Задача A. Инопланетные мелодии

Решение с динамическим программированием

Заметим, что после того как инопланетянин сыграет первые i нот, есть всего 4 варианта для клавиши, на которой он закончил. Для дальнейшей игры важна лишь последняя клавиша и число совершённых нарушений перед этим.

Тогда посчитаем $dp[i][j]$, i — количество сыгранных нот, j — последняя нажатая клавиша, $dp[i][j]$ — минимальное число нарушений, с которыми возможно сыграть первые i нот и закончить на клавише j.

Жадное решение

Удалив подряд равные ноты, можно рассматривать только переходы между различными соседними нотами. Каждый такой переход либо вверх, либо вниз. Если существует четыре подряд восходящих или четыре подряд нисходящих перехода, то их невозможно отобразить на четырёх клавишах, так как потребуется пятая клавиша соответствующей полярности. Обратная сторона этого утверждения заключается в том, что при отсутствии подряд четырёх шагов одного направления всегда найдётся корректное отображение: обозначив клавиши как A,B,C,D от низкой к высокой, можно назначать первой ноте A при первом шаге вверх и D при первом шаге вниз, затем сдвигать назначение на одну клавишу по направлению шага, а при локальной смене направления внутри трёх нот (вверх затем вниз или вниз затем вверх) присваивать средней ноте соответствующую крайнюю клавишу, что предохраняет от выхода за границы диапазона.

Отсюда следует простая жадная стратегия: идти слева направо по последовательности без подряд равных элементов и поддерживать два счётчика подряд идущих восходящих и нисходящих шагов. Пока ни один счётчик меньше четырёх, фрагмент остаётся корректным. Как только один из счётчиков достигает четырёх, требуется разрыв, то есть одно нарушение, после чего счётчики сбрасываются, и текущая пара нот рассматривается как начало нового фрагмента. В любой оптимальной разбивке через точку первого появления четырёх однонаправленных шагов нельзя пройти, поэтому такой жадный выбор локально и глобально оптимален.

Алгоритм работает за линейное время относительно количества нот: удаление подряд повторов и один проход по оставшейся последовательности дают сложность $O(k)$, а дополнительная память константна. Реализация ниже соответствует описанному подходу: сначала формируется массив без подряд равных элементов, затем подсчитываются подряд направления и выполняются разрывы при достижении длины четыре.

Задача B. Пропуск КП

Научимся сначала отвечать на запрос `D i j` — длина маршрута от пункта i до пункта j, если проходить пункты по очереди. Скажем что точка p_i соответствует пункту i.

В этом случае путь будет проходить по всем пунктам от i до j, проходя расстояния $dist(p_i, p_{i+1}), dist(p_{i+1}, p_{i+2}), \ldots, dist(p_{j-1}, p_j)$. Достаточно поддерживать массив $d[i] = dist(p_i, p_{i+1})$ и уметь отвечать на запросы суммы на отрезке этого массива. Поддерживать значения несложно — за одно изменение координат точки p_i меняется всего два расстояния d_{i-1}, d_i. Отвечать на запросы суммы на отрезке будем с помощью дерева отрезков.

Научились обновлять координаты i-го пункта и вычислять длину пути между пунктами i, j. Осталось научиться учитывать возможность пропуска одного пункта.

Пропуск пункта номер l меняет пройденное расстояние следующим образом (по сравнению с проходом по всем пунктам): $-dist(p_{l-1}, p_l) - dist(p_l, p_{l+1}) + dist(p_{l-1}, p_{l+1})$. Из всех таких изменений (пропусков) нас интересует самый выгодный. Тогда заведём массив c, заполненный выгодой каждого возможного пропуска, то есть $c_l = d_{l-1} + d_l - dist(p_{l-1}, p_{l+1})$.

Теперь для ответа за запрос `Q i j` посчитаем сумму $\sum_{l=i}^{j-1} d_l$ и вычтем максимальную выгоду по возможным пропускам $\max_{l=i+1}^{j-1} c_l$. Запросы максимума на отрезке и обновления массива c также выполняем с помощью дерева отрезков.

