Разбор задачи «Sheet Metal»

Для решения задачи требуется рассмотреть варианты как можно расположить прямоугольный лист под формой:

- если форма больше или равна листу, то есть $w_1 \leqslant w_2$ и $h_1 \leqslant h_2$, то ответ 0, так как форма может покрыть полностью лист.
- если форма меньше листа, то есть $w_1 > w_2$ и $h_1 > h_2$, то как бы мы ни двигали лист то он не распадется и площадь будет $w_1 \cdot h_1 - w_2 \cdot h_2$.
- Пусть $w_1 \leqslant w_2$ и $h_1 > h_2$ тогда надо делить лист на два равных куска, чтобы максимальная площадь среди всех оставшихся кусков была минимальной, в этом случае площадь $\frac{w_1 \cdot (h_1 - h_2)}{2}$.
- Пусть $w_1 > w_2$ и $h_1 \leqslant h_2$ тогда надо делить лист на два равных куска, чтобы максимальная площадь среди всех оставшихся кусков была минимальной, в этом случае площадь $\frac{h_1 \cdot (w_1 - w_2)}{2}$.

Важно не забыть что лист можно повернуть, поэтому надо рассмотреть такие же случаи. Ответом будет минимум из ответов для варианта с поворотом листа и без.

Разбор задачи «Lunchtime Fruits»

Немного формализуем задачу. У нас есть объекты типов A, B, C и D. Из них можно составлять наборы $AABC$, $AACC$ и $ABCCD$. Сделаем замену $X = AB$, $Y = AC$, $Z = BCD$. Тогда наборы заменятся на XY, YY и YZ. К сожалению, мы не можем однозначно определить числа X, Y, Z. Если мы по-разному распределим их количество, могут получиться разные ответы. Но зато мы можем легко определить, можно ли сделать определенное количество наборов.

Для этого заметим, что в каждом наборе есть Y. Мы можем сразу вычесть эту часть из исходных данных, и останутся наборы типов X, Y и Z. Дальше проверить можно, например, так. (a — количество объектов типа A; b — количество типа B и так далее; k — количество наборов, которое мы хотим набрать). Если $b + c \leqslant a$, то максимум мы можем набрать $b + c$ наборов, поэтому просто проверяем, что $b + c \geqslant k$. В ином случае мы набираем сначала X и Y поровну (если какого-то типа слишком мало, то как можно ближе к половине), чтобы осталось как можно больше Z. Формульно эта проверка выражается как $min(b, c, d, \frac{b+c-a}{2}) \geqslant k - a$.

Теперь, когда мы научились проверять, можно ли сделать k наборов, просто запустим двоичный поиск по ответу по величине k.

Разбор задачи «Sliding Dominoes»

Посмотрим на какую-то клетку, покрытую доминошкой. В какой ситуации она станет пустой в первый раз? Для этого нужно подвинуть доминошку, которая накрывает эту клетку в направлении «от нее», то есть, перед этим, должна быть свободна клетка на расстоянии два.

Таким образом, для каждой клетки A есть клетка B, которую нужно сперва освободить, чтобы потом иметь возможность передвинуть домино и освободить A. Назовем клетку B родителем клетки A. Учитывая, что на поле также есть одна свободная клетка, все клетки разбиваются на недостижимые (те, которые невозможно сделать пустыми, в частности, все клетки, не совпадающие с изначально пустой в шахматной раскраске) и дерево с корнем в стартовой пустой клетке.

Осталось научиться решать задачу на дереве: мы можем обходить дерево, начиная с корня, и должны максимизировать сумму посещенных вершин. Такую задачу можно решать динамическим программированием: `dp[v]` равно ответу для поддерева вершины v, включая ее саму. Тогда $\texttt{dp[v]} = \max(0, \texttt{cost[v]} + \sum_u \texttt{dp[u]})$. Такое решение работает за линейное время.

Разбор задачи «Lost in Translation»

Разберем подзадачи по порядку. Начнем с $m = 2n$, $k = 4$. В этой подзадаче можно действовать, например, следующим образом: заменим `0` на `AB`, заменим `1` на `CD`. Теперь, например, если удалили `C`, то можно удалить `A` и получить изначальную битовую строку.

Рассмотрим случай $k = 3$, когда можно использовать только три символа. Заменим `0` на `AC`, а `1` на `BC`. Таким образом, если удалить символ `C`, то получается в точности изначальная битовая строка. Случаи, когда удаляются `A` и `B` симметричны, рассмотрим для простоты удаление `A`. Если первый символ в строке — `B`, то за ним следует `C`, и значит, декодированная строка будет начинаться с `1`. Если же закодированная строка начинается с `C`, значит, там раньше было `AC`, а значит, нужно поставить `0`.

Однако, при таком подходе мы не сможем отличить ситуацию, когда в строке `ACACAC` удалили `A` и в строке `BCBCBC` удалили `B`, поэтому для строк `000` и `111` нужно (в таком подходе) специальное поведение.

