Задача A. План защиты

Автор задачи и разработчик: Мария Жогова

Основные тезисы из условия:

1. можно сражаться против атак с силой не больше m;
2. нельзя сражаться против атак с силой больше m;
3. «пропускать» можно ровно x атак подряд, при этом на момент окончания последней атаки необходимо быть «дома».

В первой подгруппе подойдет любое решение, находящее ответ. В частности, можно написать рекурсивный перебор, который каждый раз рассматривает два варианта: встретить текущую атаку или пропустить следующие x, если это не запрещено условием. Если рекурсивный перебор дойдет до последней атаки, ответ существует, можно его запомнить, чтобы в конце вывести минимальный из всех найденных. Иначе ответа нет. Псевдокод приведен ниже:

```
go(skipped, i):
    if i > n:
        запомнить ответ skipped
        return
    if a[i] <= m:
        go(skipped, i + 1) // не пропускаем
    if i + x - 1 <= n:
        go(skipped + [i], i + x) // пропускаем
```

Заметим, что силы атак не важны, важно только больше они, чем m, или нет. Все атаки, которые имеют силу больше m, необходимо переждать на островах. Таким образом, подгруппа $x = 1$ требует просто вывести номера всех атак, сила которых $> m$.

Остальные группы являются подсказками к полному решению. Так, например, придумав решение для $x = 2$, может быть проще придумать решение для произвольного x. Ограничение $x = 2$ означает, что атаки можно пропускать только по две подряд, в частности, нельзя пропустить последнюю без предпоследней. Если последняя атака имеет $a_n > m$, то ее необходимо пропустить, значит перед $n - 1$-й атакой придется просить убежище. Иначе нет смысла пропускать ее, и можно перейти к рассмотрению предыдущей.

Проблема может возникнуть, когда такой алгоритм дойдет до a_1, и окажется, что ее надо пропустить, а a_2 уже покрыта другим отрезком времени в убежище. В таком случае сделаем следующее:

1. покроем первые две атаки одним отрезком;
2. если вторая была покрыта отрезком $[2, 3]$, то третью атаку надо пропустить — сдвинем отрезок вправо на 1, получим $[3, 4]$;
3. если при этом четвертая была покрыта $[4, 5]$, сдвинем его вправо на 1, и так далее рекурсивно.

Наша рекурсия займет $\mathcal{O}(n)$ времени и либо выставит все отрезки на непересекающиеся позиции, либо окажется, что ни одна атака не была встречена, и Оматикайя провели все время в полетах между островами и обратно, тогда последний отрезок $[n - 1, n]$ нельзя сдвинуть вправо на 1. Но в таком случае массив a имеет вид $[> m, ?, > m, ?, \ldots, ?, > m]$, и все его элементы $> m$ в принципе нельзя покрыть отрезками длины 2.

Это подводит нас к полному решению. Во-первых, можно доказать, что такой жадный алгоритм действительно находит минимальное количество раз, которое клану придется просить убежище. Это стандартная задача. Каждый элемент $> m$ должен быть покрыт каким-то отрезком длины x. Если существует какое-то другое покрытие, можно в нем сдвинуть несовпадающие с нашим покрытием отрезки влево и получить наше покрытие, не увеличив число отрезков.

Собственно, для написания решения формально доказывать этот факт не требовалось, корректность жадного алгоритма в данной задаче кажется достаточно интуитивно понятной, чтобы от нее можно было отталкиваться.

Во-вторых, надо понять, как разбираться со случаями, когда, выбирая самый левый из покрывающих очередную атаку отрезков, мы приходим к тому, что не можем покрыть несколько первых атак. Но на самом деле полностью аналогичное случаю $x = 2$ рассуждение работает и для $x > 2$. Таким образом, псевдокод алгоритма получается довольно простым:

```
for i = n ... 1:
    if a[i] > m и (i не покрыто последним отрезком):
        segments.push_front((i - x + 1, i))
    i -= 1

for t = 0 ... |segments|:
    if segments[t] не пересекает предыдущий: // и левую границу массива
        break
    сдвинуть segments[t] вправо

if segments.back().right > n: // последний отрезок вышел за границу
    вывести -1
else:
    вывести segments
```

Задача B. Паякан в беде

Автор задачи и разработчик: Даниил Голов

Как обычно, первая подгруппа решается полным рекурсивным перебором. Достаточно для текущего состояния перебрать все возможные следующие и запуститься от них. В конце из всех найденных путей требуется выбрать минимальный.