Задача C. Последовательное разбиение

Заметим, что если группа, начинающаяся с индекса $x = i$, заканчивается в индексе j, то далее процесс будет полностью совпадать с процессом для $x = j+1$. Чтобы получить для каждого элемента

длину первой группы достаточно воспользоваться методом двух указателей — идти слева направо и поддерживать индекс конца группы, ведь правая граница группы не убывает при увеличении индекса левого конца группы.

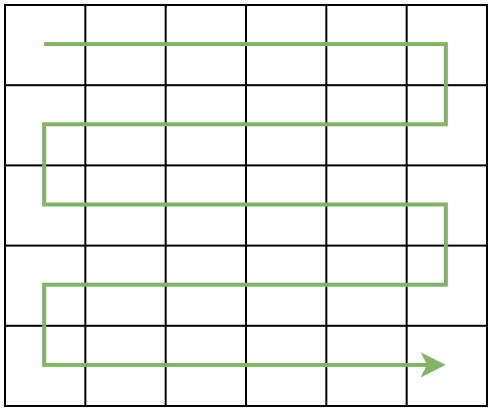
Теперь если для $x = j + 1$ ответ это k_{j+1} — количество групп и s_{j+1} — сумма элементов в последней группе, если начинать с $x = j + 1$, то для $x = i$ ответы будут равны соответственно $k_{j+1} + 1$ и s_{j+1}.

Остаётся пройти с конца массива в начало и пересчитать ответы для всех индексов.

Задача D. Путь по кампусу ФПМИ

Заметим, что если r нечётно, то возможно посетить все клетки поля. Если c нечётно — аналогично, возможно посетить все клетки поля.

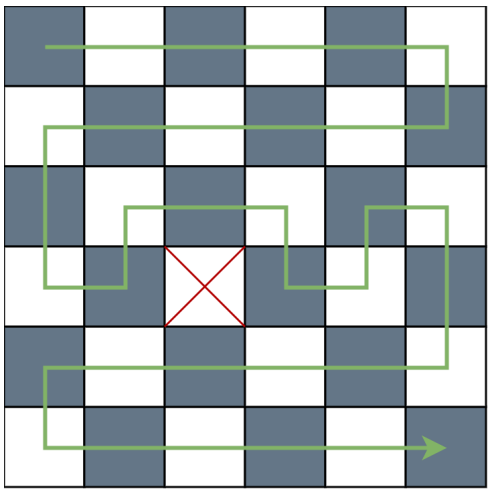
Пример, когда хотя бы одна из сторон нечётная:



Но если r чётно и одновременно c чётно, то невозможно посетить все клетки. А именно, если использовать шахматную раскраску (верхнюю-левую клетку покрасим в чёрный цвет), то цвета в пути чередуются, причём первый и последний цвета одинаковые, а значит хотя бы одна клетка белого цвета осталась не посещённой.

Оказывается в этом случае для любой белой клетки существует путь, который проходит по всем клеткам доски кроме неё.

Приведём пример:



А именно всегда возможно по всем парам строк перед выбранной клеткой ходить зигзагом, по всем парам строк после выбранной клетки также ходить зигзагом, а внутри пары строк, включающих выбранную белую клетку, ходить вертикальным зигзагом, сменяющим направление после выбранной белой клетки.

Построенный таким образом путь даёт конструктивный ответ на задачу, если в качестве не посещённой белой клетки выбирать белую клетку с наименьшим значением интересности.

Задача E. Взвешенная точка с манхеттенской окружностью

Заметим, что круг в смысле манхэттенского расстояния это ромб на плоскости. Повернём координаты — получим набор квадратов. Причём при повороте координаты каждой точки это номера диагоналей, на которых она находится, то есть $(x+y, x-y)$, а сторона квадрата становится равной $2k$.

