

## Задача А. Сеты в архиве исследователя

Рассмотрим произвольные три карты. Чтобы они образовывали сет, необходимо и достаточно, чтобы для каждого из четырёх признаков выполнялось одно из двух условий: либо все три значения совпадают, либо все три попарно различны. Заметим, что признаки независимы друг от друга, поэтому проверка ведётся покомпонентно.

Так как  $n \leq 81$ , полного перебора всех троек карт, которых всего  $\binom{n}{3}$ , вполне достаточно по времени. Для каждой тройки последовательно проверяются все признаки. Если хотя бы по одному признаку значения устроены так, что среди них имеются ровно два равных и одно отличное, то данная тройка не является сетом. В противном случае тройка засчитывается.

Отдельно стоит отметить обработку формы. Во входных данных форма может быть записана в единственном или множественном числе в зависимости от количества фигур. Однако для проверки нас интересует лишь тип фигуры, а не грамматическое число, поэтому удобнее заранее привести значение к единому виду. Замечаем, что форма во множественном числе всегда образуется добавлением буквы *s* в конец слова, поэтому достаточно при чтении карты удалить эту букву, если она присутствует.

После унификации записей по всем признакам процедура проверки становится прямолинейной. Для каждого признака можно сравнить строки: тройка признаков либо совпадает полностью, либо состоит из трёх различных слов. Эти два случая легко отличить, проверив равенства и неравенства соответствующих строк. Если условие выполнено для всех четырёх признаков, тройка считается сетом.

Общая асимптотика решения равна  $O(n^3)$ , что укладывается в ограничение при  $n \leq 81$ . Реализация использует три вложенных цикла для перебора троек и прямую проверку признаков.

## Задача В. Аномалия степени сети

Чтобы определить, мог ли набор  $N$  чисел быть степенями некоторого простого неориентированного графа на  $N$  вершинах, достаточно проверить два типа условий. Во-первых, каждое число должно лежать в диапазоне от 0 до  $N - 1$ , а сумма всех чисел обязана быть чётной, так как она равна  $2|E|$ . Во-вторых, необходимо убедиться, что эти степени можно «раздать» вершинам так, чтобы каждая из них действительно получила нужное количество соседей.

Проверка проводится следующим образом. Отсортируем степени в невозрастающем порядке:  $d_1 \geq d_2 \geq \dots \geq d_N$ . Рассмотрим первую вершину, у которой степень  $d_1$ . Если она должна иметь  $d_1$  соседей, то естественно пытаться соединить её с вершинами, у которых степени наибольшие среди оставшихся, то есть со следующими  $d_1$  вершинами. Тогда их степени уменьшаются на 1, после чего первую вершину можно удалить из задачи. Если в какой-то момент оказывается, что  $d_1$  превосходит количество оставшихся вершин или у кого-то степень становится отрицательной, то данный набор не может быть степенной последовательностью графа.

Если же описанная процедура успешно отрабатывает до конца, значит нужный граф существует. Таким образом, чтобы найти все подходящие индексы, достаточно по каждому  $i$  удалить число  $a_i$ , проверить диапазоны, чётность суммы, отсортировать оставшиеся  $N$  чисел и попытаться последовательно «раздать» степени, уменьшая их, как описано выше.

Поскольку  $N \leq 500$ , прямой перебор всех  $N + 1$  вариантов и применение указанной проверки укладываются в ограничение по времени: сортировка занимает  $O(N \log N)$ , а сама проверка — линейное время, так что суммарно получается порядка  $O(N^2 \log N)$ .

Замечание. Используемая проверка является стандартной. В литературе она известна как проверка по неравенствам Эрдоса–Галлаи или как жадная процедура Хавела–Хакими, но для решения задачи достаточно понимания описанной интуиции: при каждой попытке разумнее всего соединять вершину с наибольшей степенью с вершинами, у которых степени тоже максимальны.

## Задача С. Игра жизни на круге

Ключ к задаче — заметить, что если в круге есть две последовательные единицы или два последовательных нуля, то их значение никогда не изменится. Поэтому нужно сосредоточиться на блоках чередующихся битов, окружённых с обеих сторон последовательными одинаковыми битами.

