Задача A. Побег Майлза

Автор задачи: Александр Гордеев, разработчик: Егор Юлин

Будем воспринимать небоскребы как вершины графа, а возможные перемещения между ними — как ребра. В задаче интуитивно хочется построить граф на n вершинах, между которыми есть ребра двух типов: ребра из первого мира и из второго. Но тогда становится сложно учитывать, что для смены типа ребра нам также нужно потратить x секунд для перемещения между мирами.

Стандартный подход в задачах такого рода — завести больше вершин в графе. Каждая вершина нового графа будет отвечать за пару $(v, \texttt{state})$, где v — вершина исходного графа, а `state` — «состояние» путешествующего по графу. В данном случае состояние будет задавать то, в каком мире Майлз находится, поэтому нам понадобится $2n$ вершин: n соответствующих небоскребам первого мира, n — второго. Далее в качестве индекса у вершины будем указывать номер мира, которому она соответствует.

Теперь осталось построить ребра в таком графе. Для этого просто проведем по ребру между любыми двумя небоскребами, между которыми возможно перемещение:

- если в первом мире можно было переместиться из u в v за время c, проведем в новом графе ребро $u_1 \to v_1$ с тем же весом c;
- аналогично для второго мира;
- и проведем ребра $v_1 \to v_2$ и $v_2 \to v_1$ веса x для всех $v \in [1, n]$ — эти ребра соответствуют перемещению между мирами.

Теперь любому пути, удовлетворяющему условию, соответствует путь в нашем графе, и наоборот. Поэтому нам остается только найти кратчайший путь из вершины s_1 (небоскреб s в первом мире) в вершину t_2 (небоскреб t во втором мире). Поскольку все веса в графе неотрицательны, для этого можно воспользоваться алгоритмом Дейкстры, и получить решение за время $\mathcal{O}((m_1 + m_2 + n) \log n)$.

Задача B. Ловля пауков

Автор задачи и разработчик: Егор Юлин

Поймем, что означает условие про возможность распределить корм поровну вне зависимости от количества пауков. Если для любого t от 1 до k есть возможность распределить $m-1$ порций корма на t пауков поровну, то $m-1$ делится на t для любого t от 1 до k.

Следовательно, если мы выбрали какое-то количество корма m, то для каждого t от1 до k для выбранного m, $m-1$ должно делится на t (так как один корм мы отдали на анализ, а оставшееся количество должны поровну распределить между пауками). И теперь наша задача сводится к поиску такого m, что

1. $m \leqslant n$;
2. $m-1$ делится на все числа от 1 до k.

Давайте вместо этого искать подходящий $m' = m - 1$. Поскольку он делится на все числа от 1 до k, то он делится и на $\texttt{lcm}(1, 2, \ldots, k)$, где `lcm` означает наименьшее общее частное. Известный факт: наименьшее общее частное выражается через наибольший общий делитель, который можно посчитать алгоритмом Евклида, следующим образом: $\texttt{lcm}(a, b) = \frac{a \cdot b}{\gcd(a,b)}$. А тогда вычислить $\texttt{lcm}(1, 2, \ldots, k)$ можно, например, за время $\mathcal{O}(k \log k)$, так:

```
result = 1
for i = 2..k {
    result = result * i / gcd(result, i)
}
```

Теперь имеет смысл расмотреть два случая (на самом деле не обязательно рассматривать эти случаи явно, можно было просто считать `lcm`, пока оно не превысит $n-1$, но мы в явном виде укажем границу на k, при которой это происходит):

1. $k \geqslant 43$, а тогда $\texttt{lcm}(1, 2, \ldots, k) > 10^{18}$, и единственное подходящее нам m' — это 0, так как $m' \leqslant n - 1 < 10^{18}$;

2. $k < 43$, тогда можно «по-честному» с помощью последовательного применения алгоритма Евклида вычислить интересующий нас $\texttt{lcm}(1, 2, \ldots, k)$ и взять максимальное число, не превосходящее $n - 1$, которое на него делится — это будет

$$m' = (n - 1) - ((n - 1) \bmod \texttt{lcm}(1, 2, \ldots, k)).$$

Таким образом, решение работает в общем случае за $\mathcal{O}(k \log k)$.

