Задача A. Рекордные ДОД

Заметим, что первое условие эквивалентно неравенству $v_i > \max_{1 \leqslant j < i} v_j$. Для проверки этого условия при одном проходе по массиву достаточно хранить текущий максимум $mx = \max_{1 \leqslant j < i} v_j$. Изначально $mx = -\infty$ (или $mx = -1$, так как $v_i \geqslant 0$).

Алгоритм: проходим по дням $i = 1 \dots n$. Для каждого дня проверяем два условия: $v_i > mx$ и (либо $i = n$, либо $v_i > v_{i+1}$). Если оба выполняются, увеличиваем счётчик рекордных дней. После проверки обновляем $mx = \max(mx, v_i)$.

Временная сложность алгоритма $O(n)$, память $O(1)$ (помимо хранения входных данных).

Корректность. По индукции: при проходе до позиции i переменная mx действительно равна максимуму посещений за предыдущие дни. Тогда тест $v_i > mx$ корректно проверяет первое требование. Второе требование проверяется напрямую по v_{i+1} (или по условию $i = n$). Поэтому оба условия корректно проверяются и день отмечается как рекордный тогда и только тогда, когда они выполнены.

Задача B. Двуцветный путь

Первое наблюдение связано с чётностью длины пути. Пусть s — число посещённых клеток. Равенство числа белых и чёрных требует, чтобы s было чётным. Путь, соединяющий две клетки, состоит из t шагов, где $s = t + 1$. Следовательно, требование «s чётно» эквивалентно требованию, что начальная и конечная клетки имеют разные чётности суммы координат:

$$(sr + sc) \bmod 2 \neq (er + ec) \bmod 2.$$

Если эти чётности равны, путь невозможен, и ответ равен -1.

Далее рассматривается построение пути при допустимой чётности. Ключевое наблюдение заключается в наличии на доске локальных «регуляторов»: если есть хотя бы одна пара соседних белых клеток и хотя бы одна пара соседних чёрных клеток, можно регулировать число посещений каждого цвета. Конструктивно это делается так: от стартовой клетки идём к одной паре, «накручиваем» нужное количество посещений, затем идём к паре другого цвета, «накручиваем» посещения и продолжаем путь к финишу. Таким образом, суммарное число белых и чёрных посещений можно сделать равным.

Если на доске нет ни одной пары соседних белых клеток и ни одной пары соседних чёрных клеток, сетка по сути окрашена как шахматная доска. В этом случае на любом простом пути цвета идут попеременно. Для клеток разной чётности существует путь с нечётным числом шагов, а значит с чётным числом посещений; любой простой путь от старта к финишу удовлетворяет условию.

Если же есть пара соседних одинаковых клеток только одного цвета, то абсолютная разница между числами посещённых белых и чёрных клеток на любом пути не превышает 1. В этой ситуации применяется BFS в пространстве состояний (r, c, d), где d — текущая разность числа чёрных и белых посещений, инициализированная смещением. Путь существует тогда и только тогда, когда достигнуты координаты финиша с $d = 0$. Поскольку $d \in \{-1, 0, 1\}$, размер пространства состояний остаётся $O(n \cdot m)$.

Реконструкция пути осуществляется стандартно: при переходах по состояниям запоминается предшественник и направление хода. Достигнув состояния $(er, ec, 0)$, восстанавливается последовательность шагов.

Оценка времени и памяти: BFS по состояниям (r, c, d) с $d \in \{-1, 0, 1\}$ требует $O(n \cdot m)$ времени и памяти. В случаях шахматной раскраски или наличия обеих пар соседних одинаковых клеток BFS выполняется по обычным координатам, что также укладывается в $O(n \cdot m)$.

Корректность алгоритма обеспечивается следующими наблюдениями: проверка чётности исключает невозможные случаи; наличие соседних пар обоих цветов обеспечивает возможность регулировки счёта; при наличии пары только одного цвета разность ограничена, и BFS на трёх слоях надёжно находит путь с равным числом белых и чёрных посещений. Таким образом метод покрывает все возможные конфигурации сетки и укладывается в заданные ограничения по времени и памяти.