Заметим общий принцип: мы заменяем `0` и `1` на какие-то коды, и дальше пытаемся раскодировать, используя знания о том, как выглядят эти коды. Можно попытаться обобщить это решение: для каждой двоичной строки длины c мы придумываем код из букв `A`, `B`, `C` и кодируем целую строку, разбив ее на куски по c бит. Закодированные куски должны обладать свойством, что при удалении всех букв `A` (или `B`, или `C`), все куски должны быть различными. Однако, это не единственное требование, при склеивании разных кусков тоже должны получаться разные. Это не всегда верно, например, существует кодирование 2 битов в 3 символа, но из него нельзя сделать кодирование 4 битов в 6 символов.

Однако, если брать достаточно случайное соответствие c-битным кускам, то окажется, что часто бывает так, что заданную строку можно восстановить однозначно. Найти число способов восстановить строку можно с помощью динамического программирования по префиксам (`dp[i][j]` — количество представить первые i символов как конкантенацию j кусков с удаленными символами). Если это количество равно 1, то во втором запуске можно будет с помощью этого же соответствия (его можно предпосчитать в программе, или запомнить сид рандома) восстановить однозначно исходную строку. В зависимости от реализации и эффективности, можно построить отображение 10-битных кусков в 19 или 18 символов.

Также, существует отображение 20-битных кусков в коды длины 36 таким образом, что каждый код содержит по 12 символов `A`, `B` и `C`. Плюс этого кода в том, что дальше любая строка восстанавливается однозначно, поскольку при удалении любого символа остаются коды длины ровно 24.

Разбор задачи «Summer School»

Нам задан двудольный граф, описанный неявно, в одной доле школьники, в другой места в летней школе. Требуется найти размер максимального подмножества школьников, для которых существует паросочетание их покрывающее.

Первые подзадачи можно решать с помощью алгоритмов нахождения паросочетания и потока в двудольном графе.

Для полного решения воспользуемся обобщенным утверждением из теоремы Холла: размер минимального подмножества школьников, которым нельзя найти место — $\max\left\{0, \max\limits_{A} |A| - f(A)\right\}$. В теореме Холла проверяется, что это число равно 0.

Заметим, что для каждого уровня набор подходящих параллелей это некоторый отрезок $[L_i, R_i]$. И эти отрезки монотонны в следующем смысле: $L_i \leqslant L_{i+1}$ и $R_i \leqslant R_{i+1}$.

Если a_i — число школьников уровня i, то $|A| - f(A) = a_i - k \cdot (R_i - L_i + 1)$, когда A — это школьники уровня i.

Наша задача найти $\max\limits_{A} |A| - f(A)$, то есть выбрать какие-то уровни, чтобы соответствующее выражение было максимальным. Проблема в том, что отрезки второй доли могут пересекаться, а учесть их надо один раз.

Сделаем функцию динамического программирования: $d(i)$ — максимальная $|A| - f(A)$, если выбирали уровни только $\leqslant i$ и уровень i выбрали. Тогда либо $d(i) = d_j + a_i - k \cdot (R_i - L_i + 1)$ для некоторого $j < i$, либо $d(i) = d(i-1) + a_i - k \cdot (R_i - R_{i-1})$.

Это основано на том, что если мы взяли два уровня, у которых отрезки $[L_i, R_i]$ и $[L_j, R_j]$ пересеклись, то все уровни между ними тоже выгодно взять, так как в левой доле элементов прибавится, а в правой не прибавится, что только увеличит сумму.

В первом случае $d(i) = d_j + a_i - k \cdot (R_i - L_i + 1)$ предполагается, что $[L_i, R_i]$ и $[L_j, R_j]$ не пересекаются, а во втором — пересекаются. Однако в первом случае можно рассматривать все возможные $j < i$, так как если пересечение есть, то это учтет элементы в правой доле дважды, что даст меньшую сумму, что будет меньше найденного максимума, соответственно, не нарушит посчитанное значение.

Получается, что чтобы посчитать значение $d(i)$, нужно хранить $d(i-1)$ и $\max\limits_{j<i} d_j$.

Для простоты в переходе $a_i - k \cdot (R_i - L_i + 1)$ назовем $open_i$, а $a_i - k \cdot (R_i - R_{i-1})$ — add_i. Тогда переходы будут как $d(i) = d_j + open_i$ и $d(i) = d_{i-1} + add_i$.

Можно представить это как у нас есть вектор (x, y), где $x = d(i-1)$, а $y = \max\limits_{j<i} d_j$. Переход от i к $i+1$ — $(\max\{y + open_i, x + add_i\}, \max\{y, y + open_i, x + add_i\})$.

Это можно представить в виде матрицы, с переопределенной операцией умножения: $c_{ij} = \max\limits_k a_{ik} + b_{kj}$. Тогда переход будет:

$$\begin{pmatrix} add_i & open_i \\ add_i & \max\{0, open_i\} \end{pmatrix} \oplus \begin{pmatrix} d(i) \\ \max\limits_{j<i} d_j \end{pmatrix} = \begin{pmatrix} d(i+1) \\ \max\limits_{j \leqslant i} d_j \end{pmatrix}$$

Нужно создать дерево отрезков, в который положить матрицы и считать их произведение, домножая матрицу $i+1$ к i слева. При изменении одного элемента, изменяется a_i, соответственно изменяется add_i и $open_i$, можно заново создать матрицу и сделать присвоение одного элемента дерева отрезков. Решение работает за $\mathcal{O}((m+n)\log n)$.