Задача C. Прыжки между вселенными

Автор задачи: Даниил Орешников, разработчик: Константин Бац

Для решения этой задачи можно воспользоваться системой непересекающихся множеств (`disjoint set union`, `dsu`). Будем считать элементами внутри этой системы — миры, а множествами — наборы миров, взаимнодостижимых друг из друга при некотором текущем значении энергетического уровня w_{cur}. Это напрямую следует из того, что если в какой-то момент мир становится достижимым, то после мы всегда сможем попасть в этот мир независимо от наших следующих действий, так как энергетический уровень часов не уменьшается. А тогда «достижимая область» всегда представляет из себя некоторый связный подграф исходного графа.

Тогда все, что надо делать — реализовывать «расширение» нашей достижимой области при увеличении энергетического уровня часов, а также увеличивать этот уровень при возможности посетить новые миры. Давайте внутри СНМ хранить множества взаимодостижимых по ребрам веса не больше w_{cur} миров. Помимо этого для каждого мира запомним его хранить характеристику a_i и в множестве будем поддерживать сумму всех a_i по элементам множества (достаточно хранить ее отдельным полем в представителе множества).

Заметим, что текущее значение энергетического уровня всегда равно сумме a_i в множестве, в котором содержится мир с номером s, потому что в соответствующем множестве перед тем, как двигаться за его пределы, мы можем сначала посетить все его вершины и по максимуму увеличить энергетический уровень. Тогда w_{cur} можно найти в СНМ за время $\mathcal{O}(\alpha(n))$, что можно считать константой. Теперь посмотрим, как достижимых миров меняется:

1. Отсортируем порталы по возрастанию w_i.

2. Рассматривая очередной портал (u, v), сравним его w_i с текущим w_{cur}, и если $w_i \leqslant w_{\text{cur}}$, то объединим множества, в которых содержатся миры u и v. При этом не забываем обновить сумму a_i в корне объединения.

3. Если же $w_i > w_{\text{cur}}$, то мы этим порталом воспользоваться не сможем, да и всеми последующими порталами тоже (так как они отсортированы по w_i) — в этот момент можно завершить алгоритм.

Таким образом, значение w_{cur} после рассмотрения всех порталов равно значению энергетического уровня часов, то есть ответу на задачу. В таком решении на обработку каждого портала потребуется $\mathcal{O}(\alpha(n))$ времени, поэтому время работы такого решения — $\mathcal{O}(n + m\alpha(n))$.

Задача D. Тест на интеллект

Автор задачи и разработчик: Даниил Голов

Переформулируем условие без легенды. Необходимо составить из букв латинского алфавита строку длины n. Далее использованной в строке букве алфавита будет сопоставлена стоимость от 1 до количества уникальных букв так, чтобы суммарная стоимость всех букв строки была минимальна. Нужно найти строку максимальной возможной итоговой стоимости.

Для начала докажем, что для этого необходимо, чтобы количество вхождений любых двух букв в строку отличалось не более чем на 1. Обозначим за $\texttt{count}(c)$ количество вхождений буквы c в строку. Докажем от противного — пусть есть такие две буквы x и y, что $\texttt{count}(x) - \texttt{count}(y) \geqslant 2$. Тогда при сопоставлении стоимостей букве y будет назначена стоимость q, букве x — стоимость p, и $p < q$, так как если $p > q$, то

- буквы x и y дают вклад $\texttt{count}(x) \cdot p + \texttt{count}(y) \cdot q$ в стоимость строки;
- если стоимости этих букв поменять местами, то получится меньшее значение $\texttt{count}(x) \cdot q + \texttt{count}(y) \cdot q$.

А в таком случае, если заменить одну букву x на букву y, то стоимость, которую бы две эти буквы принесли в сумму, была бы $(count(x) - 1) \cdot p + (count(y) + 1) \cdot q$, что на $q - p$ больше. Значит, такая строка не является максимальной по стоимости, что и требовалось доказать.

Более того, нам «выгодно» брать больше различных букв. Действительно, если мы рассмотрим строку, в которой $a < 26$ различных букв, и заменим одну самую частую из них на еще неиспользованную, мы получим большую суммарную стоимость (как следует из предыдущего утверждения, машина назначает меньшие веса более часто встречающимся буквам).