Задача C. Кубики в коробке

Заметим, что существует **верхняя граница** для минимального объёма, который нам потребуется. В условии задачи прямо сказано, что ответ помещается в 32-битное знаковое целое число. Быстрая проверка ограничений подтверждает это:

$$10^6 \text{ ящиков размера } 10 \times 10 \times 10 \Rightarrow 10^9 \text{ единиц объёма,}$$

и ещё 10^9 ящиков размера $1 \times 1 \times 1$ дают столько же. Следовательно, максимальный объём около $2 \cdot 10^9$, что помещается в 32-битный диапазон. Этот максимум достигается, если все $10 \times 10 \times 10$ ящики выстроить в линию, а затем увеличивать высоту до тех пор, пока не поместятся все остальные ящики. Следовательно, для любой конфигурации выполняется:

$$x \cdot y \cdot z \leqslant LIMIT \approx 2 \cdot 10^9.$$

Выбор размеров

Пусть (x, y, z) — размеры коробки. Объём:

$$V = x \cdot y \cdot z \leqslant LIMIT.$$

Сколько различных трёхмерных коробок существует? Даже если для каждого объёма выбирать единственную коробку, количество таких троек (x, y, z) очень велико. Поэтому попробуем фиксировать только два параметра — размеры основания (x, y), а затем минимизировать высоту z.

Оптимизации по симметрии

Перед перебором всех пар (x, y) отметим:

- Пары $(x = 2, y = 4)$ и $(x = 4, y = 2)$ дают одинаковую минимальную высоту, поэтому можно предполагать $x \leqslant y$.
- Всегда можно предположить $x, y \leqslant z$, иначе меньшую из трёх величин можно было бы перенести в основание.
- Следовательно:
$$x \cdot x \cdot x \leqslant LIMIT, \qquad x \cdot y \cdot y \leqslant LIMIT.$$

Из первого условия получаем $O(LIMIT^{1/3})$ возможных значений x. Из второго условия следует $y^2 \leqslant LIMIT/x$, что даёт $O(\sqrt{LIMIT/x})$ значений y. Суммарно число пар (x, y) порядка:

$$O\Big(LIMIT^{2/3}\Big),$$

что достаточно быстро, если минимальная высота z вычисляется за $O(1)$.

Вычисление минимальной высоты z за $O(1)$

Пусть даны размеры основания (x, y). Нужно разместить N_b кубов размера $L \times L \times L$ и N_s кубов $1 \times 1 \times 1$.

1. Если $L > x$ или $L > y$, то ни один большой куб не поместится — такой случай пропускаем.
2. Иначе найдём максимально возможные $b_x = \lfloor x/L \rfloor$, $b_y = \lfloor y/L \rfloor$. Нужно подобрать b_z:
$$b_x \cdot b_y \cdot b_z \geqslant N_b, \quad b_z = \left\lceil \frac{N_b}{b_x \cdot b_y} \right\rceil.$$
Высота, необходимая для больших кубов:
$$z_{\text{big}} = L \cdot b_z.$$

3. Объём, занятый большими кубами:

$$V_{\text{big}} = N_b \cdot L^3.$$

Оставшийся объём:

$$V_{\text{free}} = x \cdot y \cdot z_{\text{big}} - V_{\text{big}}.$$

Если $V_{\text{free}} < N_s$, нужно увеличить высоту на:

$$w = \left\lceil \frac{N_s - V_{\text{free}}}{x \cdot y} \right\rceil.$$

Общая высота:

$$z = z_{\text{big}} + w.$$

Искомый объём:

$$V = x \cdot y \cdot z.$$

Итоговый алгоритм

1. Выбрать верхнюю границу $LIMIT$ (можно взять $2 \cdot 10^9$).
2. Перебрать все пары (x, y), такие что $x \cdot x \cdot x \leqslant LIMIT$ и $x \cdot y \cdot y \leqslant LIMIT$, при $x \leqslant y$.
3. Для каждой пары вычислить минимальную высоту z по описанной формуле за $O(1)$.
4. Минимизировать $V = x \cdot y \cdot z$.

Задача D. Наиболее значимый шаблон

Ключевое наблюдение заключается в том, что каждая подстрока $s[i..j]$ может быть однозначно сопоставлена с *паттерном* — нормализованной формой, полученной последовательным перенумерованием различных букв в порядке их первого появления. Например, подстрока «cabb» и «xyzz» имеют один и тот же паттерн «abcc». Эквивалентность подстрок сводится к совпадению их паттернов. Таким образом, задача сводится к поиску такого паттерна, который встречается максимально часто и при этом даёт наибольшее значение произведения количества вхождений на длину.

