Задача A. Цепная реакция

Автор задачи и разработчик: Григорий Хлытин

Посмотрим на задачу под таким углом: узлы — вершины графа, связи — ребра, и требуется найти кратчайший путь из вершины u в вершину v, если некоторые ребра могут «закрываться» в определенные периоды времени.

Чтобы найти кратчайший путь во взвешенном графе, можно воспользоваться *алгоритмом Дейкстры*. Остается только учесть тот факт, что по некоторым ребрам нельзя перемещаться сразу же после попадания в их начало. Например, если первый момент времени, в который энергия добирается до некоторой вершины a, равен t, а ближайший период времени, в течение которого ребро $a \to b$ веса w (проход по которому занимает w времени) пропускает энергию, равен $[x, y]$, то энергия попадет в вершину b по этому ребру в момент времени $x + w$.

Давайте запустим алгоритм Дейкстры, но будем считать, что вес такого ребра равен не w, а $(x - a) + w$, то есть сумме времени ожидания до «открытия» ребра плюс его исходный вес. В таком случае после того, как такой алгоритм отработает, в каждой вершине будет отмечено кратчайшее расстояние от вершины u до нее.

Недостающая деталь — необходимость быстро находить для каждого такого ребра ближайший интервал времени, в который энергия может по нему перемещаться. Для этого, если алгоритм Дейкстры запоминает кратчайшие расстояния в массив `d`, достаточно в вершине a научиться быстро находить такой минимальный $i \leqslant k$, что $y_i \geqslant \mathtt{d}[a]$. Это можно было сделать двоичным поиском (с небольшой вероятностью это решение могло пройти), либо, заметив, что расстояния в алгоритме Дейкстры не уменьшаются, двигаться по i аналогом метода *двух указателей*.

Время работы такого решения — $\mathcal{O}(m \log n + k)$.

Задача B. Производство Мерцания

Автор задачи и разработчик: Владимир Рябчун

Для начала научимся находить, сколько максимум компонентов можно произвести за время T, если T зафиксировано. Для этого посчитаем для каждого станка, сколько он произведет деталей. Очевидно, что до станков выгодно добираться по кратчайшему пути. Запустим алгоритм Дейкстры из первой вершины и найдём d_i — величину кратчайшего пути до вершины i, для всех i. Тогда, имея в распоряжении T единиц времени, i-й станок сможет произвести $\max\left(0, \left\lfloor \frac{T - 2d_{v_i} - h_i}{t_i} \right\rfloor\right)$.

Осталось выбрать k максимальных значений. Это можно сделать с помощью встроенной в C++ функции `nth_element`, которая просто реализует линейный алгоритм поиска порядковой статистики, или просто с помощью сортировки.

Теперь заметим, что если за время T_1 возможно набрать V компонентов, то за время $T_2 > T_1$ это сделать тем более возможно. Тогда можно сделать двоичный поиск по T, и при помощи описанного выше алгоритма проверять, подходит ли данное T. При проверке, можно ли набрать за данное время хотя бы V компонентов, надо аккуратно обрабатывать случаи переполнения, постоянно проверяя, что текущая сумма меньше V.

Решение требует одного запуска алгоритма Дейкстры и бинпоиска по ответу. Можно показать, что существует тест, где ответ имеет порядок 10^{18}. Поэтому суммарное время работы будет $O(m \log n + s \log 10^{18})$ или $O(m \log n + s \log s \log 10^{18})$

Задача C. Опасные игры

Автор задачи и разработчик: Даниил Голов

Будем решать задачу бинпоиском. Пойдем в одну из центральных клеток квадрата и спросим, где относительно нее находится искомая. Заметим, что результаты «`NW`», «`NE`», «`SW`», «`SE`» однозначно обозначают верхний левый, верхний левый, нижний левый и нижний правый квадраты от точки запроса, и можно сократить область для следующего запроса в 4 раза. Таким образом, достаточно поддерживать текущие левую, правую, нижнюю и верхнюю границы актуальной зоны поиска.

