Разбор вступительной работы ЛКЛ 2018
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‘] Самое простое **1**

B. [C‘, C] Лестницы **2**

C. [C‘, C] Саша и воздушные шарики **4**

D. [C, B‘] Саша и диагонали **5**

E. [C, B‘] Простой цикл **6**

F. [B‘, B] Вова и интернет **7**

G. [B‘, B] Шахматное поле **9**

H. [B, A‘] Все смены одинаковые **10**

I. [B, A‘, A] Калифорнийский стиль **11**

J. [A‘, A] Пермское метро **13**

K. [A‘, A] Пермское метро-2 **14**

L. [A] Назови стартап **16**

A. [C‘] Самое простое

Автор задачи: Кирилл Симонов.
Автор разбора: Владимир Линд.

Можно было действовать прямолинейно: помнить, какие параллели уже были — для этого можно использовать структуру данных множество, булевый массив с индексами от A до Z, обычный список или просто максимальную параллель из встречавшихся (с учетом того, что задачи даны по возрастанию первой буквы). Когда приходит очередная задача, нужно посмотреть, была ли уже эта параллель (первая буква), и если нет, то вывести название задачи и запомнить, что такая параллель уже была.

Альтернативное решение.

Условием гарантировано, что задачи сгруппированы по параллелям. Тогда, первая задача в каждой группе либо вообще самая первая в списке, либо идет после задачи другой параллели. А все задачи, которые не являются первыми для своей параллели, идут после задачи из той же параллели. Таким образом, достаточно вывести самое первое название, а затем сравнить первые символы каждой пары соседних строк и, если они разные, вывести вторую строку. Одна из возможных реализаций:

```
int n;
cin >> n;
vector <string> tasks(n);
for (int i = 0; i < n; i++) {
    cin >> tasks[i];
}
cout << tasks[0] << endl;
for (int i = 1; i < n; i++) {
    char firstLetter1 = tasks[i - 1][0];
    char firstLetter2 = tasks[i][0];
    if (firstLetter1 != firstLetter2) {
        cout << tasks[i] << endl;
    }
}
```

Код написан на C++, на других языках реализация подобная.

B. [C‘, C] Лестницы

Автор задачи: Ярослав Свиридов.
Автор разбора: Ярослав Свиридов.

Нужно было пройти по матрице и посчитать сумму модулей разностей значений в соседних клетках, а также найти количество пар соседних клеток с различными значениями. Реализовать подсчет можно было следующим образом:

```
int sum = 0, cnt = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        //рассматриваем клетку ниже нашей
        if (i + 1 < n && a[i][j] != a[i + 1][j]) {
            sum += abs(a[i][j] - a[i + 1][j]);
            cnt++;
        }
        //теперь правее нашей
        if (j + 1 < m && a[i][j] != a[i][j + 1]) {
            sum += abs(a[i][j] - a[i][j + 1]);
            cnt++;
        }
    }
}
cout << sum << " " << cnt;
```

Одним проходом по матрице можно посчитать сразу оба значения.

Заметим, что не нужно рассматривать клетку сверху и клетку слева — для них наша клетка это клетка снизу и клетка справа соответственно, то есть лестницы в эти клетки мы уже посчитали, когда обрабатывали их.

C. [C‘, C] Саша и воздушные шарики

Автор задачи: Александр Чухарев.
Автор разбора: Кирилл Симонов.

Пусть есть некоторое число Y. Двоичная запись числа $2Y$ — это двоичная запись числа Y, сдвинутая на 1 (например, $5 = 101_2$, а $10 = 1010_2$). Старший бит числа $2Y$ будет на один больше старшего бита Y, а значит старший бит числа $2Y \oplus Y$ тоже будет на 1 больше старшего бита Y. Таким образом, если $Y \geq 2^{15}$, $2Y \oplus Y \geq 2^{16} = 65536$, а значит не подойдет ни для какого допустимого по условию X.

То есть достаточно рассматривать возможные Y только от 1 до $2^{15} - 1$. Перед чтением запросов, мы можем предподсчитать для каждого X оптимальный Y следующим способом.