То есть, нам нужно как можно больше разных букв, и при этом как можно более равные количества различных букв (отличающиеся не более чем на 1). Тогда для ответа, например, подойдет строка «`abcdef...zabcdef...zabc...`» длины n. Если $n \leqslant 26$, можно вывести просто первые n букв латинского алфавита. Решение чисто конструктивное, работает за время $\mathcal{O}(n)$.

Задача E. Прямоугольное Пятно

Автор задачи: Даниил Орешников, разработчик: Константин Бац

Подойдем к задаче с другой стороны. Сначала рассмотрим версию задачи, в которой прямоугольники нельзя поворачивать. Тогда для любых двух идущих подряд выбранных прямоугольников i и j должно выполняться, что $w_i \leqslant w_j$ и $h_i \leqslant h_j$. Сфокусируемся на первом условии и сделаем так, чтобы оно выполнялось автоматически: отсортируем все прямоугольники по возрастанию h_i (а при равенстве — по возрастанию w_i).

Теперь задача свелась к тому, чтобы выбрать такую подпоследовательность прямоугольников, в которой для любых двух подряд идущих i и j выполняется $h_i \leqslant h_j$. А это уже стандартная задача о наибольшей возрастающей последовательности (НВП), которая решается за время $\mathcal{O}(n \log n)$. Остается решить проблему того, что мы не учли возможность поворачивать прямоугольники на 90°.

Заметим, что все прямоугольники, не являющиеся квадратами, можно скопировать, поменяв местами ширину и высоту, и это не повлияет на ответ задачи. Действительно, поворачивать квадраты нет смысла — они не изменятся, а если у нас есть и прямоугольник (h_i, w_i), и прямоугольник (w_i, h_i), то максимум один из них войдет в ответ — при $h_i \neq w_i$ ни один из них нельзя вложить в другой.

Тогда выполним описанное выше действие и отсортируем прямоугольники, например, по ширине. Тогда задача сводится к поиску наибольшей возрастающей последовательности высот прямоугольников, как мы уже отметили ранее.

Один из стандартных способов нахождения НВП за время $\mathcal{O}(n \log n)$ с восстановлением ответа — это динамическое программирование с двоичным поиском. Будем строить следующую динамику: для $0 \leqslant i \leqslant n$ пусть $\texttt{d}[i]$ — наименьшее число, на которое может оканчиваться возрастающая последовательность длины i.

Изначально мы предполагаем, что $\texttt{d}[0] = -\infty$, а все остальные элементы $\texttt{d}[i] = +\infty$ — после последовательности длины 0 можно поставить что угодно, а вот последовательности положительных длин мы пока не обнаружили. И теперь будем по очереди обрабатывать элементы массива, обновляя эти значения. Заметим два важных свойства:

- $d[i-1] \leqslant d[i]$ для всех $1 \leqslant i \leqslant n$;
- каждый элемент h_i обновляет максимум один элемент массива `d` — после элементов, больших h_i, поставить h_i нельзя, а если можно поставить h_i на k-е место, то уже существовала последовательность длины $k-1$, заканчивающаяся на что-то, меньшее h_i.

Это означает, что при обработке очередного h_i, мы можем за $\mathcal{O}(\log n)$ с помощью двоичного поиска в массиве `d` найти первое число, которое больше либо равно текущего h_i, и обновить его. Отдельно стоит уделить внимание восстановлению ответа, то есть последовательности прямоугольников, которые необходимо использовать — для этого достаточно просто в момент обновления `d` запоминать для нового элемента стоящий перед ним. Время работы — $\mathcal{O}(n \log n)$.

Задача F. Гвен отдыхает

Автор задачи: Владислав Власов, разработчик: Егор Юлин

Для определенности будем считать, что на каждом шаге $n > m$ *(Если это не так, то мы можем просто поменять стороны прямоугольника, от этого исход игры не изменится).*

Заметим, что перед началом хода Гвен у нас может быть три случая:

1. n и m отличаются более чем на 1, т.е. их абсолютная разница превосходит 1 или $|n - m| > 1$. Тогда Гвен в своем ходу будет оптимальнее увеличить сторону m, после чего она получит n очков. Но после этого компьютеру тоже будет оптимально увеличить сторону m, после чего он так же получит n очков. Не сложно заметить, что после такого хода разница в очках между Гвен и компьютером не изменилась, но стороны прямоугольника станут равны $m + 2$ и n.