Результаты «`N`», «`S`», «`W`» и «`E`» смещают лишь только одну из данных границ вместо двух. Однако по ним можно однозначно понять, в каком столбце или какой строке находится искомая клетка, так что на самом деле область уменьшается намного больше.

Таким образом, при каждом запросе область поиска будет сокращаться в 4 раза, пока конечная точка поиска не будет локализована в одном столбце или одной строке, после чего зона поиска будет сокращаться в 2 раза. А тогда итоговая асимптотика составит $\mathcal{O}(\log_4(n^2)+\log_2(n)) = \mathcal{O}(\log_2 n)$. Для ограничений задачи количество запросов в любом случае не превысит 50.

Задача D. Тренировки миротворцев

Автор задачи и разработчик: Даниил Орешников

Рассмотрим какую-то последовательность действий, приводящую к площади $\frac{s}{2}$ за минимальное число шагов.

Заметим, что если какие-то два миротворца совершали движения в одну и ту же сторону, можно было бы добиться той же площади, подвинув вместо этого третьего в противоположную сторону. Такая последовательность шагов была бы на один шаг короче. По аналогичным причинам никакой миротворец не совершает два шага в противоположные стороны, иначе оба эти шага можно просто удалить из последовательности.

Не теряя общности, будем считать, что первый миротворец ходит только влево и вниз, второй — только вверх, а третий — только вправо. Тогда если первый прошел расстояние влево x_1, вниз — y_1, второй прошел вверх y_2, и третий прошел вправо x_2, формула для площади получившегося треугольника будет выглядеть как

$$D = (x_1 + x_2)(y_1 + y_2) - \frac{x_2 y_2}{2} - \frac{x_1(y_1 + y_2)}{2} - \frac{y_1(x_1 + x_2)}{2} = \frac{x_1 y_2 + x_2 y_1 + x_2 y_2}{2}$$

Несложно заметить, что если $x_1 > 0$, то если рассмотреть последовательность шагов, в которой x_1 на один меньше, а x_2 на один больше, итоговая площадь увеличится на y_1 и окажется не меньше D. Аналогично для y_1 и y_2. Таким образом, если искомая площадь достижима за какое-то число шагов, то она достижима и за то же число шагов, но при условии, что ходят только второй и третий миротворец.

В таком случае формула для площади получается $D = \frac{x_2 y_2}{2}$, что максимально при фиксированной сумме $x_2 + y_2$, если $x_2 = y_2$. Получаем, что достаточно было взять число $x_2 = \lceil \sqrt{s} \rceil$ — округленный вверх квадратный корень из s. Соответствующий ему y_2 либо равен ему, либо на один меньше, это можно было проверить одним условием на их произведение. В ответ затем следовало вывести $x_2 + y_2$. Время работы решения — $\mathcal{O}(1)$.

Задача E. Джинкс и лагерь миротворцев

Автор задачи и разработчик: Константин Бац

Давайте отдельно решим задачу для горизонтальных и вертикальных подходов.

Решим задачу для подходов первого вида, то есть для прямоугольников (x, y_1), (x, y_2), при помощи *сканирующей прямой* (scanline). Будем идти по возрастанию x и поддерживать минимумы для всех точек на прямой $(x, 0)$, $(x, \max y)$. Таким образом, у нас будут операции трех видов:

1. Добавить прямую со значением v в точки $(\overline{y_1}, \overline{y_2})$ и пересчитать минимумы в точках;
2. Удалить прямую со значением v в точки $(\overline{y_1}, \overline{y_2})$ и пересчитать минимумы в точках;
3. Найти минимум на отрезке $(\overline{y_1}, \overline{y_2})$.

Первую операцию будем выполнять, когда при текущем x будет начинаться зона ответственности какого-то стража; вторую — когда при текущем x будет заканчиваться зона ответственности какого-то стража; а третью — когда нужно найти минимальную защищенность среди точек, в которые ведет подход.