Задача свелась к поиску точки на плоскости, покрытой максимальным по сумме весов набором квадратов. Такая задача решается с помощью сканирующей прямой — пойдём слева направо по возможным координатам x и будем поддерживать для каждого значения y-координаты сумму весов всех квадратов, которые пересекают точку (x, y). Состояние сканирующей прямой меняется только на границах квадратов. Достаточно завести массив границ (открывающихся и закрывающихся) и в порядке слева направо обрабатывать границы и каждый раз обновлять значения на сканирующей прямой. Когда новый квадрат открывается — прибавляем значение g_i на отрезке $[y_1, y_2]$, когда закрывается — вычитаем. После каждого изменения ищем максимальное значение по всем y-координатам и обновляем ответ.

Задача F. Построй путь для каравана

Заметим, что для кратчайшего пути достаточно рассматривать окрестность размера $150 \cdot 150$. Если не учитывать направление — путь из точки A в точку B займёт не больше чем сумма разностей их координат, а на разворот (получение нужного направления) необходимо не более чем $3n$ дополнительного расстояния по каждой координате (что видно из примера в условии). Из этого следует что достаточно некоторого запаса по координатам, в данном случае буфера в 50 в каждую сторону хватает для получения вердикта OK.

Заметим также, что для каждой точки из этой окрестности и каждого из восьми направлений нас интересует лишь кратчайший способ попасть сюда в конце одного из отрезков ломаной. Значит можем запустить алгоритм Дейкстры и найти ответ на задачу.

Запустим алгоритм Дейкстры одновременно из обоих концов первого отрезка, и обновим ответ, когда доберёмся до концов второго отрезка под нужным направлением.

Задача G. Угадай граф

Начнём с вершины 1 и удалим все ребра, входящие в неё. Будем добавлять рёбра $(1,2), (1,3), \ldots$ по одному, проверяя связность после каждого добавления. После добавления некоторого ребра $(1,k)$ граф станет связным (он должен быть соединен после добавления всех ребер). Это означает, что

1. $(1,k)$ является ребром графа
2. После удаления ребер $(1,k+1), (1,k+2), \ldots (1,n)$ ребро $(1,k)$ является мостом в результирующем графе.

Теперь мы удаляем ребро $(1,k)$ и добавляем ребро $(1,k+1)$. Если результирующий граф связен, то $(1,k+1)$ является ребром графа. Затем мы добавляем ребро $(1,k+1)$ и удаляем $(1,k+2)$, из чего мы можем определить, является ли $(1,k+2)$ ребром графа. Повторяя это, мы можем определить, находится ли в графе каждое ребро $(1,i)$ для $i \geqslant k$.

Для всех будущих запросов мы удаляем каждое ребро $(1,i)$ для $i > k$, и никогда больше не удаляем ребро $(1,k)$. Для каждого из этих ребер мы знаем, принадлежит ли оно графу. Для любого будущего запроса к графу ребро $(1,k)$ будет мостом.

Мы повторяем этот процесс с новой произвольно выбранной вершиной i. Мы удаляем все ребра, инцидентные i, которые еще не были идентифицированы как мосты. Если результирующий граф связен, то мы можем доказать, что ни одно из этих ребер не присутствовало в графе. В этом случае мы удаляем все эти ребра для всех будущих запросов. В противном случае мы продолжаем, как мы делали с вершиной 1, и находим другой мост.

Мы продолжаем до тех пор, пока не будет определен весь граф. Каждый раз, когда мы повторяем этот процесс, мы либо удаляем все оставшиеся немостовые ребра, входящие в вершину, либо находим новое ребро, которое является мостом во всех будущих графах запросов. Таким образом, этот процесс может быть повторен только 40 раз, и каждый раз мы делаем не более 39 запросов.

Если быть чуть более аккуратными, мы можем сократить количество запросов до $41 * 40/2 + 40$. Для этого мы гарантируем, что в каждой точке найденный нами набор мостов образует дерево, и мы всегда выбираем вершину дерева для следующего выполнения внутреннего цикла. Нам не нужно проверять ни одно из ребер между этой вершиной и любой другой вершиной дерева. Поскольку каждый запрос либо увеличивает дерево мостов, либо удаляет вершину из рассмотрения, для K-го вызова внутреннего цикла требуется выполнить не более $(42 - K)$ запросов. Общее количество запросов не должно превышать $41 * 40/2 + 40$.