Заведем массив с индексами от 1 до 65536, они будут соответствовать X, а значением в ячейке X будет оптимальное значение Y. Исходно заполним массив -1. Теперь переберем по возрастанию все возможные значения Y (от 1 до $2^{15} - 1$), и для каждого вычислим $X = 2Y \oplus Y$. Если в ячейке X записано -1, запишем туда Y, иначе не будем делать ничего — в этом случае уже найден меньший Y, который подходит. Теперь для каждого X мы умеем быстро находить ответ — нужно просто посмотреть в нужную ячейку массива.

Более сложная версия задачи.

На самом деле, можно решить и более общую задачу: находить подходящий Y для одного отдельно взятого X за длину его битовой записи k. Представим, что значения битов Y это неизвестные, и тогда у нас есть k равенств вида $x_i = y_i \oplus y_{i-1}$. По этим равенствам мы можем однозначно восстановить Y: $y_{k-1} = x_k$, $y_{k-2} = x_{k-1} \oplus y_{k-1} = x_{k-1} \oplus x_k$, и так далее. В конце может оказаться что ответа нет, если $x_0 \neq y_0$.

D. [C, B‘] Саша и диагонали

Автор задачи: Александр Чухарев.
Автор разбора: Кирилл Симонов.

Заметим, что итоговый порядок определяется однозначно. Даже если у двух диагоналей совпали длина и сумма, максимальные элементы в них отличаются — у разных диагоналей нет общих ячеек, а все элементы таблицы по условию различны.

Первая часть решения — для каждой диагонали вычислить длину, сумму и максимальный элемент. Нужно либо научиться обходить все клетки одной диагонали (это несложно сделать, если понять, что на диагонали с номером k лежат в точности клетки, сумма координат которых равна $k+1$) и вычислить эти значения для каждой диагонали последовательно, либо обойти все клетки в таблице, и для каждой клетки обновить значения соответствующей ей диагонали.

После этого нужно отсортировать по возрастанию упорядоченные наборы чисел (длина, сумма, максимальный элемент, номер диагонали), а после вывести номера диагоналей в этом порядке. Квадратичные сортировки тоже проходили по времени. Чтобы не писать сравнение наборов чисел вручную, в некоторых языках можно было воспользоваться встроенными структурами, например `vector` или `tuple` в C++, `tuple` в Python, которые по умолчанию именно в таком порядке и сортируются.

E. [C, B‘] Простой цикл

Автор задачи: Антон Буков.
Автор разбора: Антон Буков.

Сперва заметим, что граф, который нам дают, необычный. Это полный ориентированный граф, называющийся турниром. Чтобы решить задачу, докажем следующее утверждение.

Лемма. *Если в графе-турнире есть цикл, проходящий через вершину v, то есть и цикл длины 3, проходящий через эту вершину.*

Доказательство. Пусть есть цикл, проходящий через эту вершину $v \to u_1 \to u_2 \to \ldots \to u_k \to v$. Посмотрим на ребра между вершиной v и u_i, где $1 \le i \le k-1$. Если есть ребро $v \to u_i$ и $u_{i+1} \to v$, то мы нашли нужный цикл. Предположим, что такого не нашлось, тогда, так как есть ребра $v \to u_1$ и $u_1 \to u_2$, то ребро между v и u_2 обязано быть $v \to u_2$. Тогда аналогичными соображениями получаем, что между v и u_3 ребро $v \to u_3$. Продолжая так, получаем, что между v и u_k ребро $v \to u_k$. Но это был цикл, и в нем было ребро $u_k \to v$, а в графе-турнире есть ровно одно ребро между двумя вершинами. Пришли к противоречию. □

Тогда, так как в турнире не может быть циклов длины 1 (нет петель) и 2 (тогда есть два ребра между одной и той же парой вершин), кратчайшей простой длиной цикла всегда будет 3, если в графе вообще есть хоть какой-то цикл — иначе ответа не существует. Посчитаем количество циклов длины 3. Для этого просто переберем две другие вершины цикла a и b (в дополнение к стартовой вершине, которая известна) и проверим, что эти три вершины образуют цикл, то есть что есть ребра $v \to a$, $a \to b$, $b \to v$.

