Разбор задачи «Найти вершину»

Рассмотрим оптимальную стратегию.

Какое-то ребро будет спрошено первым, пометим его числом 0.

Дальше рассмотрим аналогичные раскраски (рекурсивно) по две стороны от этого ребра, но увеличим веса этих ребер на один (так чтобы был только один ноль)

Эта раскраска соответствует стратегии, и если k = максимальный вес ребра в ней, то стратегия соответствует поиску вершины за $k+1$ запрос в худшем случае.

У этой раскраски есть одно свойство которое всегда позволяет нам однозначно определить ход: на пути между любыми двумя ребрами одного цвета есть ребро меньшего цвета.

И любая раскраска с таким свойством является корректной стратегией!

Чтобы нам было проще в дальнейшем, «`Инвертируем`» веса в раскраске, пометим самое первое ребро цветом $k-1$, а дальше аналогично тому, как строилась раскраска по стратегии ранее, возьмем раскраски для поддеревьев, но вычитаем один из весов двух компонент (но чтобы веса остались неотрицательными).

Теперь нам нужно найти раскраску с минимальным максимальным весом, чтобы на пути между любыми двумя ребрами одного цвета было ребро большего веса.

Будем искать эту раскраску динамикой по поддеревьям.

Для фиксированной раскраски поддерева v, посмотрим какие цвета *видны* если смотреть с v вниз.

Скажем, что цвет видно, если есть ребро такого цвета, что на пути от этого ребра до v нет ребер большего цвета.

Введем *потенциал* раскраски: минимальная сумма $2^{c_1} + 2^{c_2} + \ldots + 2^{c_k}$, где c_i — видимые цвета. Обратите внимание, что этот потенциал это длинное двоичное число! Так как ответ может быть большим.

Утверждение: от поддерева интересует только раскраска с минимальным потенциалом.

Пусть поддеревья фиксированной вершины v уже покрашены в раскраски с минимальными потенциалами.

Тогда нам нужно выбрать веса ребер исходящих вниз из v, чтобы после привешивания этих ребер к поддеревьям вершины v, разные поддеревья не имели общих видимых цветов.

Можно покрасить ребро из вершины v в ее сына u в цвет c, если цвет c не является видимым из вершины u. После этого из множества видимых цветов пропадут все веса меньше c, но добавится цвет c.

Нам нужно поменять раскраски всех поддеревьев, чтобы при сложении потенциалов не возникло переносов (это будет соответствовать тому, что в разных поддеревьях нет общих видимых цветов). При этом условии нужно минимизировать итоговую сумму — потенциал v.

Из такой постановки *доказательство* предыдущего утверждения становится понятным: при большем потенциале возможные ходы строго не лучше.

Нужно решить задачу: дан массива длинных двоичных чисел $a_1, a_2, \ldots, a_k$. Нужно найти массив с минимальной суммой $b_1, b_2, \ldots, b_k$, что $b_i > a_i$, и при сложении $b_1 + b_2 + \ldots + b_k$ не возникнет переносов.

Эту задачу можно решить с помощью довольно простой жадности: будем выставлять биты жадно слева направо, и проверять, что можно дополнить ответ. Чтобы проверить, что можно построить какой-то ответ для данного множества чисел и верхней границы на число занятых бит, можно идти слева направо и каждый раз жадно заменять на текущий бит число с наибольшим суффиксом. Это можно реализовать совсем наивно за $\mathcal{O}(n^3)$, но также это может быть реализовано за $\mathcal{O}(n \log n)$.

Нам нужно решить эту подзадачу n раз, и мы получаем решение за асимптотику $\mathcal{O}(n \cdot T(n)) = \mathcal{O}(n^2 \log n \ldots n^4)$.

Разбор задачи «Фэйк Ньюз»

Посмотрим на постановку задачи в контексте префиксных сумм на массиве подсчета: нам в начале дается число $x = 1$, после чего происходит несколько переходов $x \to p_x + 1$, до того момента, когда $x = p_x + 1$. Нам нужно найти финальный x.

Будем поддерживать массив $t_x = p_x - x + 1$, в котором нам нужно найти какой-то ноль. Утверждается, что закончится алгоритм в первом нуле. Почему? Потому что если бы мы смогли в какой-то момент этот ноль «перепрыгнуть», то $p_y + 1 > x$ для $y < x$, но тогда $p_x + 1 > x$, противоречие.