Задача H. Стоимость задач

Если заранее знать порядок решения — можно будет рассматривать исходную задачу как задачу о рюкзаке, максимизируя число полученных очков в момент времени t.

Оказывается возможно определить оптимальный порядок для задач. Пусть команда решала задачи в некотором порядке. Попробуем поменять местами две задачи i и j, если они были решены по очереди. Тогда число полученных очков изменится на $-b_i \cdot c_j + b_j \cdot c_i$. Видно, что число полученных очков увеличится, если $-b_i \cdot c_j + b_j \cdot c_i > 0$, то есть если $b_j \cdot c_i > b_i \cdot c_j$, что эквивалентно $\frac{b_j}{c_j} > \frac{b_i}{c_i}$. Значит задачи решают в порядке уменьшения $\frac{b_i}{c_i}$ (при другом порядке мы можем улучшить получаемые очки, упорядочив все решения). Тогда будем насчитывать $dp_i[T]$, равное максимальному числу очков, которое можно набрать, решив некоторые из первых i задач и закончив решать в момент времени $0 \leqslant T \leqslant t$. Такие значения пересчитываются при добавлении новой задачи как $dp_i[T + c_i] = \max(dp_{i-1}[T + c_i], dp_{i-1}[T] + a_i - (T + c_i) \cdot b_i)$.

Задача I. Биномиальные коэффициенты

Из условия минимальности следует, что достаточно рассматривать пары n, k такие, что $2k \leqslant n$.

Переберём значение k. Докажем что достаточно перебрать несколько первых значений k.

При $2k \leqslant n$ $C_n^k \leqslant C_{n+1}^k$. К тому же при $2k = n$ $C_{2k}^k \geqslant \frac{2^{2k}}{2k+1}$, так как сумма всех комбинаций равна 2^{2k}, а C_{2k}^k принимает наибольшее значение. Из этого следует что достаточно рассмотреть только первые $0 \leqslant k \leqslant 175$, так как при $k = 175$ значение $C_{2k}^k \geqslant \frac{2^{350}}{350} \approx \frac{10^{105}}{350} \geqslant 10^{100}$.

Теперь для каждого значения k хотим найти подходящее значение n — для этого воспользуемся бинарным поиском. Заметим, что в процессе бинарного поиска будем производить следующее сравнение: $X \leqslant \frac{n \cdot (n-1) \cdot \cdots \cdot (n-k+1)}{k!}$, что эквивалентно сравнению без деления $X \cdot k! \leqslant n \cdot (n-1) \cdot \cdots \cdot (n-k+1)$.

Из всех найденных решений выберем оптимальное.

В процессе решения приходится использовать вычисления в длинной арифметике, а именно — умножение, сложение и деление на 2. Эти операции возможно определить в `C++` или воспользоваться языками программирования со встроенными вычислениями с длинными числами, например, `Python` или `Java`.

Задача J. Игра со связностью

На каждом ходу происходят следующие изменения: меняется число рёбер, которое можно добавить не меняя число компонент связности и может поменяться число компонент связности, если ребро было проведено между двумя различными компонентами связности. Тогда нас интересуют следующие параметры: число компонент, размеры компонент и количество доступных рёбер, которые возможно добавить не меняя число компонент связности.

У компоненты связности размера a всего $\frac{a \cdot (a-1)}{2}$. Когда объединяются компоненты связности размеров a и b, то число доступных рёбер меняется на $\frac{(a+b) \cdot (a+b-1) - a(a-1) - b(b-1)}{2} = \frac{a^2+b^2+2ab-a-b-a^2+a-b^2+b}{2} = \frac{2ab}{2} = ab$.

Предположение — достаточно учитывать только чётность числа доступных рёбер.