F. [B‘, B] Вова и интернет

Автор задачи: Владимир Линд.
Автор разбора: Владимир Линд.

Есть очередь из m людей. Нужно добавить в эту очередь k друзей, чтобы штраф был минимальным.

Рассмотрим друга, стоящего i-м по порядку. Перед ним стоит $i-1$ друзей, значит он должен стоять не ближе, чем на i-м месте с начала в общей очереди. За ним стоит $k-i$ друзей, значит он должен стоять не дальше, чем на $(k-i+1)$-м месте с конца в общей очереди, то есть не дальше, чем на $(m+i)$-м месте с начала. Следовательно, у каждого друга есть не более, чем $m+1$ вариантов мест. Для каждого из таких мест посчитаем минимальный штраф, которого можно добиться, каким-либо образом расставив впереди $i-1$ друзей.

В этом нам поможет метод динамического программирования. Пусть `dp[i][place]` — описанная выше величина, где `i` — номер друга, изменяется от 1 до k; `place` — одно из вариантов мест, изменяется от 1 до $m+1$, но фактически означает место в очереди с номером `i-1+place`. К примеру, для первого друга `place = 3` будет означать третье место в очереди, а для четвертого друга `place = 3` будет означать шестое место в очереди. Для удобства величину `place` будем называть *возможное место*, а соответствующую величину `(i-1+place)` — *место в очереди*.

Начальными значениями динамики будут `dp[1][place] = a[place]` — штраф за место `place`, занимаемое первым другом в очереди. Так как перед ним нет друзей, он не платит штраф за людей, стоящих после ближайшего друга спереди.

Переход от друга `i` к другу `i+1`: посчитаем `dp[i+1][place]`. Он должен заплатить штраф за *место в очереди*, которое он занял — `a[i+place]`. Еще он должен заплатить штраф за людей, стоящих между ним и другом спереди. Если этот друг занимает *возможное место* `p`, то людей между ними будет `place-p`, за каждого из которых нужно будет заплатить штраф d. Также не забываем, что штраф нужно платить не только `(i+1)`-му другу, но и всем друзьям спереди — это уже посчитанная величина `dp[i][p]`. При этом из всех *возможных мест* `p` нужно выбрать такое, чтобы общий штраф был минимальным. Итого `dp[i+1][place] = a[i+place] + min(dp[i][p] + (place-p)*d)` для `p` $\in$ `[1; place]`.

Научимся быстро считать такой минимум. Обозначим для рассмат-

риваемого (i+1)-го друга вспомогательный массив mn, где mn[place] = min(dp[i][p] + (place-p)*d), p ∈ [1; place]. То есть теперь dp[i+1][place] = a[i+place] + mn[place]. Будем поочередно считать значения mn для place от 1 до m+1.

Для place = 1 единственное значение mn = dp[i][1], так как при *возможном месте* place = 1 единственное *возможное место* p будет 1. Посчитаем для place = j. Если минимумом будет значение не при p = j, то нужно выбрать минимальное из значений при p ∈ [1; j-1]. Сравним это с уже посчитанной величиной mn[j-1]:

mn[j] = min(dp[i][p] + (j-p)*d), p ∈ [1; j-1]
mn[j-1] = min(dp[i][p] + (j-1-p)*d), p ∈ [1; j-1]

Значение mn[j] отличается только величиной d, которую можно вынести за функцию минимума. То есть, если минимум достигается не при p = j, то mn[j] = d + mn[j-1]. Если же минимум достигается при p = j, то mn[j] = dp[i][j]. Итого mn[j] = min(dp[i][j], d + mn[j-1]).

Финальное решение: проходимся поочередно по k друзьям, для каждого, кроме первого, заполняем вспомогательный массив mn, далее заполняем слой динамики dp[i], используя посчитанный mn. Ответом будет минимальное из значений на слое динамики dp[k] — минимальный штраф, которого можно добиться, расставив k друзей. Асимптотика решения — $\mathcal{O}(km)$.

