$ для закрывающей. Значения в остальных вершинах будем вычислять снизу вверх, вычисляя p и q в очередной вершине по правилу выше, пользуясь тем, что она — конкатенация двух своих сыновей.

Тогда ответ на запрос (l, r) происходит за $O(\log n)$: просто находим вершины ДО, покрывающие данный запрос, обычным спуском, а затем по правилу выше вычисляем p и q для этой подстроки — ведь она является конкатенацией найденных вершин.

Дерево строится за $O(n \log n)$, каждый запрос обрабатывается за $O(\log n)$, итоговая асимптотика — $O((n+m)\log n)$.

Альтернативное решение.

Можно использовать ту же идею с балансом немного по-другому. Нам нужно для последовательности скобок с l-й по r-ю найти две величины: минимальный баланс и суммарный баланс. Пусть мы вычислили баланс в каждом месте исходной строки. Тогда вычисленные значения, относящиеся к отрезку $[l, r]$, почти равны соответствующим значениям баланса для подстроки $[l, r]$, кроме того, что они все оказались увеличены на суммарный баланс строки $[1, l-1]$.

Таким образом, можно найти суммарный баланс строки $[1, r]$ и минимальный из балансов исходной строки на отрезке $[l, r]$, вычесть из каждого значения суммарный баланс строки $[1, l-1]$, и это и будет соответственно суммарный и минимальный баланс подстроки $[l, r]$, взятой отдельно.

Для нахождения минимума баланса на отрезке нужно воспользоваться стандартной структурой над массивом вычисленных балансов исходной строки — например, деревом отрезков. С деревом отрезков сложность решения будет $O((n+m)\log n)$.

K. [A', A] Плагиат

Автор задачи: Алексей Гордеев.
Автор разбора: Алексей Гордеев.

В задаче даны две строки s и t, требуется посчитать сумму количеств вхождений всех подстрок строки s в строку t.

Рассмотрим строку, состоящую из строк s и t, разделённых символом, не встречающимся ни в одной из них — $s\#t$, и посчитаем z-функцию для этой строки. $z[i]$ равняется максимальной длине подстроки, начинающейся с i-го символа и совпадающей с началом строки. Это то же самое, что и количество подстрок, начинающихся с i-го символа и совпадающих с каким-то префиксом строки. Тогда сумма значений $z[i]$ по индексам, соответствующим символам строки t, равняется суммарному количеству подстрок t, совпадающих с каким-то префиксом строки s (благодаря разделительному символу $\#$ ни одно значение $z[i]$ не может быть больше длины строки s).

Итак, мы посчитали сумму количеств вхождений всех префиксов строки s в строку t. Нам же нужно посчитать такую величину для всех подстрок s, а не только префиксов. Но любая подстрока s является префиксом какого-то суффикса s, поэтому если мы выполним те же действия не только для строки s, но и для всех её суффиксов, и просуммируем результат, то мы получим ответ на задачу.

Пусть строка s имеет длину m, строка t имеет длину n. У строки s есть m суффиксов, подсчёт z-функции занимает линейное время от длины строки, длина строки, от которой мы считаем z-функцию, не превосходит $m + n + 1$. Получаем решение за $O(m(m + n + 1)) = O(m(m + n))$.

Полезное упражнение — решить задачу тем же способом, но пользуясь префикс-функцией вместо z-функции.

L. [A] Уровень допуска

Автор задачи: Алексей Гордеев.
Автор разбора: Алексей Гордеев.

Подвесим дерево за какую-нибудь вершину, после чего для каждой вершины и каждого подмножества цветов посчитаем количество путей из этой вершины в её поддерево, в которых встречаются все цвета подмножества и только они.

Будем считать эти величины динамикой по дереву. Пусть $dp[v][mask]$ — искомое количество путей для вершины v и подмножества цветов $mask$ (будем сопоставлять подмножеству $\{a_1, \ldots, a_n\}$ число $mask = \sum_{i=1}^{n} 2^{a_i}$).

Для листа u $dp[u][0] = 1$, остальные состояния динамики равны нулю.

Для произвольной вершины v заметим, что любой путь из неё в поддерево — это либо путь длины ноль (с пустой маской), либо путь, проходящий через какого-то её сына u. Таким образом, нужно инициализировать значения динамики для вершины v так же, как и для листа, после чего перебрать сына u и подмаску $mask$, и добавить $dp[u][mask]$ к $dp[v][mask \mid 2^c]$, где $\mid$ — это битовое или, а c — это цвет ребра между u и v.

Осталось посчитать ответ, для этого заметим, что любой путь либо идёт только вверх по дереву, либо сначала вверх, а потом вниз. Для того, чтобы подсчитать ответ для путей первого типа, нужно просто просуммировать посчитанные состояния динамики, умножая их на количество цветов в соответствующем подмножестве.

Для того, чтобы подсчитать ответ для путей второго типа, нужно перебрать вершину v — самую близкую к корню вершину на пути, вершину u — её сына, из которого мы пришли в неё, маску куска пути, начиная с вершины v, и маску куска пути до вершины v. После чего нужно перемножить количество путей из u в её поддерево с выбранной маской и количество путей из v в её поддерево с выбранной маской, не проходящих через u. Кроме того, нужно не забыть учесть цвет ребра между u и v.

Если дерево состоит из n вершин, а рёбра бывают k различных цветов, получаем решение за $O(2^k 2^k n) = O(4^k n)$.

Альтернативное решение.

Приведём ещё одно решение, которое немного сложнее придумать, зато проще писать. Кроме того, это решение имеет лучшую асимптотику.

Для каждой маски посчитаем $cnt[mask]$ — количество путей в дереве, рёбра которых имеют цвета только из этой маски (но не обязательно все эти цвета). Посчитать это количество очень просто — нужно запустить dfs, который может ходить только по рёбрам разрешённых цветов. Если он найдёт компоненты связности размера $m_1, \ldots, m_l$, то

$$cnt[mask] = \sum_{i=1}^{l} \frac{m_i(m_i - 1)}{2}.$$

Теперь можно воспользоваться формулой включений-исключений, чтобы посчитать $cnt2[mask]$ — количество путей в дереве, рёбра которых имеют цвета только из этой маски, причём каждый цвет встречается хотя бы один раз.

Ответ на задачу — это просто сумма $cnt2[mask]$, умноженных на количество цветов в маске.

Полученное решение имеет асимптотику $O(2^k n + 4^k)$, или $O(2^k n + 3^k)$, в зависимости от того, насколько аккуратно вычисляется $cnt2[mask]$.