Прямой перебор всех подстрок требует $O(n^2)$ шагов, так как их всего $\frac{n(n+1)}{2}$, что для $n = 4000$ составляет около восьми миллионов подстрок — достаточно много, но вполне допустимо. Основная трудность состоит в том, чтобы быстро группировать эквивалентные подстроки. Для этого строится компактный ключ, однозначно определяющий паттерн. Пусть при фиксированном левом конце i мы идём вправо и поддерживаем массив соответствий `mapChar` длины 26, который хранит, каким порядковым номером кодируется каждая буква. При встрече новой буквы ей присваивается следующий код $0, 1, 2, \ldots$ и код прибавляется в конец хеша. Хеш можно строить инкрементально, например, по модулю 2^{64} с помощью базы B:

$$h \leftarrow h \cdot B + (\text{код} + 1).$$

Чтобы различать подстроки разной длины, в итоговый 64-битный ключ упаковывается и значение хеша, и длина подстроки. Таким образом, каждой подстроке соответствует пара $(\text{ключ}, i)$, где i — позиция начала.

После построения всех пар их необходимо отсортировать по ключам. Сортировка сравнением (`std::sort`) на стольких элементах будет работать слишком медленно, поэтому используется поразрядная сортировка (radix sort) по 64-битным ключам с шагом 16 бит, что даёт всего четыре линейных прохода. Такая сортировка стабильна и работает за $O(n^2)$ с небольшой константой. После сортировки все эквивалентные подстроки оказываются подряд, поэтому достаточно одного линейного прохода, чтобы подсчитать количество элементов в каждой группе и вычислить значение $a \cdot |q|$. Ведётся учёт максимума, а также позиции и длины, на которых этот максимум достигается.

Для восстановления ответа берётся подстрока $s[i..i + |q| - 1]$, соответствующая лучшему паттерну, и снова проводится нормализация, чтобы вывести именно паттерн, а не исходную подстроку.

Полученный результат является корректным решением, так как любая другая подстрока с тем же ключом даёт такой же паттерн и такую же значимость.

Асимптотика решения равна $O(n^2)$ как по времени, так и по памяти. По времени алгоритм состоит из построения всех ключей, четырёх проходов поразрядной сортировки и линейного сканирования, каждая из стадий занимает $O(n^2)$. По памяти хранятся два массива длины $O(n^2)$: массив ключей типа `uint64_t` и массив стартовых позиций типа `uint32_t`, а также временные буферы для сортировки. Для $n = 4000$ это около 180 мегабайт, что укладывается в лимит 256 мегабайт. Использование хеширования по модулю 2^{64} делает вероятность коллизий пренебрежимо малой; при желании можно дополнить двойным хешем для полной детерминированности.

Задача E. Нумерация домов

Ключевая идея состоит в том, что ответ **монотонен**: если у нас достаточно цифр для построения номеров домов от 1 до x, то у нас тем более достаточно цифр для построения номеров домов от 1 до любого $y < x$. Это позволяет найти оптимальный ответ с помощью **двоичного поиска**.

Проверочная функция

Для реализации двоичного поиска нам нужна функция, которая принимает число n и проверяет, достаточно ли цифр в нашей «коробке», чтобы составить числа от 1 до n. Эта функция должна посчитать, сколько раз каждая цифра встречается в этих числах, и сравнить полученные значения с количеством соответствующих цифр в коробке.