Рассмотрим различные случаи. Будем считать, что блок слева ограничен двумя одинаковыми битами  $X$ , а справа — двумя одинаковыми битами  $Y$ .

Случай 1:  $(X, Y) = (0, 0)$  и длина блока равна  $2n + 1$ . Например, 001010100. На каждом шаге число единиц уменьшается на один, и после  $n + 1$  шагов все единицы исчезнут. Таким образом, после  $T$  обновлений число единиц равно  $\max(n + 1 - T, 0)$ .

Случай 2:  $(X, Y) = (1, 1)$  и длина блока равна  $2n + 1$ . Например, 110101011. На каждом шаге число единиц увеличивается на один, и после  $n + 1$  шагов все нули исчезнут. Следовательно, после  $T$  обновлений число единиц равно  $\min(n + T, 2n + 1)$ .

Случай 3:  $X \neq Y$  и длина блока равна  $2n$ . Например: 11010100. Тогда после каждого шага число единиц не меняется. Происходит следующее: одна из единиц «приближается» к последовательным единицам, и пример переходит в 11101000 после первого шага, затем в 11110000 после второго, и структура больше никогда не изменится.

Повторим нашу строку (закольцуем). Далее введём структуру  $\text{edge}[i][b_1][j][b_2]$ ,  $0 \leq b_1, b_2 \leq 1$ ,  $0 \leq i < j \leq 2N$ , определённую следующим образом. Пусть  $s' = \text{start}[i \dots j - 1]$ .

Тогда:

-  $\text{edge}[i][b_1][j][b_2] = -1$ , если невозможно заменить символы ? в  $s'$  так, чтобы получить чередующийся блок, начинающийся с  $b_1$  и заканчивающийся  $b_2$ ;

- иначе  $\text{edge}[i][b_1][j][b_2]$  равно числу единиц в  $s'$  после  $T$  обновлений, предполагая, что блок окружён слева битом  $b_1$ , справа — битом  $b_2$ , и первые и последние биты не могут изменяться.

Иными словами,  $\text{edge}[i][b_1][j][b_2]$  отвечает на вопрос: можно ли «перейти» от позиции  $i$  к позиции  $j$  с помощью блока чередующихся битов, и если да — сколько единиц этот переход «потребляет» после  $T$  шагов.

Общий план алгоритма: перебрать все возможные пары  $(\text{startpos}, \text{startbit})$ , где  $\text{startpos}$  — первая позиция в исходной строке, где  $s[\text{startpos}] = s[(\text{startpos} + N - 1) \% N]$ , а  $\text{startbit}$  — бит, который подставляем в  $s[\text{startpos}]$ . Для каждой такой пары добавляем к ответу значение, вычисленное при данном предположении.

Фиксировав  $(\text{startpos}, \text{startbit})$ , определим динамику:

$dp[\text{pos}][\text{bit}][\text{ones}]$  — количество способов построить последовательность битов  $q$  из отрезка  $s[\text{startpos} \dots \text{pos} - 1]$ , при условии что  $s[\text{pos}] = \text{bit}$ , и что после  $T$  обновлений в  $q$  будет ровно  $\text{ones}$  единиц.

Базовый случай:  $dp[\text{startpos}][\text{startbit}][0] = 1$ .

Для заполнения остальных значений используем следующее. Если  $\text{edge}[i][i_2][j][j_2] \neq -1$ , то можно «расширить» любое состояние  $dp[i][i_2][\text{one}]$  до состояния  $dp[j][j_2][\text{one}2]$ , где  $\text{one}2 = \text{one} + \text{edge}[i][i_2][j][j_2]$ .

Также важно учитывать, что  $\text{startpos}$  — первая позиция, где встречаются два совпадающих бита, и это должно оставаться выполненным. Нам также следует рассмотреть особый случай. Если в круге нет последовательных единиц или нулей, то есть ячейки имеют вид 1010...10 или 0101...01, то обновление просто перевернёт биты (единицы станут нулями, а нули — единицами), а количество единиц не изменится. В этом случае  $K \leq N/2$ .

## Задача D. Мутационная последовательность

В задаче требуется превратить строку  $S$  в строку  $T$  с помощью не более чем трёх операций. Каждая операция выбирает одну букву из  $A, B, C$  и заменяет все её вхождения на некоторую строку длины от 1 до 3, состоящую из тех же букв.