Чтобы проверять, возможно ли определенное действие,

- при движении надо проверить, что символ в новой клетке равен '.';
- при повороте надо проверить, что в каждой новой занимаемой клетке символ равен '.', это можно сделать за время k, просто итерируясь по ним.

Проще всего итерироваться в данном направлении, храня массивы $\texttt{dx} = [1, 0, -1, 0]$ и $\texttt{dy} = [0, -1, 0, 1]$, тогда сдвиг в направлении d — это сдвиг из (i, j) в $(i + \texttt{dy}[i], j + \texttt{dx}[j])$, тогда как поворот по или против часовой стрелки — это изменение d на ± 1.

Во второй подгруппе можно заметить, что для $n, m > 1$ даже при одном рифе на поле один из двух путей «вправо, затем вниз» и «вниз, затем вправо», всегда свободен (если риф не в $(1, 1)$ и не в (n, m)), поэтому ответ равен $n + m - 2k + 1$ ($n - k$ и $m - k$ перемещений, и один поворот). Если же на доске есть риф, и $n = 1$ или $m = 1$, либо если риф занимает первую или последнюю клетку, то пути нет, и ответ на задачу — «-1».

Для решения следующих подгрупп стоит воспользоваться поиском в ширину для нахождения минимального пути в графе. Вершинами в графе будут состояния (i, j, d) — позиция и текущее направление. Из каждой вершины ведет не более четырех ребер — движение по направлению и против направления и поворот по или против часовой стрелки. Требовалось просто аккуратно реализовать рассмотрение этих четырех ребер, для каждого из новых состояний проверив, возможно ли оно.

Для решения последней подгруппы нельзя проверять возможность поворота за $\mathcal{O}(k)$. Требуется быстро проверять, свободны ли от рифов определенные k клеток подряд, или нет. Для этого предподсчитаем $\texttt{right}[i, j]$ и $\texttt{down}[i, j]$ — максимальное количество свободных подряд клеток, начиная от (i, j) вправо и вниз, соответственно.

Эти значения можно вычислить динамическим программированием: для клеток, занятых рифами, эти значения равны 0, а для остальных $\texttt{right}[i, j] = \texttt{right}[i, j+1]+1$ и $\texttt{down}[i, j] = \texttt{down}[i+1, j]+1$. Используя эти значения, можно быстро проверять возможность поворота. Полное решение имеет асимптотику времени работы $\mathcal{O}(nm)$.

Задача C. Связь с Эйвой

Автор задачи: Даниил Орешников, разработчик: Константин Бац

Для решения первой подзадачи нужно было сделать несколько переборов. Переберем генетические коды первого и второго Аватаров для рекомбинации, сравним полученный код со всеми кодами в первом поколении. Все существует n кодов, сравнение двух кодов происходит за $\mathcal{O}(n)$. Таким образом, сложность такого решения — $\mathcal{O}(n^4)$.

Во второй подзадаче в строке было не больше двух символов ‘b’, а все остальные символы были равны ‘a’. Рассмотрим случаи:

- В строке только символы ‘a’. Тогда ответ вопрос задачи — 0.
- В строке только один символ ‘b’. Тогда, вне зависимости от расположения символа ‘b’, путем рекомбинации можно получить коды, в которых содержится от нуля до двух символов ‘b’. Новыми из них будут только те, которые содержат ноль или два символа ‘b’.

  Получить код без символа ‘b’ можно рекомбинацией кода, у которого символ ‘b’ на четной позиции, и кода с символом ‘b’ на нечетной позиции. Кодов обоих видов $\frac{n}{2}$, поэтому существует $\frac{n^2}{4}$ различных пар индексов.

  На самом деле, получить код с двумя символами ‘b’ можно получить из тех же пар индексов, если элементы пар поменять местами. То есть существует столько же $(\frac{n^2}{4})$ подходящих пар индексов.

  Таким образом, ответ в этом случае равен $\frac{n^2}{2}$.
- В строке два символа ‘b’. Для решения этого случая надо сделать ключевое наблюдение: *количество способов получить каждый циклический сдвиг s с помощью рекомбинации одинаково.* Действительно, если рекомбинация i и j дает k, то рекомбинация $(i + t) \bmod n$ и $(j + t) \bmod n$ даст $(k + t) \bmod n$.