Подсчёт вхождений цифр

Определим $C_{d,n}$ — количество раз, которое цифра d встречается в числах от 0 до $n-1$. Эту величину можно вычислить рекурсивно за $O(\log n)$ следующим образом:

- Если $n = 0$, то $C_{d,0} = 0$.
- Если $n \bmod 10 \neq 0$, то

$$C_{d,n} = C_{d,n-1} + \text{количество вхождений } d \text{ в числе } n-1.$$

- Если $n \bmod 10 = 0$, мысленно перечислим все числа от 0 до $n-1$:
  1. Рассмотрим последнюю цифру. Она пробегает значения $0, 1, \ldots, 9$ ровно $\frac{n}{10}$ раз. Следовательно, на последнем разряде цифра d встречается ровно $\frac{n}{10}$ раз.
  2. Теперь «стираем» последнюю цифру. Что остаётся? Последовательность чисел $1, 2, \ldots, \frac{n}{10} - 1$, каждое повторяется 10 раз. Для $d \neq 0$ имеем:
  $$\text{количество вхождений} = 10 \cdot C_{d,\frac{n}{10}}.$$
  3. Для $d = 0$ есть особый случай: первые 10 чисел (0–9) были однозначными, и после «стирания» не осталось ведущих нулей. Поэтому нужно вычесть эти лишние 10 нулей:
  $$\text{количество вхождений} = 10 \cdot \left(C_{0,\frac{n}{10}} - 1\right).$$

Таким образом, мы можем вычислить все $C_{d,n}$ для $d = 0, 1, \ldots, 9$.

Использование в бинарном поиске

В процессе бинарного поиска для текущего кандидата n вычисляем:

$$C_{0,n+1} - 1, \quad C_{1,n+1}, \quad C_{2,n+1}, \ldots, C_{9,n+1}.$$

Далее сравниваем их с количеством соответствующих цифр, которое дано во входных данных. Если цифр достаточно — продолжаем бинарный поиск в правой половине, иначе — в левой.

Задача F. Выбор команды

Заметим, что если в команде из трёх студентов некто не является уникальным максимумом ни по одной из характеристик, то команда не удовлетворяет условию задачи: у этого студента не найдётся характеристики, в которой он строго превосходит остальных двух.

Следовательно, при поиске оптимальной команды можно отбрасывать таких студентов: они не могут входить ни в одну корректную тройку, ведь их присутствие сразу нарушает условие. Более того, удаление таких студентов может сделать других студентов уникальными максимумами, поэтому процесс можно повторять до тех пор, пока это возможно.

Алгоритм:

1. Считаем входные данные: массивы x, y, z.
2. Построим по три массива индексов p_x, p_y, p_z, отсортированные по x, y, z соответственно.
3. Будем поддерживать три указателя t_x, t_y, t_z, которые указывают на текущие максимальные элементы по каждой характеристике (начиная с конца массивов).
4. Повторяем:
   - Берём тройку кандидатов: $i_x = p_x[t_x], i_y = p_y[t_y], i_z = p_z[t_z]$.
   - Рассматриваем максимальные значения $m_x = x[i_x], m_y = y[i_y], m_z = z[i_z]$.
   - Если хотя бы один из этих студентов совпадает по двум характеристикам с текущими максимумами (например, $x[i_x] = m_x$ и $y[i_x] = m_y$), он не может быть уникальным максимумом — его исключаем (отмечаем как использованного) и сдвигаем соответствующий указатель.
   - Иначе все три студента являются уникальными максимумами (каждый максимален хотя бы по одной характеристике). Сумма $m_x+m_y+m_z$ — это сила команды, и она максимальна из-за того, что мы всегда поддерживаем максимальные значения. Выводим её и завершаем программу.
5. Если в какой-то момент один из указателей стал равен -1, значит подходящей тройки не существует — выводим -1.

Корректность.

Алгоритм всегда поддерживает множество студентов, которые ещё могут образовать корректную команду. На каждом шаге он удаляет студента, который не может быть уникальным максимумом ни по одной характеристике при текущих максимальных значениях (и значит, не может попасть в оптимальную команду). Так как мы всегда выбираем текущие глобальные максимумы по x, y, z, то первая тройка, которая проходит проверку на уникальность, и будет оптимальной по сумме.

Сложность.

- Сортировки трёх массивов занимают $O(n \log n)$.
- Каждый студент удаляется не более одного раза — указатели t_x, t_y, t_z двигаются не более n шагов.
- Следовательно, суммарная асимптотика: $O(n \log n)$ по времени и $O(n)$ по памяти.

Задача G. Схождение уровней

Рассмотрим двух игроков, находящихся на уровнях a и b соответственно.

Случай 1: одинаковая чётность a и b

Если a и b имеют одинаковую чётность, то первый игрок должен проигрывать все шаги, а второй выигрывать. В этом случае они встретятся через

$$\frac{b-a}{2}$$

раундов.