Поскольку длины исходных строк не превышают 10, а гарантируется, что оптимальное решение существует и использует не более 3 операций, пространство перебора оказывается очень небольшим. Можно просто осуществить перебор по всем возможным операциям и остановиться, как только удастся получить строку  $T$ .

Идея перебора проста. На каждом шаге выбирается буква  $x \in A, B, C$ , затем выбирается строка  $y$  длиной от 1 до 3, и выполняется замена всех букв  $x$  в текущей строке на  $y$ . После этого рекурсивно продолжаем поиск. Глубина рекурсии ограничивается числом 3. Если в какой-то момент строка совпала с  $T$ , найденная последовательность операций является решением. Если длина промежуточной строки стала больше длины  $T$ , дальнейший поиск можно прекращать, так как замены только увеличивают длину или оставляют её прежней.

Для перебора строк-замен удобно заранее породить все строки над алфавитом  $A, B, C$  длины 1, 2 и 3. Их всего  $3 + 9 + 27 = 39$ , что позволяет перебрать все варианты мгновенно.

Полное число состояний мало: глубина — 3, на каждом шаге три возможных исходных символа и около сорока вариантов замен, так что полный перебор выполняется крайне быстро. Как только найдено совпадение с  $T$ , можно выводить последовательность операций и завершать работу.

## Задача Е. Построение дерева кабелями

Произвольно подвесим дерево и рассмотрим пайку как «отрезание» некоторого поддерева. Это оставляет один (или ни одного) «отрезанный провод», который выходит из поддерева к родителю, и набор проводов, полностью лежащих внутри поддерева. Важно, что значимыми являются только длина «отрезанного провода» внутри поддерева и суммарная стоимость всех остальных проводов. Это связано с тем, что именно отрезанный провод — единственный, чья стоимость зависит от остальной структуры пайки.

Это приводит к относительно простому решению динамическим программированием: для каждой вершины (определяющей поддерево) храним для каждой возможной длины отрезанного провода минимальную стоимость остальных проводов; если отрезанного провода нет, можно считать, что имеется провод длины 0. Мы вычисляем эти значения снизу вверх. Чтобы вычислить эти величины, заметим: если есть отрезанный провод, он обязан спускаться в одного из детей; стоимость отрезанного провода, проходящего через конкретного ребёнка, равна стоимости такого же провода в его поддереве плюс минимальные стоимости пайки, покрывающие остальные поддерева. Если же отрезанного провода нет, то ребро, ведущее к родителю, должно быть припаяно к середине другого провода; тогда можно просто проверить все пары длин и различных детей, чтобы найти две «отрезанные ветви», которые будут слиты в один провод. Максимальная возможная длина отрезанного провода для данного поддерева — число вершин в нём; отсюда число пар длин для двух различных детей не превосходит числа пар вершин в этих двух поддеревьях; суммируя по всем детям, это даёт число пар вершин, для которых наименьший общий предок — корень поддерева. Тогда вся работа алгоритма — величины порядка  $O(N^2)$ .

На этом этапе удобно считать, что каждая вершина имеет не более двух детей. Это не представляет проблемы: вершину  $V$  с более чем двумя детьми можно «разбить», дав ей прямое ребро к одному из детей, а остальных присоединив к новой вершине  $V'$  с ребром длины 0 (длина не нарушает алгоритм, хотя по условию все рёбра имели длину 1). Повторяя это, добиваемся, что у всех вершин не более двух детей.

Чтобы ещё уменьшить время работы, нужно заметить свойства выпуклости функции квадрата длины. Если рассматривать пару (длина, стоимость)  $(l, c)$  для поддерева, она соответствует функции

$$(L + l)^2 + c,$$

где  $L$  — длина отрезанного провода вне поддерева. Но нас интересуют только те функции, которые минимальны при некоторых значениях  $L$ : поскольку

$$(L + l)^2 + c = L^2 + 2Ll + (l^2 + c),$$

мы получаем нижнюю оболочку этих функций, эквивалентную выпуклой оболочке. Все пары, которые не входят в оболочку, можно отбросить. Тогда можно бинарным поиском по выпуклой оболочке находить оптимальное сочетание для заданной длины провода вне поддерева. Чтобы найти оптимальные пары длин двух детей, которые будут слиты в один провод, можно взять все длины из меньшего поддерева и для каждой бинарным поиском найти лучшую пару в выпуклой оболочке большего поддерева.