Для быстрой обработки таких операций будем использовать дерево отрезков с поддержкой минимума и массовой операцией «установить минимум на отрезке». Тогда мы научимся обрабатывать запросы 1 и 3 за $\mathcal{O}(\log(\max y))$. Однако из-за необратимости функции минимум просто так обработать запрос второго вида не получится. Поэтому в каждом узле дерева отрезков требуется дополнительно хранить структуру данных, которая позволяет быстро добавлять в нее элементы по

некоторому ключу, удалять их и пересчитывать минимум среди всех элементов. Давайте в качестве ключа в такой структуре данных использовать пару из силы стража и номера стража. В качестве такой структуры можно использовать двоичное дерево поиска логарифмической глубины (например, декартово дерево). В такое дерево мы умеем быстро добавлять и удалять элементы, а самый левый элемент будет иметь минимальное значение, среди всех элементов.

Общий план решения задачи с подходами вида $(x, y_1), (x, y_2)$, соответственно, такой:

Создадим очередь из событий вида «добавить в рассмотрение вертикальный прямоугольник с некоторым значением», «посчитать минимум на отрезке», «удалить из рассмотрения вертикальный прямоугольник». Перед добавлением отсортируем события по x, а при равенстве — по типу операции (сперва мы хотим добавить новые прямоугольники, потом обработать запросы, потом удалить старые).

- При добавлении нового прямоугольника с некоторыми y_1 и y_2, разделим отрезок $[y_1, y_2]$ на $\mathcal{O}(\log(\max y))$ отрезков в дереве отрезков. В каждом узле, отвечающем за какой-то отрезок, обновим минимум, добавим пару из значения и номера прямоугольника в дерево поиска, возьмем значение минимального элемента дерева поиска и пересчитаем минимум. Это работает за $\mathcal{O}(\log^2(\max y))$.
- При удалении нового прямоугольника с некоторыми y_1 и y_2, также разделим отрезок на $\mathcal{O}(\log(\max y))$ отрезков в дереве отрезков, в каждом узле удалим соответствующий удаляемому прямоугольнику элемент из дерева поиска и пересчитаем минимум. Это тоже работает за $\mathcal{O}(\log^2(\max y))$.
- При обработке запроса на нахождение минимума найдем минимум на нужном отрезке при помощи дерева отрезков. Это работает за $\mathcal{O}(\log(\max y))$.

Аналогичную последовательность действий можно сделать и для подходов другого вида.

Максимальные координаты в задаче не превышают $2 \cdot 10^5$, поэтому $\mathcal{O}(\max x) = \mathcal{O}(\max y) = \mathcal{O}(\log n)$. За все время работы программы у нас будет $\mathcal{O}(n)$ запросов на изменение дерева отрезков и $\mathcal{O}(m)$ запросов поиск минимума в дереве отрезков. Значит общая сложность программы: $\mathcal{O}(n \log^2(n) + m \log(n))$.

Задача F. Поезда в Зауне

Автор задачи: Константин Бац, разработчик: Мария Жогова

Представим описанную в задаче модель в виде графа, в котором вершинами являются линии, а ребра обозначают факт их пересечения. В такой формулировке каждый поезд может оставаться на ночь в любом депо внутри своей компоненты связности.

Требование на возможность разместить все поезда на ночь можно записать как то, что суммарная вместимость всех депо в компоненте связности должна быть не меньше, чем суммарное количество поездов, которые курсируют по соответствующим линиям.

Будем «строить» и «расширять» депо по мере добавления новых линий, чтобы в каждый момент времени их вместимости хватало на все суммарно открытые линии. Действительно, заметим, что один из оптимальных ответов на момент после открытия новой линии может быть получен из предыдущего ответа либо открытием нового депо на новой линии, если она не пересекается ни с какой другой, либо увеличением вместимости одного из депо в ее компоненте связности, что позволит не создавать новое депо и минимизировать их количество. В конце, после открытия всех линий, мы получим такой набор депо, для которого в каждый момент времени все поезда со всех открытых линий можно разместить в них.