Случай 2: разная чётность a и b

Если a и b имеют разную чётность, то при движении игроков навстречу друг другу они смогут максимально приблизиться (остаться на соседних уровнях), но не встретятся. Единственный способ изменить чётность это:

- выиграть дополнительный матч на уровне 1, или
- проиграть дополнительный матч на уровне n.

Таким образом, нужно рассмотреть два варианта:

1. Первый игрок идёт к уровню 1, выигрывает дополнительный шаг и начинает двигаться к второму игроку, который всё это время движется к уровню 1, пока они не встретятся.
2. Второй игрок идёт к уровню n, проигрывает дополнительный шаг и начинает двигаться к первому игроку, который всё это время движется к уровню n, пока они не встретятся.

Минимальное число шагов, необходимых до их встречи, равно:

$$\min(a-1,\, n-b) + 1 + \frac{b-a-1}{2}.$$

Здесь:

- $\min(a-1, n-b)$ — число шагов, необходимых, чтобы достичь уровня 1 или n;
- 1 — один шаг, необходимый для изменения чётности;
- $\frac{b-a-1}{2}$ — число шагов до встречи после изменения чётности (равно половине расстояния между игроками).

Задача H. Выбор отрезков из множеств

Поскольку n мало, можно перебрать все подмножества наборов: всего существует 2^n подмножеств. Для каждого подмножества $m \subseteq \{0, 1, \ldots, n-1\}$ определим

$$dp[m] = \max\{x \mid \text{отрезки из } m \text{ покрывают } [0, x) \text{ непрерывно}\}.$$

Если $dp[m] < l$, можно попробовать добавить ещё один набор $i \notin m$ и выбрать в нём такой отрезок, чтобы покрытие стало как можно длиннее.

Предподсчет переходов

Чтобы быстро находить подходящий отрезок при добавлении нового набора, предподсчитаем для всех возможных позиций x, наборов i и отрезков j **наиболее поздний отрезок** j, который можно поставить так, чтобы он начинался не позже x. Обозначим:

$$\text{next}[x][i] = \max\{\text{конец отрезка} \mid \text{отрезок из набора } i \text{ начинается} \leqslant x\}.$$

Это позволяет за $O(1)$ узнать, насколько дальше можно продвинуть покрытие, если текущая правая граница равна x и мы хотим добавить один отрезок из набора i.

Предвычисление делается так:

- для каждого набора i отсортируем его отрезки по началу;
- для каждой позиции x (например, для всех концов отрезков) найдём максимальный конец отрезка с началом $\leqslant x$;
- сохраним это значение в $\text{next}[x][i]$.

Сложность предвычисления $O(Cn)$, где C — количество отрезков на набор (в сумме по всем наборам).

Переход динамики

Для каждого состояния m:

- известна текущая покрытая длина $x = dp[m]$;
- для каждого $i \notin m$ находим $y = \text{next}[x][i]$;
- если $y > x$, обновляем:

$$m' = m \cup \{i\}, \qquad dp[m'] = \max(dp[m'], y).$$

Итоговый ответ

Минимальный ответ:

$$\min\{|m| \mid dp[m] \geqslant l\}.$$

Если такого m нет, ответ -1.

Сложность

- Предвычисление: $O(Cn)$.
- Основной цикл: $O(2^n \cdot n)$, так как для каждого подмножества пробуем добавить каждый из оставшихся наборов.

Общая асимптотика:

$$O\big((2^n + C) \cdot n\big),$$

Задача I. Откуда ни стартуй

Задача является классической задачей о **синхронизирующей последовательности** (synchronizing word) для детерминированного конечного автомата (ДКА). Цель — найти такую последовательность команд (индексов рёбер), которая из любого начального состояния приведёт автомат в фиксированное состояние (в нашем случае — вершину 1).

Прямой перебор всех 2^n подмножеств состояний невозможен при $n \leqslant 200$. Однако ключевое наблюдение состоит в следующем. При применении любой команды множество возможных текущих состояний **не расширяется**, а может только сужаться. Если удаётся уменьшить количество возможных состояний хотя бы на 1, то повторяя этот процесс, мы в конечном итоге придём к единственному состоянию — вершине 1.