Чтобы восстановить ответ динамики — узнать конкретные места, на которых должны стоять друзья, чтобы заплатить минимальный штраф, для каждого состояния динамики при пересчете будем запоминать, на каком из *возможных мест* стоит ближайший друг спереди. Полностью посчитав динамику, пройдемся по запомненным местам, начиная с ячейки с минимальным штрафом на последнем слое, и запишем соответствующие *места в очереди* в отдельный массив.

G. [B‘, B] Шахматное поле

Автор задачи: Ярослав Свиридов.
Автор разбора: Владимир Линд.

Рассмотрим произвольную пару соседних по стороне клеток (далее — просто соседние). По определению шахматного множества, если эти клетки одинакового цвета, они должны находиться в разных шахматных множествах. Что если клетки разного цвета?

Утверждение. *Соседние клетки разного цвета должны находиться в одном шахматном множестве, чтобы кол-во множеств в разбиении было минимально.*

Доказательство.

Пусть не так: поле клеток разбили на минимальное кол-во шахматных множеств, и есть пара соседних клеток $\{a, b\}$ разного цвета, которые находятся в разных шахматных множествах. Объединим их и убедимся, что полученное множество является шахматным. Очевидно, что оно связно. Поочередно рассмотрим каждую пару соседних клеток $\{c, d\}$ в новом множестве такую, что клетка c была в том же множестве, что и клетка a, а клетка d в том же, что и b. Докажем, что клетки c и d разного цвета.

Определим расстояние между клетками как сумму модуля разницы по координате x и модуля разницы по координате y: $\rho(m, n) = |x_m - x_n| + |y_m - y_n|$. Обозначим за $p(m, n)$ четность расстояния между клетками m и n: $p(m, n) = (\rho(m, n) \mod 2)$. Клетки c, d соседние, значит $p(a, c)$ отличается от $p(a, d)$ на 1. Клетки a, b тоже соседние, значит $p(a, d)$ отличается от $p(b, d)$ тоже на 1. Следовательно $p(a, c) = p(b, d)$. Клетки a и c находились в одном шахматном множестве, значит если $p(a, c) = 0$, то эти клетки одного цвета, иначе разного. Аналогично с клетками b и d. Получается, что если $p(a, c) = p(b, d) = 0$, то цвет клетки a тот же, что цвет клетки c, цвет клетки b тот же, что цвет клетки d, значит c и d разного цвета. Если $p(a, c) = p(b, d) = 1$, то цвет клетки a отличается, от цвета клетки c, цвет клетки b отличается от цвета клетки d, значит опять c и d разного цвета.

Таким образом, в рассмотренном объединении множеств каждая пара соседних клеток разного цвета. Это доказывает, что получившееся множество является шахматным, и кол-во шахматных множеств уменьшилось на 1, значит разбиение было не минимальным. Противоречие. □

Решение: запустим DFS, в котором будем переходить между соседними клетками, если они различны. Таким образом, DFS пройдет по всем клеткам в одном шахматном множестве. Будем запускать DFS от тех клеток, которые не были посещены предыдущими запусками, пока все клетки не окажутся посещенными. Количество запусков и будет ответом на задачу.

Асимптотика решения: $\mathcal{O}(NM)$

H. [B, A‘] Все смены одинаковые

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

Давайте переформулируем задачу: всё, что нам требуется — это найти все вхождения одного массива в другой с точностью до сдвига всех элементов на какое-то число. Заметим, что если такое вхождение найдётся, то массивы $Adelta : Adelta_i = a_i - a_{i-1}$ и $Bdelta : Bdelta_i = b_i - b_{i-1}$ разностей соседних элементов будут совпадать на этом вхождении. Значит, можно посчитать массивы $Adelta$ и $Bdelta$ и искать вхождения второго в первый; в самом деле, каждому вхождению $Bdelta$ в $Adelta$ будет соответствовать вхождение b в a со сдвигом. Посчитать вхождения $Bdelta$ в $Adelta$ можно найти, например, алгоритмом Кнута — Морриса — Пратта за $\mathcal{O}(|a| + |b|)$.

Альтернативное решение хешами Давайте посчитаем полиномиальный хеш от массивов a и b и научимся с помощью хешей искать все вхождения b в a со сдвигом. Приставим b к a в каждой позиции и проверим вхождение за $\mathcal{O}(1)$ следующим образом.