2. n и m отличаются ровно на 1 или $|n - w| = 1$. Тогда в свой ход Гвен будет оптимальнее увеличить сторону m, после чего она получит n очков. После этого $m + 1 = n$ и при любой выбранной стороне компьютер получит n очков. В таком случае разница в очках также не изменится, а стороны прямоугольника станут равны $(m + 1, n + 1)$ или $(m + 2, n)$. Не трудно заметить, что стороны прямоугольника все еще отличаются ровно на 1.

3. $n = m$. Тогда в свой ход Гвен получит n очков. А после этого для компьютера оптимальным ходом будет увеличение меньшей стороны, после чего он получит $n+1$ очков. Разница в очках при этом увеличится на 1 в сторону компьютера, а стороны прямоугольника после этого станут равны $n + 1$ и $m + 1$.

Иными словами, всегда выгодно увеличивать меньшую сторону прямоугольника, чтобы получить количество очков, равное большей стороне. Понятно, почему в любой конкретный момент такой ход выгоднее, чем увеличение большей стороны, остается только доказать, что такой ход действительно выгоднее в долгой игре. Для этого рассмотрим любую игру и последний момент, в который кто-то из игроков решил увеличить большую из двух сторон прямоугольника.

1. После момента, когда стороны прямоугольника сравняются, игра будет идти так же.

2. А между этими моментами были ходы вида «увеличение меньшей стороны», которых у нас останется столько же, а у противника либо останется столько же, либо станет на один больше. Иными словами, если аккуратно расписать последовательность получаемых каждым игроком очков, разность между очками противника и нашими не уменьшится.

Теперь для полного решения данной задачи можно заметить, что пока разница n и m превосходит 1, на игру это никак влиять не будет, так как игроки будут получать поровну очков. Поэтому мы увеличим m до n или $n - 1$ в зависимости от четности, на это будет потрачено $\left\lfloor \frac{n-m}{2} \right\rfloor$ ходов. Если ходы после этого закончились, то будет ничья. Если же у нас после этого остались ходы, то если $n - m = 1$, то снова будет ничья, а если $h = m$, то победит компьютер, при этом разница в очках будет равна количеству оставшихся ходов.

Задача G. Человек-паук Нуар и кубик Рубика

Авторы задачи: Даниил Орешников и Егор Юлин, разработчик: Даниил Орешников

Задачи о поиске пути в матрице из левого-верхнего в правый нижний угол часто оказываются либо на динамическое программирование, либо на потоки. В случае, когда еще и ходить разрешается только вправо или вниз, это с очень большой вероятностью динамика.

Перед тем, как описать, какие состояния в данной динамике понадобятся, отметим несколько ключевых идей.

1. Если есть возможность попасть в клетку (i, j), и мы знаем, был ли инвертирован столбец j, то мы однозначно восстанавливаем, была ли инвертирована строка j. Действительно, только эти две инверсии могли менять значение клетки (i, j), а в конце ее значение должно было стать равно 1, что задает черный цвет клетки.

2. Соответственно, обратное тоже верно — по факту, инвертировался ли столбец, мы однозначно восстанавливаем, была ли инвертирована строка, если клетка лежит на пути.

3. В клетку (i, j) можно попасть только из $(i-1, j)$ или $(i, j-1)$.

4. Если в клетку (i, j) мы пришли из $(i-1, j)$, то, зная, была ли инвертирована строка i, мы можем восстановить, была ли инвертирована строка $i-1$ по цепочке $\texttt{inv\_row}[i] \to \texttt{inv\_column}[j] \to \texttt{inv\_row}[i-1]$.

5. Аналогично для столбцов при переходе из $(i, j-1)$ в (i, j).