Таким образом, стратегия решения:

1. Последовательно **сливать пары различных состояний** в одно.
2. Для каждой пары (a, b) заранее найдём кратчайшую последовательность команд, которая приведёт оба состояния в одну и ту же вершину (в идеале в вершину 1).
3. Применяя такие последовательности поочерёдно ко всем парам, мы сведём всё множество состояний к $\{1\}$.

Сведение к задаче на парах

Рассмотрим все неупорядоченные пары вершин (a, b), где $a \leqslant b$. Их количество — $\frac{n(n+1)}{2} = O(n^2)$. Для каждой пары (a, b) определим: какая команда (индекс ребра) позволяет перейти к паре (a', b'), где $a' = s_{a,i},\ b' = s_{b,i}$.

Цель: из любой пары (a, b) достичь пары $(1, 1)$. Это можно сделать с помощью **обратного BFS** из состояния $(1, 1)$:

- Изначально помечаем $(1, 1)$ как достижимое.
- Для каждой команды j и для каждой пары (a', b'), уже помеченной как достижимая, находим все пары (a, b), такие что $s_{a,j} = a'$ и $s_{b,j} = b'$.
- Такие пары (a, b) также помечаем как достижимые и запоминаем, что для их синхронизации нужно сначала применить команду j, а затем последовательность для (a', b').

Это позволяет за $O(n^2m)$ построить таблицу `sign_to_follow`$[a][b]$, хранящую первую команду в кратчайшей синхронизирующей последовательности для пары (a, b).

Моделирование слияния

При слиянии пары (a, b) важно знать, куда перейдут **все остальные** состояния после применения найденной последовательности. Для этого используется динамическое программирование:

`simulate_merge`$[x][a][b]$ = состояние, в которое перейдёт x после полной синхронизации пары (a, b).

Это значение вычисляется рекурсивно: если первая команда — j, то

$$\texttt{simulate\_merge}[x][a][b] = \texttt{simulate\_merge}[s_{x,j}][a'][b'],$$

где (a', b') — следующая пара после применения команды j к (a, b).

Финальный алгоритм

1. Изначально все вершины считаются возможными (`occupied`$[i]$ = `true`).
2. Пока не все вершины сведены к 1:
   - Выбираем две различные занятые вершины a и b (если только одна — сливаем её с 1).
   - Применяем последовательность команд, синхронизирующую (a, b).
   - Обновляем множество возможных состояний с помощью `simulate_merge`.
3. Выводим все команды в порядке применения.

Корректность и сложность

Корректность следует из гарантии существования синхронизирующей последовательности и того, что каждая операция слияния строго уменьшает количество возможных состояний. Обратный BFS работает за $O(n^2m)$, так как для каждой из $O(n^2)$ пар и m команд мы перебираем входящие рёбра. Общая длина последовательности не превосходит $O(n^3)$.

Задача J. Волейбол Филиппа

Существует два случая, когда искомого матча не существует:

1. Если выполняется неравенство $n < 3a$, так как нам нужно иметь как минимум 3 сета с как минимум a очками в каждом.

2. Если $a = 2$ и $2 \nmid n$, так как в этом случае каждый сет заканчивается разницей ровно 2, следовательно, общее число очков должно быть чётным.

Во всех остальных случаях решение можно найти ровно в 3 сетах. Предположим, без ограничения общности, что первая команда победила со счётом 3:0 по сетам. Пусть первые два сета закончились со счётом a:0. Тогда в третьем сете было сыграно

$$n' = n - 2a$$

очков. Возможны два случая:

Случай 1: $n' \leqslant 2a - 2$

То есть

$$n' - a \leqslant a - 2.$$

В этом случае третий сет не играется «на разницу» и заканчивается со счётом

$$a : (n' - a).$$

Случай 2: $n' > 2a - 2$

То есть

$$\frac{n'}{2} + 1 > a,$$

и третий сет играется «на разницу», то есть первая команда набирает более чем a очков и выигрывает с разницей ровно 2.

Если n' чётно, третий сет заканчивается со счётом

$$\frac{n'}{2} + 1 : \frac{n'}{2} - 1.$$

По условию задачи этот результат валиден.