  – Пусть два символа ‘b’ стоят в s на позициях разной четности. В таком случае, какие бы два кода для рекомбинации мы ни выбрали, из каждого будет выбрана ровно одна буква ‘b’, и в итоговом коде тоже будут ровно две буквы ‘b’.

    Заметим, что четные и нечетные позиции любого циклического сдвига сами являются циклическим сдвигом строки «aa...ab» (длины $\frac{n}{2}$). Тогда сколько есть способов получить строку s с помощью рекомбинации? Необходимо выбрать два сдвига, у которых ‘b’ стоит на строго определенных позициях среди четных и нечетных соответственно. И тех, и тех, ровно по 2, поэтому всего есть ровно 4 способа получить строку s.

    Всего есть n^2 пар особей, из них надо вычесть по 4 на каждый различный циклический сдвиг s. Всего различных циклических сдвигов s либо n, либо $\frac{n}{2}$, если ‘b’ стоят на расстоянии $\neq \frac{n}{2}$ или $= \frac{n}{2}$ соответственно.
  – Последний случай — когда два символа ‘b’ стоят на позициях одной четности. Тогда у каждого циклического сдвига позиции одной четности содержат только буквы ‘a’, а другой — две буквы ‘b’ на одном и том же расстоянии друг от друга.

    Результатом рекомбинации, таким образом, будет либо строка только из букв ‘a’ (новая), либо строка с двумя буквами ‘b’ на том же расстоянии, что и в s (старая), либо строка с четырьмя буквами ‘b’ (новая).

    Чтобы получить строку из всех ‘a’, каждую из особей для рекомбинации можно выбрать $\frac{n}{2}$ способами. Чтобы получить строку с четырьмя ‘b’, тоже. Ответ — $\frac{n^2}{2}$.

Для решения третьей подзадачи нужно было немного оптимизировать решение первой подзадачи. Давайте предварительно сохраним `unordered_set` генетические коды в первом поколении, и чтобы проверить, встречался ли полученный код среди кодов в первом поколении, будем использовать не линейный поиск, а запросы к `unordered_set`. Время работы такого решения будет $\mathcal{O}(n^3)$.

Для решения предпоследней подзадачи заметим, что чтобы получить какую-то строку q, являющуюся циклическим сдвигом s, в ходе рекомбинации, надо взять два циклических сдвига, у первого из которых символы на нечетных позициях совпадают с символами на нечетных позициях q, а у второго — аналогично с четными позициями.

Посчитаем полиномиальные хеши отдельно на четных позициях строки s, отдельно на нечетных. Соберем хеши всех циклических сдвигов s и положим их в `unordered_set`. Теперь, перебирая i и j для рекомбинации, будем вычислять соответствующие частичные хеши их четных/нечетных позиций и проверять, лежит ли их сумма в посчитанном множестве. Такое решение работает за $\mathcal{O}(n^2)$.

Полное решение требует обратного подхода. Заметим, что количество пар особей, дающих новый код, равно $n^2 - t$, где t — количество пар особей, дающих код из первого поколения. То есть можно найти количество пар, дающих уже существующий код, а затем вычесть это число из общего количества пар.

Для каждого уникального циклического сдвига строки s найдем количество способов его получить в ходе рекомбинации. Для этого, обратно предыдущему решению, заведем множество `cnt`, в котором для каждого хеша циклического сдвига четных/нечетных позиций s будем хранить количество раз, которое он встречается. Количество способов получить q, некоторый циклический сдвиг s, в таком случае равно $\texttt{cnt.count}(q_{\text{even}}) \cdot \texttt{cnt.count}(q_{\text{odd}})$.

Авторам также известно альтернативное полное решение задачи, использующее префикс-функцию. Для него обратимся к ключевой идее, указанной в решении подзадачи 2.

1. Количество уникальных циклических сдвигов s равно ее периоду p.
2. Количество способов получить строку s в ходе рекомбинации равно произведению количества способов выбрать циклический сдвиг с теми же четными позициями и количества способов выбрать циклический сдвиг с теми же нечетными позициями:
   - количество способов перевести четные символы s в четные сдвигом равно $m_{00} = \frac{n}{2p_0}$, где p_0 — период строки s_0 из четных символов s;
   - количество способов перевести нечетные символы s в нечетные, аналогично, равно $m_{11} = \frac{n}{2p_1}$, где p_1 — период строки s_1 из нечетных символов s;
   - количество способов перевести четные в нечетные и наоборот, равно 0, если s_0 и s_1 не являются циклическими сдвигами друг друга, и $m_{01} = m_{00} = m_{11}$ иначе.