Чтобы эффективно вычислять выпуклые оболочки для всех поддеревьев, можно представлять их с помощью бинарных деревьев поиска (например, `std::set`). Для получения возможных значений от поддеревьев детей можно сдвинуть оболочку большего поддерева (путём хранения смещений, которые добавляются ко всем парам, так как и длина, и стоимость меняются при объединении поддеревьев), а затем вставлять элементы из меньшего поддерева (использовать приливайку). Общее число операций с деревьями поиска не превосходит суммы размеров меньших дочерних поддеревьев

для всех вершин (а может быть меньше, так как выпуклая оболочка содержит меньше элементов, чем поддереву).

Это можно оценить как  $O(N \log N)$ : каждая позиция объединяется с более крупной группой не более  $\log N$  раз, так как при каждом объединении инкрементируются только значения в меньшей половине. Каждая операция в дереве имеет стоимость  $O(\log N)$ , так что итоговое время работы —  $O(N \log^2 N)$ .

## Задача F. Восстановите холст Севы

Сначала заметим, что все команды `SAVE` и `LOAD` можно удалить за  $O(M)$  времени. Для этого обрабатываем последовательность команд в обратном порядке, начиная с последней и двигаясь к первой. Каждый раз, когда встречается команда `LOAD`, можно игнорировать все команды между этой `LOAD` и соответствующей `SAVE`. После такой оптимизации последовательность команд сводится только к командам `PAINT`.

После этого задачу можно решать с помощью алгоритма сканирующей прямой. Проходим по холсту построчно. Для удобства заводим два дерева отрезков: одно для черных клеток, другое для белых. Красное дерево хранит шаблоны шахматки, в которых закрашены чётные клетки текущей строки, а белое дерево — для нечётных клеток. При переходе к следующей строке деревья меняются местами. Вершины дерева содержат наборы узоров, покрывающих соответствующие отрезки, отсортированные по времени закрашивания.

Вставка или удаление узора в дерево выполняется за  $O(\log N \cdot \log M)$ , а запрос цвета для конкретной клетки — за  $O(\log N)$ . Общая асимптотика алгоритма получается  $O(N^2 \cdot \log N + M \cdot \log N \cdot \log M)$ .

Альтернативно, задачу можно решить с помощью СММ за  $O(N \cdot M)$ . Идея заключается в следующем. Рассматриваем клетки холста отдельно для чётных и нечётных позиций шахматной раскладки, так как команда `PAINT` закрашивает прямоугольник в шахматном порядке. Для каждой строки создаём отдельный объект `DSU` для чётных и нечётных клеток, чтобы быстро переходить к следующей незакрашенной клетке в прямоугольнике.

Обработка команд производится в обратном порядке, начиная с последней команды. Это позволяет корректно учитывать последующие перекрытия прямоугольников. При встрече команды `SAVE` или `LOAD` используется вспомогательный массив, чтобы пропустить все команды до соответствующей сохранённой версии. Таким образом, каждая команда `PAINT` обрабатывается только один раз, когда её клетки ещё не были закрашены более поздними командами.

Алгоритм для конкретной команды `PAINT` с `x1 y1 x2 y2` следующий. Определяем, чётная ли стартовая клетка  $(x_1 + y_1) \bmod 2$ , и выбираем соответствующий `DSU`. Затем проходим по каждой строке прямоугольника и с помощью операции `get` находим первую незакрашенную клетку. Краской команды закрашиваем все последовательные клетки до правой границы прямоугольника, объединяя их в одно множество через `unite`, чтобы при следующей итерации быстро переходить к следующей незакрашенной клетке.

## Задача G. Большой сюрприз

Если лететь на фиксированной высоте  $h$ , то мешают только здания с высотой  $> h$ . Путь на уровне  $h$  имеет длину

$$|z_S - h| + |z_F - h| + D_{xy}(h),$$

где  $D_{xy}(h)$  — кратчайший путь в плоскости  $xy$ , обходя проекции зданий высотой  $> h$ .