Изменение чётности числа рёбер — чётность числа доступных рёбер меняется на один, если добавлено ребро внутри некоторой компоненты, и меняется на $ab - 1$ иначе. Значит достаточно знать чётность размеров компонент.

Итак, насчитаем $dp[c0][c1][ce]$, где $c0$ — число компонент чётного размера, $c1$ — число компонент нечётного размера, ce — число компонент нечётного размера, ce — чётность числа доступных рёбер и $dp[c0][c1][ce]$ — выигрышность позиции. При пересчёте не будем учитывать переход из $ce = 0$ в $ce = 1$.

Далее доказательство, что подсчёта таких переходов достаточно.

Докажем по индукции по числу компонент.

Пусть для всех $c0', c1'$, таких что $c0' + c1' < c0 + c1$ условие выше выполнено (то есть выигрышность позиции зависит только от этих параметров и чётности ce).

Тогда докажем для $c0, c1$ индукцией по ce.

Если $dp(c0, c1, ce) \neq dp(c0, c1, ce-2)$, то: Если ce нечётно, то набор доступных за один шаг позиций не отличается по выигрышности из предположения индукции, значит такого быть не может.

Если ce чётно, то новый переход может возникнуть при ненулевом числе доступных рёбер в виде перехода из $(c0, c1, ce)$ в $(c0, c1, ce - 1)$ (мы хотим доказать что такой переход нет необходимости рассматривать).

Пусть $(c0, c1, ce - 2)$ — проигрышная. Тогда после перехода из $(c0, c1, ce)$ в $(c0, c1, ce - 1)$ наш соперник переместится в $(c0, c1, ce - 2)$ и мы проиграем. Остальные переходы совпадают.

Пусть $(c0, c1, ce - 2)$ — выигрышная.

1. Один из переходов, меняющих число компонент, даёт выиграть. Такой переход есть и в $(c0, c1, ce)$.

2. Переход в $(c0, c1, ce - 3)$ — выигрышный. Значит позиция $(c0, c1, ce - 3)$ проигрышная и все переходы из неё — в выигрышные позиции, следовательно $(c0, c1, ce - 1)$ тоже проигрышная. Значит позиция $(c0, c1, ce)$ выигрышная.

Задача K. Генерация вхождениями шаблонов

Для начала найдём число вхождений, заканчивающихся в индексе i имеющихся шаблонов в текст (для всех i). Эта задача решается с помощью автомата Ахо-Корасик. Во всех индексах, в которых чётность числа вхождений совпадает с символом текста — шаблон не заканчивается, назовём эти индексы неподходящими. Во всех индексах, в которых чётность числа вхождений не совпадает с очередным символом текста — шаблон заканчивается, назовём эти индексы подходящими.

Для каждого индекса получим множество подстрок, которые в нём заканчиваются.

Значит мы хотим рассмотреть пересечение таких множеств в подходящих индексах (и затем вычеркнуть варианты, которые совпадают с подстроками, совпадающими с заканчивающимися в неподходящих индексах, а также вычеркнуть все уже имеющиеся шаблоны).

Чтобы получить пересечение — возьмём первый подходящий индекс i. Тогда шаблон — какой-то суффикс подстроки $s_{0..i}$. Назовём строкой $r = reverse(s_{0..i})$ развёрнутый этот суффикс. Мы хотим найти пересечение таких строк для всех i. Для этого развернём строку t (текст) и насчитаем z-функцию на строке $r\#t$. Теперь возьмём минимум значения z-функции во всех подходящих индексах и получим максимальную длину кандидата в шаблоны.

Чтобы вычеркнуть всех возможных кандидатов, заканчивающихся в неподходящих индексах — возьмём максимум значений z-функции в этих индексах. Тогда интересующая нас длина шаблона обязана превышать это значение (иначе такой кандидат обязательно заканчивается в неподходящем индексе).