Чтобы находить первый ноль, нам нужно обрабатывать запросы ± 1 на суффиксе массива t_n. Кроме этого воспользуемся тем, что $t_x - t_{x-1} \geqslant -1$, ведь вычитаем мы только число x. А это значит, что перед первым нулем не могло быть никаких отрицательных чисел, ведь них нельзя попасть раньше, чем в 0. Значит, нужно искать первый неположительный элемент в структуре данных, что можно делать с помощью спуска по дереву отрезков на минимум.

Разбор задачи «Отборочный этап»

Заметим, что ответ не меньше $\max(a + b, c + d)$, т.к. у нас найдётся хотя-бы 100 участников с суммой не меньше $a + b$ и 100 участников с суммой не меньше $c + d$ (возможно какие-то участники будут и там и там).

Если у нас будет 99 участников с баллами (a, c), и 2 участника с баллами (a, b) и (c, d) соответственно, то сумма баллов 100-го участника будет равна $\max(a + b, c + d)$, и условие задачи будет соблюдено, т.к. $a \geqslant d$, $c \geqslant b$ и $a + c \geqslant \max(a + b, c + d)$.

Разбор задачи «Сборы в поход»

В задаче даны m предметов, у каждого предмета известен тип и вес ($m \leqslant 23$). Нужно разложить предметы по минимальному количеству рюкзаков вместимостью s. В одном рюкзаке не должно быть двух предметов одного типа.

Воспользуемся методом динамического программирования. Пронумеруем предметы от 0 до $m-1$ так, чтобы предметы одного типа шли подряд. Состоянием динамического программирования будет пара из двух чисел: $mask$ — множество взятых предметов и i — номер текущего предмета, либо вспомогательный элемент равный m. Значением для состояния будет пара из двух чисел: количество использованных рюкзаков и занятое место в последнем рюкзаке. Эту значение мы будем минимизировать как пару: в первую очередь количество рюкзаков, а при равенстве — занятое место в последнем рюкзаке.

Обозначим за $next[i]$ минимальное j, что $i < j$ и тип предмета номер j отличается от типа предмета номер i, либо $j = m$, если такого предмета не существует.

Сделаем следующие переходы в решении:

- $dp[mask][i] \to dp[mask][i + 1]$, где $0 \leqslant i < m$. Пропускаем i-й предмет, не добавляем его в текущий рюкзак.
- $dp[mask][i] \to dp[mask \cup \{i\}][next[i]]$, где $0 \leqslant i < m$, $i \notin mask$. Добавляем i-й предмет в текущий рюкзак, пропускаем все предметы с тем же типом.
- $dp[mask][m] \to dp[mask \cup \{i\}][next[i]]$, где $0 \leqslant i < m$, $i \notin mask$. Заканчиваем набирать рюкзак, начинаем набирать новый пустой рюкзак, кладем туда предмет с номером i и пропускаем все предметы с тем же типом.

Несложно заметить, что для любого корректного разбиения множества предметов по рюкзакам, будет существовать последовательность переходов, соответствующая такому разбиению. Поэтому, найденный ответ будет не хуже оптимального. С другой стороны, любая последовательность переходов соответствует корректному разбиению предметов на множества, поэтому ответ будет не лучше оптимального.

Суммарно получилось $O(2^n \cdot n)$ состояний, столько же суммарно переходов. Поэтому, время работы и занимаемая память равны $O(2^n \cdot n)$.

Для того, чтобы решение уложилось в ограничение по времени, могло понадобиться дополнительно уменьшить используемую память. Заранее упорядочим все множества в порядке возрастания количества предметов в них. Тогда в каждый момент времени достаточно хранить значения для не более чем $S \cdot 2$ последовательных множеств, где S — максимальное количество различных множеств с одинаковым количеством элементов ($S = 1352078$ при данных ограничениях).

Разбор задачи «Деление»

Пусть $y = p/q$. Тогда если найдётся такое простое a, такое что y делится на a, и q не делится на a, то если домножить x на a, то результат всё ещё не будет делится на q, при этом будет делителем p. Значит для максимального x такого a не существует.

Предположим что найдутся такие простые числа a и b, такие что и y и q делятся на каждое из них. Так как x не делится на q, то существует такое простое число c (возможно равное a или b), такое что степень вхождения c в x меньше, чем степень вхождения c в q. Тогда заметим, что какое-то число из a и b точно не равно c, значит при домножении x на такое число, степень вхождения c в результат всё ещё будет меньше, чем в q, а значит результат не будет делиться на q. Значит для максимального x таких чисел a и b найтись не может.