Оптимальная высота  $h$  всегда принадлежит множеству:

$$h \in \{\max(z_S, z_F)\} \cup \{h_i : h_i > \max(z_S, z_F)\},$$

то есть либо  $\max(z_S, z_F)$ , либо высоты зданий.

На фиксированном  $h$  остаются препятствия — прямоугольники зданий с высотой  $> h$ .

В манхэттенской геометрии кратчайший путь вокруг осевых прямоугольников “ломается” только в их углах. Значит, строим граф из:

- четырёх углов каждого здания с  $h_i > h$ ,

- стартовой точки  $S$ ,
- (конечная  $F$  проверяется отдельно).

Между двумя точками проводим ребро, если прямой манхэттенский путь между ними не пересекает препятствий.

Вес ребра:  $|x_1 - x_2| + |y_1 - y_2|$ .

По этому графу запускаем Дейкстру и получаем  $D_{xy}(h)$ .

Перебираем все подходящие  $h$ , считаем

$$L(h) = |z_S - h| + |z_F - h| + D_{xy}(h),$$

берём минимум.

Количество уровней  $O(n)$ , вершин  $\leq 4n$ , Дейкстра на полном графе  $O(n^2 \log n)$ . Итого  $O(n^3 \log n)$ .

## Задача Н. Контрольные ворота паспортного контроля

Основная идея решения заключается в том, что можно идти «от конца к началу»: рассматриваем туристов, которые исчезли между первым и вторым изображениями, начиная с самого старшего. Для каждого такого туриста известно, в какой очереди он стоял, поэтому мы знаем, через какие ворота он мог пройти. Так как выбирается старший турист из двух очередей, старший турист из очереди  $i$  может пройти через ворота  $i$  или  $i + 1$ , если он старше всех, кто стоит перед ним в соседней очереди. Для крайних очередей (первая и последняя) выбор ворот однозначен.

Будем хранить туристов, которые должны пройти ворота, в множестве пар (номер туриста, номер очереди) в порядке убывания номера туриста, чтобы всегда обрабатывать самого старшего туриста, который ещё остался. В множестве будем хранить только первых текущих в очереди, если несколько человек должны пройти ворота из этой очереди. Для каждого туриста проверяется, через какие ворота он может пройти, исходя из состояния соседних очередей. Если ни одни ворота не удовлетворяют условию, выводится “Impossible”. Иначе добавляем соответствующие ворота в результат и удаляем туриста из очереди. После удаления проверяем, есть ли ещё турист, который должен пройти ворота из этой очереди, и добавляем их в множество для обработки.

## Задача I. Жадный термит

Термит переходит от съеденного стержня  $i$  к тому стержню  $j$ , который максимизирует величину  $h_j - |x_i - x_j|$ . Если несколько кандидатов дают один и тот же максимум, выбирается кандидат с наименьшим  $|x_i - x_j|$ , а при новом равенстве — с наименьшим номером.

Для фиксированного  $i$  выражение  $h_j - |x_i - x_j|$  удобно рассматривать по сторонам: при  $j < i$  оно равно  $(h_j + x_j) - x_i$ , при  $j > i$  оно равно  $(h_j - x_j) + x_i$ . Поскольку сдвиг на  $\pm x_i$  одинаков для всех кандидатов одной стороны, для выбора максимума достаточно знать по левым индексам максимум  $h_j + x_j$ , а по правым — максимум  $h_j - x_j$ . После сравнения этих двух значений (с учётом вторичных критериев) однозначно определяется следующий индекс.

Это позволяет поддерживать данные в дереве отрезков, в каждой вершине которого храним максимум по величине  $h_j + x_j$  (с привязкой к индексу) и максимум по величине  $h_j - x_j$ . При удалении стержня его позицию заменяют нейтральным значением, которое никогда не выиграет в сравнении. Тогда каждый выбор следующего стержня сводится к паре запросов и выполняется за  $O(\log n)$ , а весь процесс занимает  $O(n \log n)$ . Осталось лишь суммировать пройденные расстояния  $|x_i - x_j|$  при каждом переходе.

## Задача J. План Б

Надо найти все вершины  $v$  такие, что при удалении  $v$  существует соседняя ей компонента связности, в которой нет ни одной военной базы. Действительно, город  $v$  можно «окружить» тогда и только тогда, когда для каждой его соседней вершины  $u$  существует путь от какой-нибудь базы до  $u$ , не проходящий через  $v$ . Это равносильно тому, что в графе  $G - v$  каждая компонента, содержащая хотя бы одного соседа  $u$  вершины  $v$ , содержит также хотя бы одну базу.