Теперь, когда мы понимаем, что достаточно знать только факт инвертирования столбца или строки какой-то клетки, чтобы восстановить такую же информацию при переходах, можно посчитать динамику: $\texttt{dp}[i][j][c]$ — это минимальное количество инверсий, которое надо сделать, чтобы существовал путь из левого верхнего угла таблицы в клетку (i, j), **при условии** c ($c = 0$ означает, что столбец j не был инвертирован, $c = 1$ — был).

Посмотрим, какие переходы возможны и через какие состояния можно обновить $\texttt{dp}[i][j][c]$.

1. Мы могли прийти в (i, j) из $(i-1, j)$. Поскольку они находятся в одном и том же столбце, то флаг его инвертирования не мог поменяться и равен c. Тогда флаг инвертирования строки i равен $r = \texttt{a}[i][j] \oplus c \oplus 1$, где $\oplus$ — операция логического «исключающего ИЛИ» (`xor`). Поскольку это новая строка, то надо учесть ее инвертирование:

$$\texttt{dp}[i][j][c] = \min(\texttt{dp}[i][j][c], \texttt{dp}[i-1][j][c] + r.$$

2. Мы могли прийти в (i, j) из $(i, j-1)$. Тогда флаг инвертирования строки i равен $r = \texttt{a}[i][j] \oplus c \oplus 1$, а флаг инвертирования столбца $j-1$ можно посчитать как $c_{j-1} = \texttt{a}[i][j-1] \oplus r \oplus 1$. Обновляем аналогично:

$$\texttt{dp}[i][j][c] = \min(\texttt{dp}[i][j][c], \texttt{dp}[i][j-1][c_{j-1}] + c.$$

Теперь, когда мы посчитали минимальное количество инверсий $\min(\texttt{dp}[n][m][0], \texttt{dp}[n][m][1])$, достаточно с помощью стандартных техник восстановления ответа «вернуться» по оптимальному пути в клетку $(1, 1)$ и выписать по пути все инвертированные строки и столбцы, после чего вывести. Проще всего по ходу подсчета динамики также запоминать, из какого состояния было самое оптимальное обновление, и для восстановления ответа возвращаться по этим состояниям назад. Общее время работы алгоритма — $\mathcal{O}(nm)$.

Задача H. Квантовая дыра

Автор задачи и разработчик: Владислав Власов

Стандартная идея в задачах, в которых надо «собрать» строку из частей — превратить строку в путь в графе. А именно, создадим граф на 2^k вершинах, соответствующих всем двоичным строкам длины k (идея похожа на граф Де-Брёйна, который часто используется в биоинформатике). После чего соединим ребрами строку s_1 и s_2, если s_2 можно получить из s_1, откинув первый символ и дописав новый в конец. Например, при $k = 4$ строки «0010» и «0101» будут соединены ребром, так как имеют общий суффикс-префикс длины $k-1$ «010».

Заметим, что теперь любая двоичная строка длины $n \geqslant k$ представляется как путь в этом графе. Отдельно рассмотрим случай $n < k$: в таком случае опасность строки всегда будет равна 0, и можно просто вывести любую битовую строку длины n. А при $n \geqslant k$ строка t может быть представлена как путь по вершинам $t_{1,\ldots,k} \to t_{2,\ldots,k+1} \to \ldots \to t_{n-k+1,\ldots,n}$. При чем каждая встреченная на пути вершина — это подстрока длины k данной строки t, поэтому опасность строки — это просто сумма весов вершин на соответствующем ей пути.

Теперь наша задача — найти самый «дешевый» путь длины $n-k+1$. Воспользуемся методом динамического программирования, пусть $\texttt{dp}[i][j]$ — минимальная опасность пути длины i, который заканчивается в вершине j. Такую динамику несложно посчитать и восстановить ответ: $\texttt{dp}[i+1][v]$ надо обновить через $\texttt{dp}[i][u] + d_v$ по всем ребрам $u \to v$.

Альтернативно можно было заметить, что этот алгоритм очень похож на стадартный алгоритм поиска кратчайшего пути Форда-Беллмана. Однако он ищет кратчайший путь в графе со взвешенными ребрами, а у нас взвешенные вершины. Одно можно свести к другому следующим образом: построим граф на 2^{k-1} вершин вместо 2^k, в котором вершины соответствуют битовым строкам длины $k-1$. После проведем ребро между двумя вершинами, если у них есть общий суффикс-префикс длины $k-2$, и при наложении они образуют строку длины 2^k. Тогда ребру назначим вес соответствующей битовой строки длины k.