Тогда $x = p/$(какой-то простой делитель q в какой-то степени). Тогда для решения задачи факторизуем число q за $O(\sqrt{q})$, и так найдём все его простые делители (их не более $\log q$). Далее будем делить p на каждый простой делитель, пока результат не перестанет делиться на q. Из таких чисел выберем максимальное, это и будет искомое q. Итоговая асимптотика: $O(\sqrt{q} + \log p \cdot \log q)$.

Разбор задачи «Парад во время чумы»

Применим в данной задаче бинарный поиск по ответу. Пусть M проверяемая величина. Воспользуемся методом динамического программирования, пусть $dp[i][j]$ - можно ли построить шеренгу с разностью не превосходящей M из первый i строк и последним будет j-й элемент i-й строки.

Для вычисления $dp[i][j]$ нужно для каждого элемента понять есть ли элемент на прошлой строке, который отличается на не более M и при этом для него есть положительное состояние.

Для этого можно применить метод двух указателей (предварительно отсортировав все строки) и воспользоваться префиксной суммой для нахождения числа ненулевых значений динамики, то есть суммы на отрезке.

Итого $O(nm \log max\_ans)$.

Разбор задачи «Прямоугольная ломаная»

Сначала заметим, что в правильной ломаной, так как горизонтальные и вертикальные отрезки чередуются, $h = v$: если это равенство не выполняется, ответ будет отрицательный. Зафиксируем какую-то вершину и обойдём ломаную в каком-то направлении. Тогда в процессе обхода на каждом из ходов мы будем смещаться в одном из четырёх возможных направлений: вверх, вниз, вправо или влево. Так как ломаная является замкнутой, это значит, что суммарно мы сместимся влево на такое же расстояние, на сколько мы суммарно сместимся влево. Аналогичное верно и для перемещений вверх и вниз.

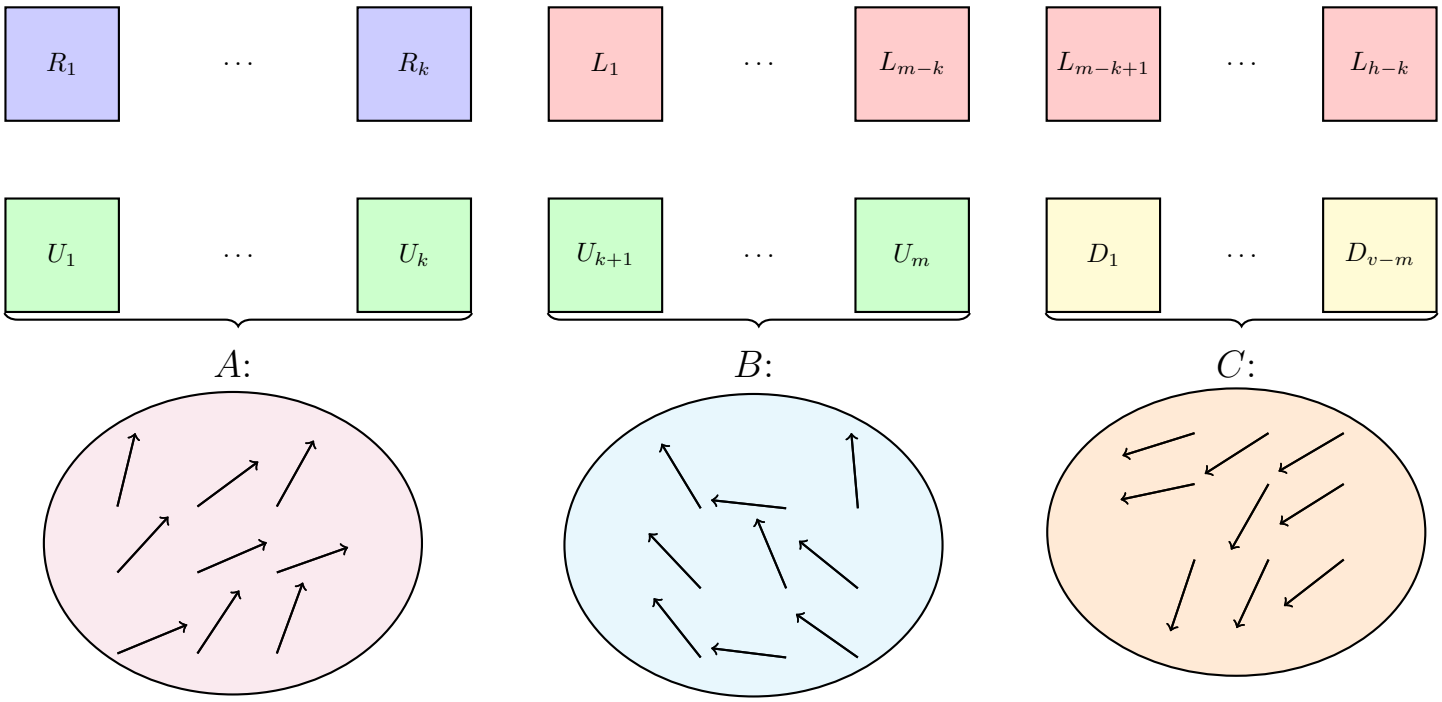
Это означает, что если мы разбили все отрезки на четыре множества — Up, $Down$, $Left$, $Right$, то суммарная длина отрезков в Up будет равна суммарной длине отрезков в $Down$, а суммарная длина отрезков в $Right$ будет равна суммарной длине отрезков в $Left$. Но отсюда следует, что множество длин всех горизонтальных отрезков можно разбить на два множества с одинаковой суммой. То же самое должно выполняться для вертикальных отрезков.