Чтобы проверку выполнять быстро для всех  $v$ , удобно разложить граф по компонентам, которые получаются при удалении точек сочленения.

Сначала выполняем DFS и находим точки сочленения, а затем строим граф блоков-точек сочленения <https://shorturl.at/IM2cU> (можно посмотреть, например, на викиконспектах). В нём есть вершины двух типов — исходные вершины графа и компонентные вершины; ребро соединяет исходную вершину и компоненту тогда и только тогда, когда исходная вершина принадлежит этой компоненте. Это дерево имеет размер  $O(n + m)$  и отражает разделение графа на участки, независимые при удалении точек сочленений.

Пометим в этом дереве компонентные вершины, в которых есть хотя бы одна военная база. Затем делаем DFS по дереву и для каждой вершины дерева вычисляем сумму пометок в её поддереве (число баз в соответствующей части). Теперь для исходной вершины  $v$  достаточно проверить все её смежные в дереве компонентные вершины: если какая-то компонентная вершина прилегает к  $v$  и в её поддереве суммарно нет баз, то при удалении  $v$  эта часть останется без баз, а значит  $v$  критична. Кроме того, для составляющей «сверху» (компонента, содержащей остаток дерева вне выбранного поддерева) нужно проверить, есть ли в ней базы: это учитывается той же суммой баз в поддереве и общим числом баз  $b$ .

Если у самой вершины  $v$  есть база, то по условию она не считается критической и сразу отбрасывается.

Все описанные шаги — построение DFS, сбор компонент и вычисление сумм по дереву — выполняются за  $O(n + m)$  времени и  $O(n + m)$  памяти. Итог: перебрав все исходные вершины и применив описанную локальную проверку по смежным компонентам в дереве, получаем искомым список критических городов.

## Задача К. Лодка

Ключевое наблюдение заключается в том, что оптимально всегда стараться отправлять вместе самого лёгкого и самого тяжёлого человека, если их суммарный вес не превышает  $w$ . Если такой пары составить нельзя, то тяжёлый человек идёт один. После сортировки весов по возрастанию можно поддерживать два указателя: один указатель на самого лёгкого, другой — на самого тяжёлого. Если  $w_{\text{лёгкий}} + w_{\text{тяжёлый}} \leq w$ , перевозит их вместе и сдвигаем оба указателя; иначе перевозим только тяжёлого и сдвигаем указатель тяжёлого.

После каждой поездки лодка возвращается за следующими пассажирами. Если  $h$  — количество жителей, которых можно перевозить в паре с самым лёгким, то оставшиеся  $n - h$  жителей потребуют индивидуальных перевозок. Общее минимальное количество рейсов, учитывая обе стороны, можно выразить формулой  $4h + 2(n - h) - 3$ , где  $-3$  корректирует первый и последний переход, так как первая поездка не требует возвращения за пассажирами.

## Задача Л. ПВО

Граф «кто кого защищает» образует лес: вершины  $0 \dots n - 1$  — ресурсы с ценностью 1, вершины  $n \dots n + d - 1$  — ПВО с ценностью 0, рёбра направлены от ПВО к защищаемым объектам. Корни деревьев — объекты, которых никто не защищает. По этому лесу посчитаем динамику по поддеревам. Для каждой вершины  $v$  строится массив  $dp_v[k]$ , который хранит максимум ресурсов, которые можно уничтожить в поддереве с корнем  $v$ , если потратить  $k$  ракет. Для ресурса  $v$  значение стоимости  $cost[v] = 1$ , для ПВО  $cost[v] = 0$ . При этом, чтобы воспользоваться детьми ПВО, её необходимо уничтожить, то есть потратить хотя бы одну ракету.