Асимптотика времени работы такого решения — $\mathcal{O}(n \cdot 2^k)$

Задача I. Стабилизация мультивселенной

Автор задачи и разработчик: Даниил Орешников

Эта задача — один из примеров не самых стандартных сведений ответа к решению 2-SAT. Догадаться до этого можно было по следующему факту: перед нами стоит множество выборов из двух опций, при этом из каждой пары опций ровно одна должна быть выбрана.

Для начала упростим условие: дан ориентированный граф на n вершинах, в которой из каждой вершины выходит ровно два ребра, и в каждую вершину входит ровно два ребра, то есть $\mathtt{deg}_{\text{in}}(v) = \mathtt{deg}_{\text{out}}(v) = 2$ для всех v. Требуется выбрать несколько циклов, покрывающих все вершины по одому разу, при чем чтобы интервалы чисел на выбранных ребрах имели общее непустое пересечение.

Сразу будем описывать сведение задачи к 2-SAT, то есть к решению логической формулы в 2-КНФ, в которой несколько *клозов* (скобок, состоящих из логического «ИЛИ» двух переменных или их отрицаний) перечислены через логическое «И». Для этого поймем, как описать в терминах 2-SAT условия задачи. Вместо «ИЛИ» будем использовать следствия (импликации), так как их можно взаимно выразить друг из друга, а решение задачи 2-SAT строится как раз на импликациях.

1. Заведем по переменной на каждое ребро графа. Переменная e_i будет означать, выбрано ли i-е ребро. Тогда:
   - Из каждой вершины должно выходить только одно выбранное ребро, то есть $e_{2i+1} \to \neg e_{2i+2}$ и, наоборот, $e_{2i+2} \to \neg e_{2i}$.
   - Аналогично, в каждую вершину должно входить ровно одно выбранное ребро, то есть если в вершину v входят ребра i и j, надо добавить условия $e_i \to \neg e_j$ и $e_j \to \neg e_i$.
2. Заведем также по переменной m_c на выражения вида «выбранная мощность m не превосходит c». Они связываются следующей логикой:
   - Из того, что $m \leqslant c$, следует, что $m \leqslant c+1$, поэтому надо добавить следствие $m_c \to m_{c+1}$ для всех c от 0 до 10^5.
   - Из того, что мы выбрали ребро i, следует, что $m \in [a_i, b_i]$, то есть $\leqslant b_i$ и $\not\leqslant a_i - 1$. Добавляем следствия $e_i \to m_{b_i}$ и $e_i \to \neg m_{a_i-1}$.

Заметим, что построенная формула полностью описывает все условия из задачи. И любое решение этой формулы однозначно транслируется в корректный ответ для нашей задачи. Только надо не забывать, что когда мы добавляем следствие $a \to b$, надо также добавить и равнозначное ему $\neg b \to \neg a$.

Теперь, для решения этой формулы, применим стандартный алгоритм решения 2-SAT, основанный на выделении компонет сильной связности. Если мы получили противоречие, то решения нет, а иначе — выбираем те e_i, которые истинны, и получаем список выбранных ребер, а также выбираем максимальное c, при котором m_c истинно, и получаем значение m. Суммарное время работы алгоритма равно $\mathcal{O}(n + m_{\max})$.

Задача J. Паутина во все стороны

Автор задачи: Даниил Орешников, разработчик: Константин Бац

Давайте заметим, что весь «свет» (паутина), исходящий из начала координат, разбивается на сектора, и каждый из секторов либо достигает внешней окружности, либо поглощается по пути. Для того, чтобы вычислить ответ на задачу, достаточно найти сумму углов всех секторов, которые достигают внешней окружности, и поделить на 2π ($360°$). А секторов в принципе не так много: каждое препятствие (дыра) задает определенный сектор, который им поглощается — он расположен между касательными к этому препятствию из начала координат. Соответственно, поглощаемых секторов не более n, а значит всего секторов не более $2n$.