Итак, мы поняли, что минимальное количество депо в ответе равно количеству линий, которые при открытии не пересекаются ни с какими другими. Теперь просимулируем процесс открытия новых линий, поддерживая *систему непересекающихся множеств*, в которой для каждого множества (компоненты связности) будем хранить ее текущую суммарную вместимость и номер произвольного депо на ней. При создании новой линии объединяем все пересекающиеся с ней множества и расширяем это самое «помеченное» в множестве произвольное депо на количество поездов на новой линии.

Время работы такого алгоритма равно $\mathcal{O}\left((n + \sum c_i)\alpha(n)\right)$

Задача G. Взрывоопасная лестница

Автор задачи и разработчик: Арсений Кириллов

Так как необходимо получить минимальную лексикографически последовательность, нужно добиться того, чтобы самый первый элемент был минимально возможным. Значит, в самом низу должен стоять уровень, у которого первый элемент минимален. Аналогично, следующий элемент должен быть минимально возможным среди всех, для которых зафиксировано минимально возможное значени первого. Поэтому из всех уровней с минимальным первым элементом нужно выбрать уровень, у которого минимален второй элемент. Из оставшихся, нужно выбрать уровень с минимальным третьим элементом и так далее.

Этот процесс можно остановить, только если мы рассматриваем все уровни с минимальными первыми $i - 1$ элементами, и среди них есть уровень длины ровно $i - 1$, то есть у которого нет следующего, i-го элемента. Если поставить этот уровень в самый низ лестницы, первые $i-1$ элемент будут минимальны лексикографически, а следующий минимальный i-й элемент можно также будет выбирать среди всех уровней длины хотя бы i, так как расположив любой из них непосредственно над этим, мы бы добились падения соответствующего элемента вниз.

А тогда, если в какой-то момент оказалось, что мы ищем минимальный i-й элемент, и среди рассматриваемых рядов есть ряд длины ровно $i - 1$, следует расширить область поиска следующего минимального элемента до всех рядов длины хотя бы i. После этого необходимо запустить (возобновить) такой же процесс: рассмотреть уровни с минимальным i-м элементом, среди них — с минимальным $i + 1$-м элементом и так далее.

На выбор каждого уровня и отсеивание неподходящих вариантов можно тратить $\mathcal{O}(n)$ времени, что дает общее время решения $\mathcal{O}(n^2)$.

Задача H. Компонентная химия

Автор задачи: Евгений Карпович, разработчик: Константин Бац

Давайте формализуем задачу. У нас есть изначально пустой массив S. Каждый запрос добавляет в этот массив некоторое число a_i или удаляет одно вхождение a_i из него. После каждой такой операции требуется найти количество неупорядоченных пар $\{S_i, S_j\}$ $(i \neq j)$, что их сумма $S_i + S_j$ кратна m.

Заметим, что, рассматривая кратность суммы пар чисел числу m, можно рассматривать эти числа по модулю m. То есть, если $x = \overline{x} + z_x \cdot m$, $y = \overline{y} + z_y \cdot m$, $0 \leqslant \overline{x}, \overline{y} < m$, $z_x, z_y \in \mathbb{Z}$, то $(y + x) \bmod m = (\overline{x} + z_x \cdot m + \overline{y} + z_y \cdot m) \bmod m = (\overline{x} + \overline{y}) \bmod m$.

Давайте последовательно обрабатывать запросы, поддерживать массив `c` такой, что для всех k от нуля до $m - 1$ `c`$[k]$ равно текущему количеству элементов в массиве M с остатком k, и текущее количество пар, сумма которых кратна m.

1. Пусть требуется добавить в массив M элемент x, $\overline{x} = x \bmod m$. Тогда количество пар, удовлетворяющих условию, увеличится на `c`$[(m - \overline{x}) \bmod m]$. Прибавим это число к счетчику пар, выведем значение на экран и увеличим `c`$[\overline{x}]$ на один.