Осталось исключить все уже имеющиеся шаблоны из кандидатов. Для этого достаточно сложить развёрнутые версии всех имеющихся шаблонов в бор и прочитать развёрнутого наибольшего кандидата в шаблоны в этом боре. Если очередная полученная вершина терминальная — нам такой вариант не подходит.

Задача L. Вкусность конфет

Насчитаем $dp[i][j]$, где i — индекс последней взятой конфеты, j — индекс предпоследней взятой конфеты, $dp[i][j]$ — максимальная суммарная полезность конфет без учёта $a[i]$.

Ближайшая конфета, которую в этом случае возможно взять, это конфета под номером $j + k$, поэтому:

$dp[j + k][i] = \max(dp[j + k][i], dp[i][j] + a[i])$

Тогда пересчёт в общем виде выглядит следующим образом:

Для всех $p, q \geqslant 0$:

$dp[j+k+p][i+q] = \max(dp[j+k+l][i+q], dp[i][j] + a[i])$

Такие переходы можем заменить следующими тремя:

1. $dp[j+k][i] = \max(dp[j+k][i], dp[i][j] + a[i])$
2. $dp[i+1][j] = \max(dp[i+1][j], dp[i][j])$
3. $dp[i][j+1] = \max(dp[i][j+1], dp[i][j])$

Задача M. Перемещения по ПСП

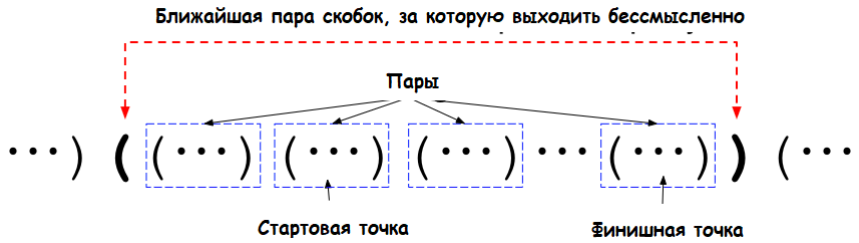
Задача предлагает строку из круглых скобок, которая является правильной скобочной последовательностью. Для каждой позиции известна стоимость перемещения курсора влево, вправо и телепортации к парной скобке. Необходимо ответить на несколько запросов: найти минимальное время, за которое можно переместить курсор из позиции s_j в позицию e_j, если можно перемещаться шагами по соседним символам или мгновенно переходить к парной скобке за стоимость p_i.

На первый взгляд кажется, что можно просто запускать поиск кратчайшего пути (например, алгоритм Дейкстры) для каждой пары (s_j, e_j). Однако при длине строки и числе запросов до 10^5 это будет слишком медленно: одно вычисление пути требует $O(k \log k)$, а значит, общее время стало бы $O(qk \log k)$, что недопустимо. Решение требует использования структуры правильной скобочной последовательности.

Каждая пара скобок образует некоторый диапазон символов, и все такие диапазоны строго вложены друг в друга. Если рассматривать каждую пару скобок как вершину, а её ближайшую внешнюю пару как родителя, то эти отношения образуют дерево вложенности. Вершины этого дерева соответствуют парам скобок, а листья — отдельным элементам внутри самых глубоких пар. Таким образом, вся строка имеет иерархическую структуру: чтобы попасть из одной области внутрь другой, нужно «пересечь» границу какой-то пары скобок. Это свойство является ключевым для эффективного решения.

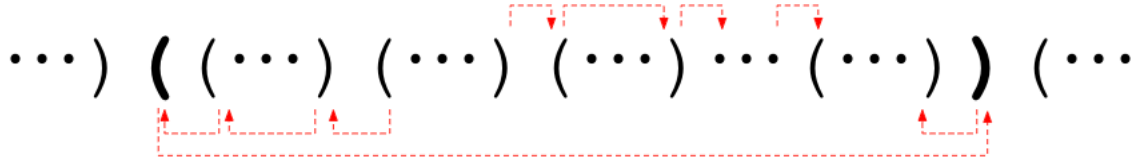
Курсор может перемещаться по строке тремя типами действий: шаг влево, шаг вправо и телепортация к парной скобке. Движение вдоль соседних символов можно рассматривать как переход по рёбрам графа, а телепортация добавляет дополнительные рёбра между соответствующими позициями. В результате получаем ориентированный взвешенный граф на k вершинах, где из каждой вершины i исходят рёбра: влево с весом l_i, вправо с весом r_i, и к позиции $\text{match}(i)$ с весом p_i. Граф ориентированный, потому что l_i и r_i могут отличаться, и стоимость пути в одну сторону не обязана совпадать со стоимостью обратного пути. Это нужно учитывать в реализации: для корректных расстояний следует запускать поиск в обе стороны.