Найдём разность между первым элементом b и элементом в a, который, как мы верим, должен задавать начало вхождения. Пусть разность между ними равна c. Заметим, что полиномиальное хеширование линейно: $hash(a) + hash(b) = hash(a + b)$ (a, b — массивы целых чисел, $a + b$ здесь и далее обозначает поэлементное сложение); это нетрудно проверить простой подстановкой. Тогда, чтобы убедиться в том, что это вхождение, достаточно сравнить хеши подстрок $[a_i...a_{i+|b|-1}]$ и $b + C$, где C — массив длины $|b|$, состоящий из чисел c. Осталось понять, что $hash(C) = c * hash(O)$, где O — массив единиц длины $|b|$. Предпосчитаем O явно и таким образом решим задачу за $\mathcal{O}(|a| + |b|)$.

I. [B, A‘, A] Калифорнийский стиль

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

Обозначим реперов зелёными и фиолетовыми точками на прямой, где i-му реперу будет соответствовать точка с координатой a_i и цветом, соответствующим цвету его банды. Фиты будем обозначать отрезками между точками. Следующее утверждение показывает, что свобода выбора пары для каждого репера на самом деле весьма ограниченная.

Утверждение. *В оптимальном решении можно считать, что если репер фитует с кем-то левее себя, то это либо ближайший слева репер из его банды, либо ближайший слева репер из противоположной банды.*

Доказательство. Предположим, что это не так: пусть репер с координатой y фитует с репером с координатой x ($x \le y$), и есть репер с координатой z того же цвета, что и репер на x, который фитует с репером с координатой t ($x < z \le y$). Тогда если мы поменяем пары так, что x будет фитовать с t, а z с y, то стоимость ответа не увеличится — цвета в парах остались те же самые, а расстояние точно уменьшилось на $z - x$ (y теперь с z, а не x), а увеличилось максимум на столько же (если $z \le t$). □

Такое ‘жадное‘ распределение пар можно промоделировать с помощью динамического программирования: Пусть $dp[i][w][b]$ — минимальное количество усилий для распределения всех фитов на префиксе i, а w и b — логические флаги ‘фитует ли уже с кем-то последний репер банды Grove Street‘ и ‘банды Ballas‘ соответственно, считаем что все кроме последнего репера каждой банды уже точно фитуют с кем-то. Тогда динамика имеет следующий вид:

```
for (int i = 0; i < n; i++)
for (int w = 0; w < 2; w++)
//w эквивалентно "есть ли на префиксе свободный репер цвета 0"
for (int b = 0; b < 2; b++) {
//b эквивалентно "есть ли на префиксе свободный репер цвета 1"
if (w && (!b || !color[i])) {
     //color[i] - булевый массив, сообщающий цвет точки под номером i
        dp[i + 1][0][b] = min(dp[i + 1][0][b], dp[i][1][b]
        + x[i] - x[last[i - 1][0]]
        + d * color[i]);
        //x[last[i - 1][0]] - координата предыдущей точки с цветом 0
    }
    if (b && (!w || color[i])) {
        dp[i + 1][w][0] = min(dp[i + 1][w][0], dp[i][w][1]
        + x[i] - x[last[i - 1][1]]
        + d * (1 - color[i]));
    }
    if (!color[i] && !w) {
     dp[i + 1][w + 1][b] = min(dp[i + 1][w + 1][b], dp[i][w][b]);
    }
    if (color[i] && !b) {
    dp[i + 1][w][b + 1] = min(dp[i + 1][w][b + 1], dp[i][w][b]);
}
}
cout << dp[n][0][0];
```

Асимптотика решения — $\mathcal{O}(n)$.

J. [A‘, A] Пермское метро

Автор задачи: Александр Гришутин.
Автор разбора: Александр Гришутин.

В условии гарантируется, что есть не более одной вершины степени хотя бы 3, то есть на самом деле данная нам схема метро это граф-звезда: можно выделить вершину (назовём её C), такую что при её удалении граф распадается на ≥ 1 граф-бамбук (последовательность вершин, где каждые две соседние соединены ребром, и больше ребер нет). Назовём эти бамбуки ветками нашего метро (включая ребро из C в первую вершину ветки). Теперь научимся отвечать на запросы.