2. Пусть требуется удалить из массива M элемент x, $\overline{x} = x \bmod m$. Найдем $\overline{y} = (m - \overline{x}) \bmod m$ – остаток чисел, которые образуют с x пару. Может быть два случая:

   - Если $\overline{x} \neq \overline{y}$, то в `c`$[\overline{y}]$ посчитано число пар, которые распадутся при удалении x из m. Значит обновим количество пар с суммой, кратной m, выведем ответ и обновим $C[\overline{x}]$, то есть вычтем оттуда единицу.
   - Если $\overline{x} = \overline{y}$, то, так как $(\overline{x} + \overline{y}) \bmod m = 0$, в `c`$[\overline{y}]$ в том числе посчитано число x, которое, как известно, само с собой пару не образует. Значит число пар, уменьшится на `c`$[\overline{x}] - 1$. После этого нужно вывести ответ и уменьшить `c`$[\overline{x}]$ на 1.

Ввиду ограничений на m (до 10^9), поддерживать массив `c` целиком не получится. Однако вместо обычного массива можно использовать произвольный словарь, например `HashMap`, `unordered_map`,

`dict`. Тогда, поскольку число запросов ограниченно 10^5, то и размер словаря будет занимать $\mathcal{O}(10^5)$ памяти, а каждый запрос будет все еще обрабатываться за $\mathcal{O}(1)$.

Замечание 1. Стоит учитывать, что некоторые языки программирования, например C++ и Java, считают остаток от деления отрицательных чисел не так, как это принято в математике. Напомним, остаток от деления числа a на b — это такое $0 \leqslant c < b$, что $a = b \cdot z + c$ и $z \in \mathbb{Z}$.

Замечание 2. На самом деле, при удалении числа из массива M можно избежать рассмотрения случаев, если сперва уменьшить $\mathtt{c}[\overline{x}]$, а потом пересчитать счетчик пар.

Итого, время работы программы: $\mathcal{O}(n)$.

Задача I. Палиндромная шифровка

Автор идеи и разработчик: Владимир Рябчун

Основная идея задачи состоит в том, что есть всего две принципиально разные ситуации при разрезании палиндрома. Пусть $p = ts$, тогда в первом случае $|t| \leqslant |s|$. Тогда t является префиксом развернутой строки s: $\overline{s} = tl$, где l — тоже палиндром. Во втором случае $\overline{s}$ — префикс t, а оставшийся суффикс t — палиндром.

Тогда построим бор на развернутых строках s и для каждой вершины бора определим, существует ли из нее путь до терминальной вершины, являющийся палиндромом. Это можно сделать при помощи алгоритма Манакера или хешей — просто для всех суффиксов $\overline{s_i}$ определим, являются ли они палиндромами и в соответствующих вершинах бора сделаем пометки. Для удобства будем считать, что пустая строка — палиндром.

Теперь, чтобы обработать запрос t_i нужно спускаться по символам t_i в построенном боре и проверять, является ли текущий префикс t_i какой-то строкой $\overline{s_j}$ плюс остаток-палиндром (что проверяется так же алгоритмом Манакера или хешами). В конце мы могли все еще остаться в боре, что означает, что существует одна или более строка $\overline{s_j}$ такая, что t_i — её префикс. Тогда надо проверить, что из этого состояния бора можно дойти до терминальной вершины и получить палиндром, а эта величина уже посчитана на этапе построения.

Итоговая сложность — $\mathcal{O}\left(\sum\limits_{i=1}^{n} |s_i| + \sum\limits_{i=1}^{m} |t_i|\right)$

Задача J. Магические часы

Авторы задачи: Григорий Хлытин и Даниил Орешников, разработчик: Даниил Орешников

Представим каждую пару делений (минутная и часовая стрелки) на часах как вершину графа. Тогда для каждой вершины можно посчитать следующее состояние, в которое переходят часы после одного тика. Затем для каждого запроса можно было бы запустить симуляцию процесса, чтобы определить, достигнется ли желаемое состояние, но такое решение работало бы слишком долго.