Чтобы понять, как устроен кратчайший путь, заметим несколько закономерностей. Если начальная и конечная позиции находятся внутри одной пары скобок, оптимальный путь никогда не выйдет за её пределы, так как любое перемещение наружу потребует дополнительных переходов через границы скобок.



Если начальная и конечная позиции находятся в разных вложенных областях, то путь обязательно проходит через ту пару скобок, которая является наименьшим общим предком этих областей в дереве вложенности.

Телепортации позволяют быстро переходить между концами одной пары, и обычно выгодно использовать их, если они не заставляют «перепрыгивать» через нужную позицию.



Из этих наблюдений следует, что для оптимального пути достаточно рассматривать перемещения внутри поддеревьев дерева вложенности и переходы через пары скобок, являющиеся предками этих областей. Таким образом, структура задачи сводится к движению по дереву, где каждое поддерево соответствует фрагменту строки, а переходы через границы пар скобок — это «выходы» из этих поддеревьев.

Эти наблюдения позволяют ограничить пространство возможных движений и подсказывают структуру оптимальных путей. Чтобы эффективно отвечать на запросы, нужно уметь быстро находить кратчайшие пути между позициями, которые могут принадлежать разным поддеревьям. Для этого удобно рассматривать специальные позиции — границы скобок, через которые проходят все пути, соединяющие разные области. Если запустить алгоритм Дейкстры из таких границ, можно вычислить расстояния от них до всех позиций в соответствующем фрагменте строки. Тогда для любого запроса, путь которого обязательно проходит через одну из этих границ, ответ можно получить как сумму расстояний от начала до границы и от границы до конца. Это позволяет обработать сразу целую группу запросов без отдельного поиска для каждого.

После того как обработаны все запросы, чьи пути проходят через выбранные границы, оставшиеся запросы можно рассматривать независимо в областях по обе стороны от этих скобок. Таким образом, задача естественным образом делится на подзадачи — внутри скобок и снаружи. В каждой подзадаче можно применить ту же идею рекурсивно. Чтобы рекурсия была сбалансированной, нужно выбирать такие скобочные пары, которые делят текущий фрагмент строки примерно пополам.

Рассмотрим, как именно выполняется такое деление. Пусть мы выбрали некоторую пару скобок и её родительскую пару — то есть ближайшую внешнюю, которая содержит первую. Вся строка между этими двумя парами скобок распадается на четыре области: внутреннюю область, которая находится внутри младшей пары; две боковые области — слева и справа от неё, но внутри родительской пары; и внешнюю область, которая лежит вне родительской пары. Эти четыре области полностью не пересекаются. Чтобы попасть из одной области в другую, нужно пересечь хотя бы одну из выбранных пар скобок, а значит, при переходах между областями путь обязательно проходит через одну из «особых» позиций — концов этих двух пар.

Например, для строки

$$A(B(C)D)E$$

родительская пара — это скобки вокруг всей подстроки `(B(C)D)`, а внутренняя пара — это скобки вокруг `(C)`. Тогда четыре области будут следующими:

```
A | (B | (C) | D)E
```

где каждая вертикальная черта условно показывает границу между областями. Левая область — всё, что вне родительской пары слева; две средние — содержимое внутри родительской пары, разделённое внутренней парой; правая — остаток снаружи. Любое перемещение из одной части в другую обязательно проходит через одну из скобок, выбранных для разделения.