Проверим, можно ли разбить множество длин горизонтальных отрезков на два множества одинаковой суммы. Эта классическая задача может быть решена, если применить метод динамического программирования для решения задачи о рюкзаке. Сложность такого решения будет $\mathcal{O}\left(nC^2\right)$ Если горизонтальные или вертикальные длины разбить на два множества равной длины невозможно, то ответ будет `«No»`. Покажем ниже, как сконструировать ответ, если такое разбиение существует.

Пусть мы разбили все горизонтальные длины на два множества равной суммарной длины. Множество меньшего размера обозначим как R, множество большего — L. То же самое сделаем со множеством длин вертикальных отрезков: множество меньшего размера обозначим как D, большего — как U. Так как $|R| \leqslant |L|$, то $|R| \leqslant h/2 = v/2$. Аналогично имеем $v/2 \leqslant |U|$, а значит $|R| \leqslant |U|$, $|D| \leqslant |L|$.

Разобьем все отрезки на пары следующим образом: каждому отрезку из R сопоставим какой-то отрезок из U. Всем оставшимся отрезкам из L сопоставим какой-то из оставшихся вертикальных отрезков. Таким образом мы разбили все данные отрезки на три множества пар: в первом в пару

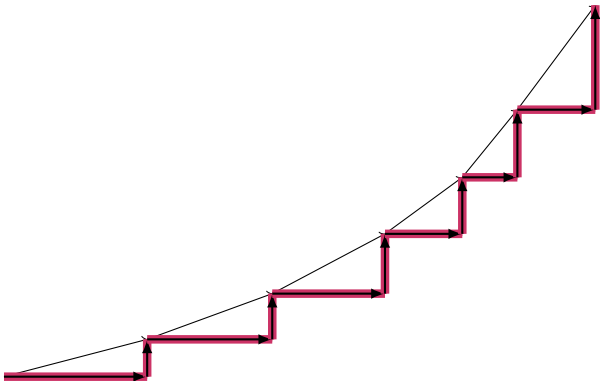
к отрезку из R ставится отрезок из U. Во втором отрезку из L ставится в пару отрезок из U, в третьем в пару к отрезку из L ставится отрезок из D.



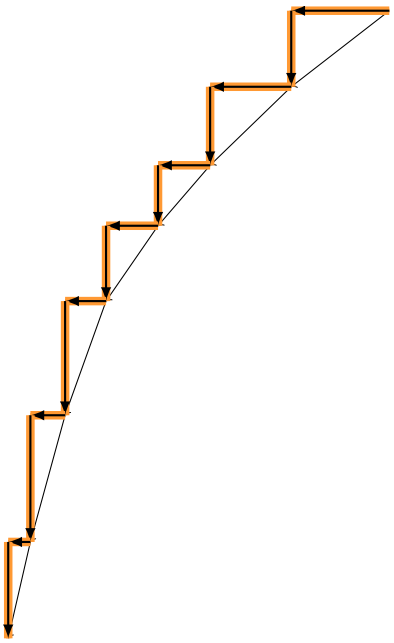
Из первого множества пар составим множество векторов, направленных вверх-вправо (из пары (r, u) составим вектор (r, u)) — получится множество векторов A. То же самое сделаем со вторым множеством пар (получим множество векторов B) и третьим множеством пар (получим множество векторов C), для лучшего понимания смотрите картинку выше. Обратите внимание, что множество B может отказаться пустым, тогда как остальные два — нет.