Заметим, что если при движении между двумя состояниями минутная стрелка хотя бы раз догоняла часовую, то существовал момент времени между двумя состояниями, когда минутная стрелка была на нуле. Обозначим за (h_1, m_1) стартовое состояние, (h_2, m_2) — конечное состояние в запросе. Тогда можно отдельно проверить, достижимо ли второе состояние из первого, без прохода минутной стрелки через ноль, а дальше отталкиваться от того, что хотя бы раз минутная стрелка перескакивала в ноль. Для проверки первого случая достаточно заметить, что часовая стрелка движется с постоянной скоростью, и проходит строго меньше целого круга, поэтому через $h_2 - h_1$ можно вычислить сколько в точности прошло времени и проверить, перешло ли m_1 в m_2 за это время.

Случай прохождения через 0 же рассмотрим так: по (h_2, m_2) однозначно восстанавливается, из какого состояния $(h_2^*, 0)$ и сколько времени назад начинала двигаться минутная стрелка (для этого должно выполняться, что $m_2 \bmod 12 = 0$), и, аналогично, сколько времени пройдет перед тем, как состояние (h_1, m_1) перейдет в $(h_1^*, 0)$, и каким будет это h_1. Это произойдет за $\left\lceil \frac{(h_1 - m_1) \bmod (60 \cdot 12 \cdot t)}{11} \right\rceil$ тиков.

После этого достаточно проверить, достижимо ли состояние $(h_2^*, 0)$ из $(h_1^*, 0)$, и если да, то за сколько тиков. Каждому такому состоянию $(h, 0)$ сопоставим вершину графа, из каждой вершины будет вести одно ребро вперед — какое следующее состояние такого вида будет получено из данного

(вес ребра будет равен количеству тиков). В общем случае минутная стрелка догонит часовую за $\left\lceil \frac{h}{11} \right\rceil$ тиков. Только надо не забыть особый случай $h = 0$, при котором пройдет $60t$ тиков перед достижением минутной стрелкой нуля.

Граф, в котором из каждой вершины выходит ровно одно ребро, может быть представлен как набор циклов, и путей, начинающихся в каких-то других вершинах, и заканчивающихся в вершинах этих циклов. С помощью `dfs` выделим все пути и циклы. Для каждого цикла предподсчитаем префиксные суммы весов ребер (начиная с произвольной его вершины) и позицию вершины в цикле, для каждого пути — сумму весов ребер на пути от каждой вершины до первой вершины цикла, в который этот путь ведет, а так же количество ребер до первой вершины цикла, и в какую конкретно вершину цикла этот путь приводит.

Теперь, чтобы ответить на запрос, достаточно проверить, что вершина h_2^* достижима из h_1^*. Для этого либо они обе должны принадлежать одному циклу, тогда расстояние можно найти префиксными суммами, либо одна h_2^* должна лежать на цикле, а h_1^* — на пути, ведущем к нему, тогда ответ сложится из суммы расстояния до цикла и расстояния в цикле. Остается случай, когда две вершины лежат на одном пути, однако, поскольку пути могут соединяться и образовывать дерево, стоит так же заранее сжать все циклы в вершины, и на получившихся деревьях обходом `dfs` посчитать время входа и время выхода для каждой вершины, с помощью которых затем за $\mathcal{O}(1)$ можно определять отношение достижимости на путях в исходном графе. Если h_2^* достижима из h_1^*, расстояние получится как разность расстояний от них до цикла.

В конце надо не забыть прибавить количество тиков для достижения h_1^* из (h_1, m_1) и для достижения (h_2, m_2) из h_2^*. Общее время работы решения — $\mathcal{O}(720t + q)$.