Сперва научимся отвечать на запросы вида ? A B.

- Если вершины A и B лежат на одной ветке метро, причём B находится дальше (в смысле количества рёбер) A от C, то $|AB| = |CB| - |CA|$, где $|XY|$ — расстояние между вершинами X и Y.
- Если же A и B лежат на разных ветках, то $|AB| = |CA| + |CB|$.

Эти рассуждения наводят нас на следующее решение: давайте для каждой ветки заведём некоторую структуру, поддерживающую суммы на префиксе с возможностью изменения в элементе по индексу (авторские решения используют дерево Фенвика и дерево отрезков). Элементами в нашем случае будут веса рёбер. Тогда запрос расстояния между вершинами — это сумма или разность соответствующих префиксных сумм (в самом начале нужно с помощью обхода в глубину для каждой вершины определить номер её ветки и порядковый номер на этой ветке при обходе из C), а запрос изменения веса ребра — это изменение элемента в соответствующей структуре (для этого нужно с помощью всё того же обхода в глубину предварительно для каждого ребра узнать его номер ветки и порядковый номер при обходе этой ветки из C).

K. [A‘, A] Пермское метро-2

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

Подвесим дерево за какую-нибудь вершину, назовём её корнем дерева. Глубиной вершины будем называть расстояние от неё до корня. Родителем вершины v будем называть вершину u, лежащую на пути из корня в v, с глубиной на 1 меньше, чем у v. Родитель есть у всех вершин, кроме корня. Поддеревом вершины v назовём множество таких вершин u, что v лежит на пути от корня до u. Дети вершины v — это вершины поддерева v с глубиной, на 1 большей, чем у v.

Рассмотрим для каждой вершины u, не являющейся корнем, величину $dp[u]$ — максимальное количество пересадок на пути из родителя u в какую-либо вершину поддерева u.

Рассмотрим путь, являющийся ответом на задачу, пусть v — вершина с минимальной глубиной на этом пути (она всегда единственна). Путь либо идёт из v в поддерево v, либо состоит из двух половинок, каждая из которых идёт из v в поддерево v. В первом случае, если следующая вершина на пути после v — это u, то количество пересадок на пути равняется $dp[u]$. Во втором случае, если следующие вершины после v на половинках пути — это u, w, то количество пересадок на пути равняется $dp[u] + dp[w] + cost(u, v, w)$. Здесь и далее $cost(u, v, w) = 1$, если рёбра $v - u$, $v - w$ лежат на разных ветках, и $cost(u, v, w) = 0$ иначе.

Таким образом, для того, чтобы найти ответ на задачу, достаточно посчитать все величины $dp[u]$, после чего взять максимум по всем величинам вида $dp[u]$, а также по всем величинам вида $dp[u]+dp[w]+cost(u, v, w)$, описанным выше.

Для того, чтобы посчитать все эти величины, воспользуемся алгоритмом обхода в глубину. Если мы пришли в лист u, то $dp[u] = 0$. Иначе $dp[u] = \max(dp[w]+cost(v, u, w))$, где w пробегает по всем детям вершины u, v — это родитель u.

Для того, чтобы вычислить все величины вида $dp[u]+dp[w]+cost(u, v, w)$, нужно для каждой вершины v перебрать все пары детей u, w. К сожалению, такое решение работает слишком медленно, так как в дереве на n вершинах могут встречаться вершины со степенью, близкой к n. На дереве с такими вершинами текущий алгоритм будет работать за $O(n^2)$, так как он перебирает все пары детей каждой вершины.

Для того, чтобы получить решение, работающее за $O(n)$, заметим, что если зафиксировать вершину v и перебирать её ребёнка u, в качестве w достаточно рассмотреть лишь двух детей v — w_1 с максимальным значением dp среди уже рассмотренных детей v, и w_2 с максимальным значением dp среди уже рассмотренных детей v таких, что рёбра $v - w_1$ и $v - w_2$ лежат на разных ветках. Такие максимумы можно поддерживать в том же цикле, в котором перебирается u.