Составим из векторов множества A выпуклую вниз ломаную — для этого отсортируем их в порядке возрастания полярного угла и в этом порядке составим из них ломаную (смотри картинку ниже).

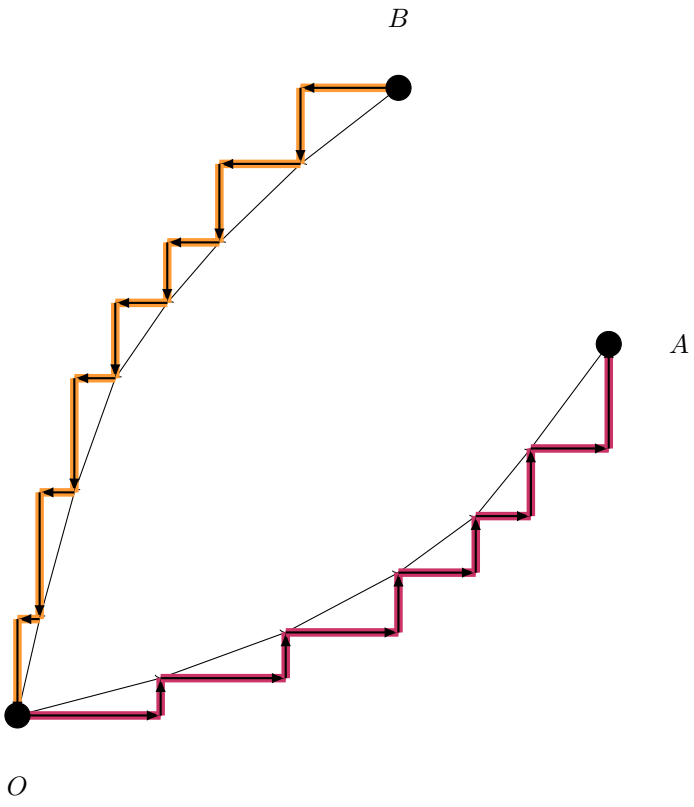
Теперь заменим каждый из векторов нашей ломаной на два вектора — один вектор, направленный вправо и один вектор, направленный вверх.



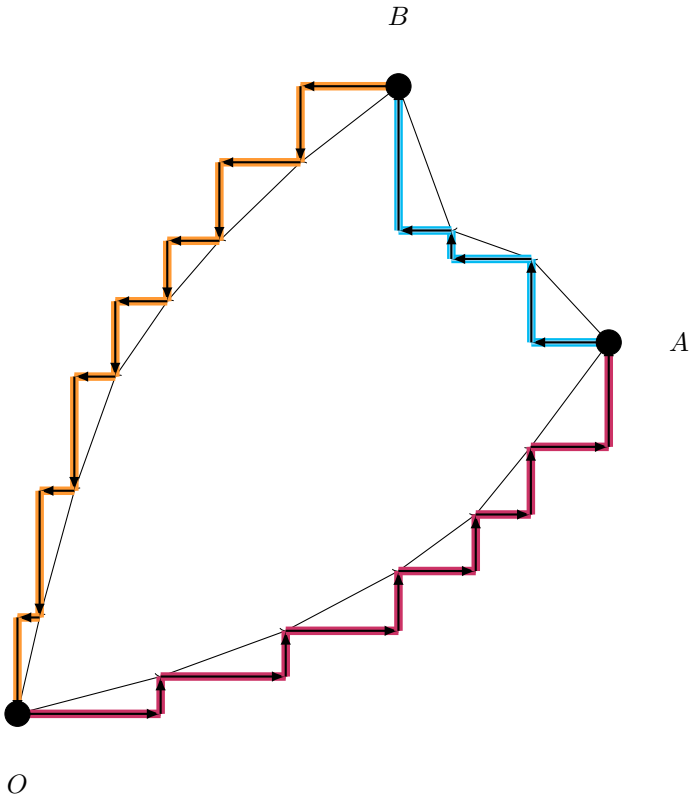
Аналогичное сделаем для векторов из C — отсортируем их в порядке возрастания полярного угла и составим из них выпуклую вниз ломаную:



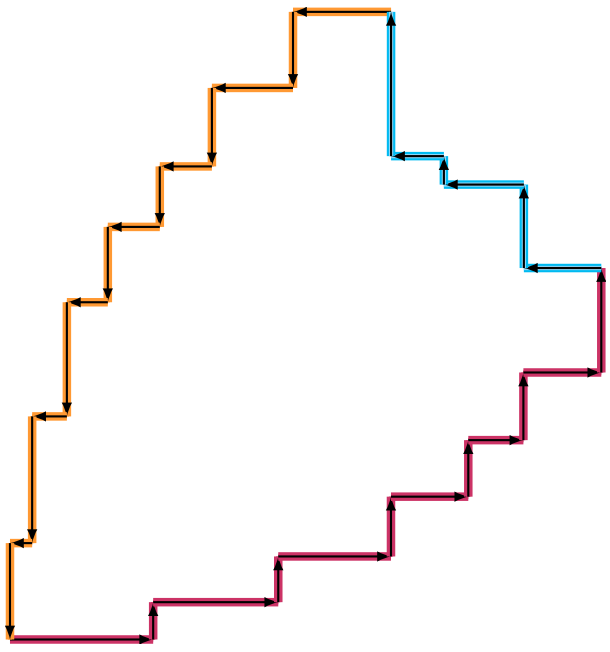
Объединим эти две ломаные таким образом, чтобы первая вела из точки O в точку A, вторая — из точки B в точку O:



Осталось соединить точки A и B с помощью векторов из множества B. Это можно сделать произвольным образом: взять эти вектора (направленные вверх-влево) в каком-то произвольном порядке, тогда, так как сумма всех векторов равна 0, то полученная ломаная, если строить её с началом в точке A, закончится в точке B. Так как первые две построенные ломаные были выпуклы, это значит, что все строго внутри угла AOB не будет лежать ни одна из точек первых двух ломаных, а значит, если заменить каждый из векторов третьей ломаной на два вектора — направленный влево и направленный вверх, то полученная замкнутая ломаная не будет содержать самопересечений.



Нетрудно показать, что полученная ломаная будет замкнутой, не содержать самопересечений не по концам отрезков и будет удовлетворять всем условиям задачи:



Разбор задачи «Тимбилдинг»

Формальная постановка задачи: дан неориентированный граф без петель и кратных ребер, у каждой вершины есть какой-то цвет от 1 до k. Необходимо посчитать количество пар цветов таких, что если оставить в графе только вершины этих двух цветов и все ребра между такими вершинами, то он будет являться двудольным.

Сначала рассмотрим медленное решение, а потом ускорим его.

Проверим подграфы, образованные каждым цветом, на двудольность (например, поиском в глубину). Это можно сделать за $O(n+m)$. Недвудольные цвета далее рассматривать не будем, они не могут участвовать ни в каких парах.

Рассмотрим какой-то цвет x. Пусть из вершин такого цвета есть ребра в вершины цвета $y_1, y_2, \ldots, y_k$. Проверим на двудольность (также поиском в глубину) пары $(x, y_1), (x, y_2), \ldots, (x, y_k)$, тем самым выяснив, какие цвета нельзя взять в пару с x. Остальные можно. Сделаем так для каждого цвета, и тем самым найдем ответ.

Заметим, что каждое ребро, соединяющее различные цвета, мы посмотрим два раза. Проблема: ребра, соединяющие один цвет, которых может быть много, мы можем посмотреть до k раз.

Теперь ключевая идея. Критерий двудольности графа: отсутствие в нем нечетного цикла. Представим какую-то двудольную компоненту связности, образованную одним цветом. Если цикл в графе, образованном двумя цветами, проходил по этой компоненте, то нам неважно, как именно он это делал. Если он начал и закончил в одной и той же доле, то путь внутри этой компоненты имел четную длину, а если в разных, то нечетную. Это позволяет нам сжать такую компоненту в две вершины — по одной на каждую долю, и провести ребро между этими вершинами. Для каждого цвета сожмем таким образом все образованные им компоненты.