Если n' нечётно, третий сет заканчивается со счётом

$$\frac{n' - 1}{2} + 1 : \frac{n' - 1}{2} - 1.$$

Из условия следует, что

$$\frac{n' - 1}{2} + 1 > a,$$

так как a — натуральное число, следовательно, результат валиден. У нас остаётся 1 лишнее очко, которое можно просто «перенести» во второй сет, получив там счёт $a : 1$. Это невозможно сделать только при $a = 2$, но этот случай для нечётного n' (а значит, и нечётного n) мы уже разобрали.

Задача K. Минимальная Галерея

Нам требуется выбрать четыре точки среди n данных точек на плоскости так, чтобы образованный простой четырёхугольник имел минимальную площадь. Площадь четырёхугольника пропорциональна затратам на обслуживание, поэтому минимизация площади напрямую соответствует задаче.

Основное наблюдение заключается в том, что любой простой четырёхугольник можно разбить одной из диагоналей на два треугольника. Пусть мы выберем диагональ, соединяющую точки p_1 и p_3. Тогда оставшиеся две вершины p_2 и p_4 лежат по разные стороны прямой p_1p_3. Площадь четырёхугольника равна сумме площадей треугольников $p_1p_3p_2$ и $p_1p_3p_4$, а каждая из этих площадей вычисляется как половина произведения длины диагонали p_1p_3 на расстояние соответствующей точки от прямой p_1p_3.

Следовательно, для каждой пары точек p_1 и p_3 можно найти минимальную площадь четырёхугольника, выбирая по одной точке с каждой стороны прямой p_1p_3, которая минимизирует высоту

треугольника. Осталось перебрать все пары точек p_1 и p_3 и для каждой пары выбирать по точке с каждой стороны, давая тем самым минимальную возможную площадь для этой диагонали.

Прямой перебор всех четырёх точек даёт сложность $O(n^4)$, что слишком много. Более эффективный метод состоит в переборе всех пар точек для диагонали ($O(n^2)$) и нахождении ближайших точек с каждой стороны ($O(n)$ на пару), что даёт $O(n^3)$. Для небольших n это допустимо, но для больших n требуется оптимизация.

Основная оптимизация основана на геометрическом приёме, называющийся вращающийся сканлайн (circular sweepline). Идея заключается в следующем. Пусть мы фиксируем пару точек p_1 и p_3, которые рассматриваем как диагональ четырёхугольника. Остальные точки можно классифицировать относительно прямой p_1p_3 по сторонам этой прямой. Для минимизации площади необходимо выбрать с каждой стороны прямой точку, ближайшую к линии, поскольку площадь треугольника пропорциональна высоте, опущенной на диагональ. Таким образом, для фиксированной диагонали достаточно рассмотреть лишь по одной ближайшей точке с каждой стороны.

Если рассматривать все возможные диагонали независимо, это даёт $O(n^3)$ перебор: для каждой из $O(n^2)$ диагоналей проверяем $O(n)$ остальных точек. Чтобы ускорить процесс, используется сортировка точек по расстоянию до прямой и постепенное вращение линии. Все прямые, образованные парами точек, сортируются по углу наклона. Начальная сортировка точек относительно горизонтальной линии позволяет быстро определить ближайших соседей для первой диагонали. Затем, при переходе к следующей прямой в порядке увеличения угла, порядок точек меняется минимально: только точки, образующие новую диагональ, меняют свои позиции в отсортированном списке.

Благодаря этому для каждой новой диагонали кандидаты для p_2 и p_4 находятся за константное время — достаточно проверить соседние элементы в списке относительно линии. Таким образом, поиск минимальных точек с каждой стороны для каждой диагонали выполняется за $O(1)$, а общая сложность алгоритма определяется лишь сортировкой всех $O(n^2)$ прямых по наклону, что даёт $O(n^2 \log n)$ по времени.

Задача L. Робот

Робот перемещается по тороидальной сетке размером $10^9 \times 10^9$. Программа содержит команды `N, S, E, W` и конструкции вида $k(P)$, которые повторяют подпрограмму P k раз. Полное раскрытие программы невозможно из-за потенциальной экспоненциальной длины при вложенных повторениях. Вместо этого будем вычислять суммарное смещение робота $(\Delta x, \Delta y)$ без явного разворачивания всех команд.