Такое разбиение гарантирует, что ни одна из областей не содержит более половины символов исходного фрагмента: родительская пара по определению охватывает больше половины, а её внутренняя дочерняя пара — не больше половины. Следовательно, внешняя часть, а также области слева и справа от дочерней пары, заведомо содержат меньше половины символов. Таким образом, на каждом шаге рекурсии мы делим текущую задачу на несколько меньших, каждая из которых имеет размер не более $k/2$.

Благодаря этому глубина рекурсии составляет $O(\log k)$. На каждом уровне выполняется несколько запусков алгоритма Дейкстры (из концов выбранных скобок), что даёт $O(k \log k)$ времени на уровень и в сумме $O(k \log^2 k)$ на всё решение. Каждый запрос участвует в обработке не более чем на $O(\log k)$ уровнях, поэтому общая сложность с учётом запросов равна $O(k \log^2 k + q \log k)$ при линейной памяти.

При реализации нужно учитывать, что граф ориентированный, а также аккуратно работать с границами строки, где движение влево или вправо не выполняется. Значение «бесконечности» для Дейкстры должно быть достаточно большим, но не превышать допустимый диапазон целых чисел. Если всё сделать корректно, алгоритм даёт минимальное время перемещения курсора для всех запросов.

Таким образом, решение строится на идее рассматривать правильную скобочную последовательность как дерево вложенных областей и использовать рекурсивное деление по парам скобок, которые делят строку примерно пополам. Предвычисление расстояний от концов выбранных пар с помощью алгоритма Дейкстры позволяет быстро отвечать на запросы, а сбалансированная рекурсия обеспечивает асимптотику $O(k \log^2 k + q \log k)$ при $O(k)$ памяти.

Задача N. Хорошие раскраски – 8

Ключевым является *теорема Визинга*, утверждающая, что для любого простого неориентированного графа хроматический индекс (минимальное число цветов, достаточное для правильной раскраски рёбер) равен либо $\Delta(G)$, либо $\Delta(G) + 1$, где $\Delta(G)$ — максимальная степень вершины. Это означает, что раскрасить рёбра в $d + 1$ цветов всегда возможно, если d — это верхняя граница степеней вершин. Следовательно, задача всегда имеет решение, и его нужно эффективно построить.

Мы будем поочерёдно рассматривать рёбра и назначать им цвета, поддерживая для каждой вершины множество свободных цветов — тех, которые ещё не использованы на её инцидентных рёбрах.

Пусть мы обрабатываем ребро (u, v). Если существует цвет c, свободный и у u, и у v, то красим ребро этим цветом — это тривиальный случай.

Если такого цвета нет, то применяем метод дополняющей цепочки. Построим такую *цепочку* от вершины u: начинаем с текущего ребра $(u, v_0) = (u, v)$, а затем добавляем рёбра $(u, v_1), (u, v_2), \ldots$, для которых выполняется условие: цвет ребра (u, v_i) свободен в вершине v_{i-1}. Этот процесс продолжается, пока можно добавить новые вершины.

Пусть c — некоторый свободный цвет в u, а d — свободный цвет в последней вершине v_t построенной цепочки. Если $c \neq d$, то можно «перекрасить» цепочку рёбер, чередующих цвета c и d, начиная от u и распространяясь вдоль соответствующих рёбер. Это так называемая *перестановка цветов* (или цветовая инверсия по пути (c, d)). После этого в вершине u освобождается цвет d, который можно назначить первому ребру цепочки. Таким образом, удаётся корректно окрасить новое ребро, не нарушая ограничений.

Вся процедура опирается на локальные изменения (перекраски по путям), что гарантирует сохранение корректности уже окрашенных рёбер и конечное завершение процесса.

Пусть n — число вершин, m — число рёбер, а d — максимальная степень. Каждое ребро рассматривается один раз, а перекраски затрагивают не более $O(d)$ рёбер. Таким образом, асимптотическая сложность — $O(nd^2)$ или $O(md)$.