Альтернативное решение.

Существует и более простое с точки зрения написания решение. Утверждается, что достаточно взять любую вершину v, найти поиском в глубину вершину u такую, что на пути от v до u максимальное число пересадок среди всех путей из v. После чего повторить тот же процесс для u, т.е. найти вершину w такую, что на пути от u до w максимальное число пересадок среди всех путей из u. Путь из u в w является ответом на задачу.

Доказательство этого утверждения оставим в качестве упражнения для читателя. Отметим лишь, что и алгоритм, и его доказательство очень похожи на алгоритм поиска диаметра дерева и его доказательство.

L. [A] Назови стартап

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

Переформулируем: нужно найти кратчайшую строку, в которой есть хотя бы K вхождений данных строк.

Заметим, что на самом деле в оптимальном ответе будет ровно K вхождений: иначе можно выкинуть несколько букв с конца, пока число вхождений не уменьшится до K. Выкидывание одной буквы с конца уменьшает число вхождений максимум на 1, так как никакая из данных строк не является подстрокой другой.

Заметим также, что оптимальный ответ всегда кончается на одну из данных строк: иначе можно было бы выкинуть из него последнюю букву и не уменьшить число вхождений, получив таким образом более короткий ответ.

Пусть S — множество данных строк, $ans(i, s)$ — наименьшая длина строки, которая содержит i вхождений и кончается на $s \in S$. Ясно, что $ans(1, s) = |s|$.

Утверждение. *Если $i \geq 2$, то*

$$ans(i, s) = \min_{t \in S} ans(i - 1, t) + d(t, s),$$

где $d(t, s)$ — минимальное число символов, которое нужно дописать к t, чтобы получить строку, кончающуюся на s.

Доказательство. Рассмотрим некоторую строку, в которой i вхождений, и которая кончается на s. Так как $i \geq 2$, в ней есть еще какие-то вхождения исходных строк, кроме s в самом конце, выберем из них самое последнее, пусть это строка t. Таким образом, в нашей строке есть вхождение t, а потом вхождение s. При этом вхождение s может накрывать только какой-то суффикс t, но не всю t целиком (тогда t было бы подстрокой s, что невозможно по условию), и не может выходить перед t. Таким образом, можно оставить максимально короткую строку, имеющую $i - 1$ вхождений, и кончающуюся на t, и за $d(t, s)$ дополнительных символов дописать s — за счет предыдущего наблюдения эти два действия независимы. □

То есть, мы получили динамическое программирование для ans. Достаточно вычислять значения по формуле из предположения по возрастанию i, в конце ответом будет $\min_{s \in S} ans(k, s)$ (разумно использовать в качестве индексов номера строк, а не сами строки). Осталось только заранее посчитать $d(t, s)$ для каждой пары исходных строк.

Заметим, что минимизировать число дописанных символов, чтобы из t получить s это все равно что максимизировать наложение s на t, то есть найти максимальный суффикс t, являющийся префиксом s. Это можно сделать за $O(|s| + |t|)$: посчитать Z-функцию строки $s\$t$ или использовать хеширование (но в задачу были включены тесты против хеширования по модулю 2^{64}). Сделать это для каждой пары строк можно за время $O(\sum_{s,t \in S} |s| + |t|) = O(N \sum_{s \in S} |s|)$. Итоговая асимптотика $O(N^2 K + N \sum_{s \in S} |s|)$.

Ещё один способ восприятия решения выше — представить, что мы строим взвешенный граф, в котором вершины это данные строки, а ребро из вершины t в вершину s имеет вес $d(t, s)$. Тогда нас интересует кратчайший путь длины K (в вершинах). Его можно найти алгоритмом Форда–Беллмана, если на каждой итерации использовать новый массив расстояний (а не всё время один и тот же). Нетрудно заметить, что это получится в точности описанное динамическое программирование. Граф, определенный таким образом, называется префиксным графом и широко используется для анализа известной задачи о надстроке — по данному набору строк найти кратчайшую строку, содержащую их все как подстроки.