Осталось научиться определять эти сектора. Сначала рассмотрим простую версию: пусть по направлению нулевого угла (ось Ox) нет препятствий. Тогда воспользуемся стандартной техникой 1D-сканирующей прямой или «сортировки событий» и заведем события вида «в направлении угла α располагается касательная к препятствию». Каждое событие будет характеризоваться этим углом α и флагом $+1$, если центр препятствия имеет больший полярный угол, и -1 иначе. Теперь отсортируем эти события и обработаем в порядке возрастания угла. Получится некоторый «сканер» в виде луча из начала координат, который поворачивается против часовой стрелки.

Заметим, что когда мы встречаем событие с флагом $+1$, это означает, что после добавится новое препятствие, и наоборот — при обработке события -1 препятствий на пути луча становится на 1 меньше. Будем поддерживать количество препятствий по направлению луча c, и при встрече нового события обновлять его как $c \leftarrow c + \texttt{flag}$. Теперь осталось только посчитать ответ — суммарный угол всех секторов, в которых c было равно 0, то есть на пути луча не было препятствий. Для этого:

1. если после очередного события c стало равно 0, запоминаем текущий угол как начало очередного сектора `from`;
2. если после очередного события c перестает быть равно 0, добавляем к ответу текущий $\alpha - \texttt{from}$.

Теперь стоит упомянуть две детали. Во-первых, при сортировке векторов по углу полезно использовать векторное произведение, чтобы не вычислять углы напрямую — это поможет избежать ошибок из-за погрешности (хотя в данной задаче касательные к препятствиям уже имели нецелые координаты, так что конкретно в этой задаче в этом не было большого смысла). Во-вторых, мы решили задачу, исходя из того, что в нулевом направлении нет препятствий. Если это не так, решение не сильно изменится: либо можно отдельно посчитать исходный c как количество препятствий на пути за $\mathcal{O}(n)$, отдельно проверив каждое препятствие, либо можно считать «свободными» все сектора, в которых c достигает своего глобально минимального значения.

Время работы решения — $\mathcal{O}(n \log n)$ на сортировку событий.

Задача K. Иерархия Паучьего сообщества

Автор задачи: Даниил Голов, разработчики: Даниил Голов и Даниил Орешников

Переформулируем задачу чуть более коротко: дано подвешенное дерево, и требуется «пометить» (назначить мнение `A`) некоторое количество вершин, чтобы количество ребер, на которых верхняя вершина помечена, а нижняя — нет, было максимально.

Эта задача всем в своем условии намекает на динамическое программирование по поддеревьям, более того, она очень похожа на задачу о максимальном независимом множестве на дереве, которая также решается с помощью динамики. В этой задаче мы заведем такие же состояния:

- $\texttt{dp}[v][0]$ — максимальное значение беспорядка в поддереве вершины v, если при этом сама вершина v не помечена (имеет мнение `B`);
- $\texttt{dp}[v][1]$ — макисмальное значение беспорядка в поддереве вершины v, если сама v при этом помечена.

Для начала найдем максимальное возможное значение беспорядка, которое мы можем получить, а затем восстановим, какие вершины для этого надо было пометить. Для этого считаем все связи из ввода и сделаем обход в глубину из вершины номер 1 — так мы сможем определить для каждого ребра, какая вершина является родителем («ментором»), а какая — ребенком («подчиненным»).

Теперь (можно и на выходе из `dfs`) будем считать значения динамики. Заметим, что если мы зафиксируем, помечена ли вершина, дальше можно оптимизировать поддеревья ее детей независимо: действительно, от изменений в одном поддереве значения беспорядка в других не изменятся. Рассмотрим оба варианта: и когда v не помечена, и когда помечена.