Объединение динамик детей реализуем аккуратно: если у поддерева  $u$  дп задана массивом  $dp_u[j]$ , а у вершины  $v$  уже есть массив  $dp_v[i]$ , то сформируем новый массив  $dp[i + j] = \max(dp[i + j], dp_v[i] + dp_u[j])$ . Идея в том, что если мы потратили  $i$  ракет на одно поддерево и  $j$  на другое, суммарно это  $i + j$  ракет, а ресурсов уничтожено  $dp_v[i] + dp_u[j]$ . Таким образом, объединение будет работать за сумму размеров детей. Если мы будем последовательно приливать таким образом детей, и после приливания очередного ребенка пересчитывать размер дерева, сложность вычисления динамики будет  $O(n^2)$ .

После обхода всех деревьев корни объединяются через такую же операцию, чтобы получить общий дп для всего леса. Ответом является  $dp_{\text{общее}}[k]$ , где  $k = \min(m, \text{максимальное число состояний})$ , то есть максимум ресурсов, которые можно уничтожить, имея не более  $m$  ракет.

## Задача М. Три этапа теста

Прямой перебор всех троек занимает  $O(N^3)$ . Чтобы ускорить решение, заметим, что при сортировке дронов по убыванию  $B_i$  максимумы  $\max(B_i, B_j)$  и  $\max(B_j, B_k)$  всегда корректно учитываются при последовательном проходе. Это позволяет свести задачу к динамическому накоплению минимальных сумм.

Пусть  $ms1$  — минимальное значение  $A_i + B_i$  среди первых рассмотренных дронов,  $ms2$  — минимальная сумма для пары дронов  $i, j$  в порядке прохождения этапов,  $ms3$  — минимальная сумма для тройки дронов  $i, j, k$ . При проходе по дронам обновляем эти значения:

$$ms1 = \min(ms1, A_i + B_i), ms2 = \min(ms2, ms1 + A_j + B_j), ms3 = \min(ms3, ms2 + A_k).$$

После обработки всех дронов  $ms3$  даёт минимальное возможное время теста. Сортировка обеспечивает, что  $\max(B_i, B_j)$  и  $\max(B_j, B_k)$  корректно учитываются, а последовательное накопление минимальных сумм гарантирует, что мы не упустим оптимальную комбинацию.

Таким образом, задача сводится к сортировке дронов по убыванию  $B_i$  и последовательному обновлению  $ms1, ms2, ms3$ , что даёт сложность  $O(N \log N)$ .

## Задача N. Лингвистическая дуэль

Наиболее заметная особенность игры состоит в том, что слово с наименьшим рангом  $L$  побеждает слово с наивысшим рангом  $H$ . Можно ли использовать эту нерегулярность? Предположим, что у нас есть способ узнать, какие слова являются  $L$  и  $H$ . Тогда мы могли бы зарабатывать очко в каждом ходу, повторяя следующий цикл:

1. Мы играем  $H$ , заставляя ИИ сыграть  $L$  в ответ.
2. Мы играем произвольное слово, не равное  $H$  или  $L$ , побеждая  $L$  и получая очко. ИИ отвечает другим словом  $W$ .
3. Мы бьём  $W$  словом  $H$ , снова заставляя ИИ сыграть  $L$ , и так далее. (Если вдруг  $W$  окажется равным  $H$ , что маловероятно, мы играем вместо этого  $L$ , и ИИ отвечает словом  $W'$ , и цикл продолжается.)

Таким образом, если мы уверенно можем найти  $L$  и  $H$  менее чем за 50 ходов, мы точно сможем решить задачу.

Есть лишь два слова, после которых у ИИ остаётся только один возможный ответ: если мы играем слово со вторым по величине рангом, ИИ должен сыграть слово с наивысшим рангом; если же мы играем слово с наивысшим рангом, то ИИ обязан сыграть слово с наименьшим рангом.

Следовательно, когда ИИ присылает нам слово  $W$ , мы можем отправить два подряд слова  $W$  и посмотреть на два ответа. Если ответы отличаются, мы берём первый из них и также отправляем его дважды, и так далее. Если же ответы совпадают, мы можем отправить  $W$  ещё несколько раз, чтобы убедиться, что ответ всегда один и тот же  $R$ . Тогда можно отправлять  $R$  несколько раз, чтобы увидеть, получаем ли мы всегда одинаковый ответ  $S$  (в этом случае  $R$  — второе по величине слово, а  $S$  — самое высокое) или разные ответы (в этом случае  $R$  — самое высокое слово, а  $S$  — самое низкое).