В результате сжатия у нас появился новый граф, где каждая компонента связности представляет из себя либо одну вершину, либо две вершины, соединенные ребром. Будем делать так же, как в решении, разобранном ранее — проверять каждую пару цветов, соединенную ребрами, на двудольность, но теперь в сжатом графе. Пусть мы хотим проверить пару (x, y). Для каждого ребра между этими цветами в исходном графе проведем соответствующее ребро в сжатом графе между вершинами, которые соответствуют компонентам и их долям, которые соединяло ребро в исходном. После этого поиском в глубину проверим на двудольность получившийся граф, затем откатим изменения и сделаем все то же самое для других пар.

За сколько работает проверка одной пары цветов (x, y)? В сжатый граф мы добавили только ребра, соединяющие цвета x и y. При поиске в глубину будем запускать его только от компонент, которые были затронуты добавленными ребрами, потому что незатронутые компоненты никак не повлияют на двудольность, но их может быть много. При таком подходе поиск в глубину рассмотрит все добавленные ребра, а также какое-то количество ребер, соединяющие вершины одного цвета. Но таких ребер будет максимум в два раза больше, чем добавленных, потому что каждое добавленное ребро соединяет максимум две новых компоненты, а в каждой компоненте максимум одно ребро. Проверка каждой пары цветов работает за количество ребер между этими цветами, а значит, суммарно при проверке всех пар мы потратим $O(m)$ времени! Итоговое время работы: $O(n+m)$ или $O(m \log n)$, в зависимости от реализации.

Разбор задачи «ТВ»

Для начала научимся решать следующую подзадачу: у нас есть k запросов, i-й имеет вид: задано два числа a_i и b_i, найдите какую первую интересную передачу вы увидите, если начнете смотреть телевизор с телеканала a_i в момент времени b_i (вернее, на каком телеканале вы увидите эту передачу). Обозначим ответ на такой запрос как $nxt(a_i, b_i)$ (будем считать, что если такой передачи нет, то ответ на запрос равен -1).

Для этого будем просматривать эти запросы в порядке увеличения значения a, при этом поддерживать множество запросов, на которые мы еще не ответили (для этого подойдет `set` в `C++` или `TreeSet` в `Java`). После того, как мы обработали все запросы с определенным значением a (например все запросы с $a_i = x$), мы можем ответить на часть этих запросов. А именно ответом на некоторые запросы из текущего множества может быть телеканал под номером x. Понятно, что это запросы, для которых выполняется условие $l_x \leqslant b_i \leqslant r_x$.

Теперь, когда мы умеем отвечать на такие запросы, мы можем решить задачу. Для этого воспользуемся динамическим программированием.Подсчитаем динамику dp_i, равную суммарному времени, которое Алексей будет смотреть интересные передачи, если начнет смотреть телеканал i в момент времени r_i. Значение dp_i будет равно $dp_{ni} + (r_{ni} - r_i)$, где $ni = nxt(i, r_i)$ (либо нулю, если $ni = -1$). При помощи такой динамики легко отвечать на первоначальные запросы. Ответ на первоначальный запрос (x_i, t_i) равен $dp_{nx} + (r_{nx} - t_i)$, где $nx = nxt(x_i, t_i)$ (либо нулю, если $nx = -1$).

Разбор задачи «Разбивай и суммируй»

Как бы мы ни разбивали массив, стоимость разбиения всегда будет одинаковой.

Докажем это. Без потери общности будем считать массив a отсортированным и обозначим за L множество элементов с номерами от 1 до n, а за R множество элементов с номерами от $n+1$ до $2n$.

Разобьем массив a на два любых массива p и q размера n. Отсортируем p по неубыванию, а q по невозрастанию. Любая разница $|p_i - q_i|$ в нашей сумме будет разницей одного элемента из R и одного элемента из L.

Допустим это не так, тогда найдется такой индекс i, что и p_i и q_i принадлежат одному множеству. Пусть это множество L.

1. Все элементы с индексами меньшими или равными i в p принадлежат L (это i элементов)
2. Все элементы с индексами большими или равными i в q принадлежат L (это $n-(i-1)$ элементов)

Тогда в L хотя бы $i + n - (i-1) = n+1$ элементов, а их должно быть ровно n. Противоречие. Для множества R доказательство аналогичное.

Тогда ответ на задачу — это (сумма элементов множества R минус сумма элементов множества L) умноженная на количество разбиений массива C_{2n}^{n}.

Время работы — $O(n \log n)$ (из-за сортировки)