1. Если вершина v не помечена, то независимо от состояний ее детей u_i, ни одно ребро (v, u_i) не будет вносить вклад в ответ. Таким образом, ответ будет просто складываться из максимальных возможных беспорядков в поддеревьях u_i, то есть
$$\texttt{dp}[v][0] = \sum_{u_i \text{ – ребенок } v} \max(\texttt{dp}[u_i][0], \texttt{dp}[u_i][1]).$$
2. Если вершина v помечена, то некоторые из ее детей будут образовывать с ней пару, дающую вклад $+1$ в ответ. А именно, непомеченные дети. Тогда помеченный ребенок u_i дает вклад $\texttt{dp}[u_i][1]$, а непомеченный — $\texttt{dp}[u_i][0] + 1$. Остается только выбрать максимум:
$$\texttt{dp}[v][1] = \sum_{u_i \text{ – ребенок } v} \max(\texttt{dp}[u_i][0] + 1, \texttt{dp}[u_i][1]).$$

Здесь хороший момент, чтобы вспомнить, что нас еще просят восстановить, какие вершины должны быть помечены, а какие — нет. Воспользуемся стандартной техникой для восстановления ответа: запомним, какой выбор привел нас к максимальному значению. А именно, заведем $\texttt{down}[v][0]$ и $\texttt{down}[v][1]$, которые будут содержать номера всех детей v, которые должны быть помечены, чтобы $\texttt{dp}[v][0]$ и $\texttt{dp}[v][1]$, соответственно, были максимальны.

Для их заполнения в формулах пересчета `dp` вместо того, чтобы просто брать максимум из двух опций, посмотрим, какая из них больше, и отметим соответствующую информацию в `down`. Так, если $\texttt{dp}[u_i][0] + 1 < \texttt{dp}[u_i][1]$, добавим u_i в $\texttt{down}[v][1]$, так как его выгоднее пометить, чем не помечать.

После того, как все вершины будут обработаны, максимальное значение беспорядка можно получить как $\max(\texttt{dp}[1][0], \texttt{dp}[1][1])$. Более того, зная, какая из двух этих величин больше, мы знаем, стоит ли отмечать вершину 1, чтобы достичь такого значения беспорядка. Запомним эту информацию, и спустимся по дереву: если очередной u_i лежит в соответствующем выбранному для первой вершины состоянию $\texttt{down}[1]$, то u_i тоже надо пометить, иначе — не надо. Так продолжаем спускаться по дереву, каждый раз восстанавливая состояния вершин.

Суммарно на подсчет такой динамики и спуск по дереву мы потратили время, пропорциональное количеству ребер в графе (по каждому ребру мы прошли константное число раз), поэтому весь этот алгоритм работает за $\mathcal{O}(n)$

Задача L. Взломать коллайдер

Автор задачи: Егор Юлин, разработчик: Константин Бац

Это довольно классическая задача на двоичный поиск. Как видно в этой задаче, двоичный поиск можно применять не только в массиве отсортированных в каком-либо порядке объектов, но и по произвольным другим монотонным критериям.

Наша цель — найти величину «сдвига» c, с которым нам сообщается информация. Если внимательно посмотреть на $f(x)$ и мысленно выписать $f(1), f(2), \ldots, f(n)$, можно заметить, что получится осортированный массив a_i, циклически сдвинутый на величину c, то есть $[a_{c+1}, \ldots, a_n, a_1, \ldots, a_c]$. Иными словами, это два отсортированных массива $[a_{c+1}, \ldots, a_n]$ и $a_1, \ldots, a_c$, выписанные подряд. Плюс, стоит не забывать, что $a_{c+1} > a_c$, то есть все элементы левой части больше всех элементов правой.

Этим критерием мы и воспользуемся. Запросим у интерактора значение $y = f(0)$ — это наш a_{c+1}. Заметим, что все элементы левой части (до a_n) не меньше y, а правой — наоборот, меньше. Таким образом, у нас теперь есть монотонный критерий, по которому можно делать бинпоиск: выберем m на середине текущего отрезка, и проверим:

- если $f(m) < y$, то этот элемент принадлежит правой части $[a_1, \ldots, a_c] = [f(n-c+1), \ldots, f(n)]$ — двигаем правую границу $r \leftarrow m$;

- если $f(m) \geqslant y$, то этот элемент принадлежит левой части $[a_{c+1}, \ldots, a_n] = [f(1), \ldots, f(n-c)]$ — двигаем левую границу $l \leftarrow m$.

В конце l и r будут соседними, но в разных частях, то есть $l = n - c$ и $r = n - c + 1$. Зная эти значения и число n, мы можем спокойно восстановить ответ. Для этого нам понадобилось всего либо $\lceil \log_2(n-1) \rceil + 1$ запросов, что с большим запасом укладывается в ограничения задачи.