Найти период строки можно с помощью префикс-функции ($p = n - \texttt{pf}[n]$, если он является делителем n, иначе $p = n$). Аналогично, с помощью алгоритма Кнута-Морриса-Пратта можно проверить, является ли строка s_1 циклическим сдвигом s_0. В конце останется просто посчитать ответ как

$$n^2 - (m_{00} + m_{01}) \cdot (m_{10} + m_{11}) \cdot p.$$

Задача D. Рейд на транспортер

Автор задачи и разработчик: Мария Жогова

Переформулируем: требуется выбрать последовательность (a_i, b_i) такую, чтобы a_i и b_i неубывали, и при этом соседние b_i отличались не более, чем на x. При этом можно добавить в множество произвольную пару (a, b), и требуется максимизировать размер итоговой последовательности.

Первая подгруппа решается произвольным перебором. Вторую и третью можно было решить, заметив, что если $x = 0$, то требование из условия превращается в требование на равенство всех b_i в итоговой последовательности. Таким образом, решение — сгруппировать все пары по b_i (например, с помощью `unordered_map`) и выбрать максимальную по размеру группу. Очевидно, что

- в любой группе можно взять все пары, просто упорядочив их по a_i;
- к любой группе можно добавить пару $(10^9, b_i)$, увеличив размер последовательности на 1.

Таким образом, ответ будет равен максимальному размеру группы плюс 1.

При ограничении, что все a_i равны, задача сводится к выбору максимальной последовательности, в которой выполняется условие на b_i. Отсортируем все пары по b_i, после чего заметим, что последовательность, заканчивающаяся в i-м элементе, может иметь в качестве предыдущего любой из отрезка $[j, i)$, где $b_j \geqslant b_i - x$. Таким образом, можно считать динамику $\texttt{dp}[i]$ как $\max\limits_{b_j \geqslant b_i - x} \texttt{dp}[j] + 1$. Это максимум на скользящем окне, его можно поддерживать с помощью очереди с максимумом.

Осталось только учесть, что произвольный (a, b) можно добавить в рассмотрение. Заметим, что такая пара может «склеить» максимальную последовательность на префиксе, заканчивающуюся в b_i, для которого $b_i \geqslant b - x$, и максимальную последовательность на суффиксе, начинающуюся в b_j, для которого $b_j \leqslant b + x$. Переберем i, заметим, что чтобы между b_i и b_j помещался b, необходимо и достаточно, чтобы выполнялось $b_j \leqslant b_i + 2x$, таким образом ответ можно найти как

$$\max_i \left(\texttt{dp}_{\text{pref}}[i] + \max_{j > i \wedge b_j \leqslant b_i + 2x} \texttt{dp}_{\text{suf}}[j] + 1 \right).$$

Опять же, данный максимум по динамике на суффиксах можно находить с помощью скользящего окна с очередью с максимумом.

Это подводит нас к полному решению. Будем также считать динамику на префиксах и суффиксах, но упорядочим пары по a_i. Ответ будет вычисляться по похожей формуле:

$$\max_i \left(\texttt{dp}_{\text{pref}}[i] + \max_{j > i \wedge b_i \leqslant b_j \leqslant b_i + 2x} \texttt{dp}_{\text{suf}}[j] + 1 \right).$$

Однако в этот раз необходимо находить максимум на суффиксе, с дополнительным условием на b_i. Это можно делать с помощью дерева отрезков или декартового дерева: будем поддерживать максимумы $\texttt{dp}$ для соответствующих b_i, и, итерируясь по i справа-налево, считать $\max\limits_{j > i \wedge b_i \leqslant b_j \leqslant b_i + 2x} \texttt{dp}_{\text{suf}}[j]$ запросом в ДО на отрезке $[b_i, b_i + 2x]$.

Суммарное время работы такого решения — $\mathcal{O}(n \log n + n \log 10^9)$, если использовать динамическое (неявное) дерево отрезков, и $\mathcal{O}(n \log n)$, если использовать декартово дерево.