Для этого используется словарь, аккумулирующий смещения по направлениям `N, S, E, W`, и стек для хранения множителей повторений. Текущий множитель отражает, сколько раз должны учитываться команды внутри текущей подпрограммы. Алгоритм проходит по символам программы слева направо. Когда встречается цифра k, это означает начало конструкции $k(P)$; текущий множитель сохраняется в стек и умножается на k, чтобы команды внутри скобок учитывались k раз. Открывающая скобка '(' не требует дополнительных действий, так как множитель уже установлен. Закрывающая скобка ')' сигнализирует о завершении повторения, поэтому из стека извлекается предыдущий множитель и восстанавливается текущий. Если встречается команда направления `N, S, E, W`, к соответствующему элементу словаря прибавляется текущее значение множителя, что корректно учитывает повторения на любом уровне вложенности.

После обработки всей программы итоговые координаты вычисляются по формулам $\text{row} = (S - N) \bmod 10^9 + 1$ и $\text{col} = (E - W) \bmod 10^9 + 1$, что учитывает тороидальность сетки. Такой подход гарантирует корректное поведение робота при выходе за границы.

```
for T in range(1, int(input()) + 1):
    P = input()
    C = {"N":0, "E":0, "W":0, "S":0}
    M = []
    mul = 1
    for p in P:
        if p == '(':
            pass
```

```
        elif p == ')':
            mul = M.pop()
        elif p in 'NEWS':
            C[p] += mul
        else:
            M.append(mul)
            mul *= int(p)

SN = (C["S"] - C["N"]) % (10**9) + 1
EW = (C["E"] - C["W"]) % (10**9) + 1

print(EW, SN)
```

Задача M. Причем тут Евровидение

Ключевые наблюдения

1. **Порядок объявления** — это порядок возрастания s_i, а при равенстве — возрастания p_i. Поскольку p_i различны, полный порядок определён однозначно для любого набора s_i.

2. **Лидер меняется на каждом шаге**. Это означает, что если до объявления было k певцов, то после объявления $(k+1)$-го певца его суммарный балл должен стать строго больше, чем у любого из предыдущих k певцов.

3. **Итоговый порядок** — это просто перестановка певцов, соответствующая порядку их объявления. Таким образом, задача сводится к подсчёту количества перестановок $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, для которых существует распределение $s_{\pi_1}, \ldots, s_{\pi_n}$, удовлетворяющее:

   - $s_{\pi_1} \leqslant s_{\pi_2} \leqslant \ldots \leqslant s_{\pi_n}$,
   - если $s_{\pi_i} = s_{\pi_j}$ и $i < j$, то $p_{\pi_i} < p_{\pi_j}$,
   - $\sum s_i = x$,
   - после объявления π_k суммарный балл $p_{\pi_k} + s_{\pi_k}$ строго больше, чем у всех $\pi_1, \ldots, \pi_{k-1}$,
   - итоговый лидер π_n отличается от исходного лидера (того, у кого был максимум p_i).

Минимальные требования к баллам

Пусть фиксирован порядок объявления π. Обозначим $a_k = s_{\pi_k}$. Тогда:

$$a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n,$$

и для каждого $k = 1..n$ должно выполняться:

$$p_{\pi_k} + a_k > \max_{1 \leqslant i < k} (p_{\pi_i} + a_i).$$

Это условие можно переписать рекуррентно. Пусть $M_{k-1} = \max_{i<k}(p_{\pi_i} + a_i)$. Тогда минимально возможное a_k:

$$a_k \geqslant \max\left(a_{k-1},\ M_{k-1} - p_{\pi_k} + 1\right).$$

Таким образом, для фиксированной перестановки π можно вычислить **минимальную сумму** $\sum a_k$, необходимую для выполнения условий. Если эта сумма $\leqslant x$, то перестановка допустима.

Однако перебор всех $n!$ перестановок при $n \leqslant 12$ возможен, но требует оптимизации, так как $12! \approx 479$ млн — слишком много. Поэтому используется динамическое программирование по подмножествам.

Динамика по маскам

Состояние — это:

- маска `mask` — какие певцы уже объявлены,
- последний объявленный певец `last`,
- текущая сумма выданных баллов `sum`.

Переход: добавить нового певца $i \notin$ `mask`, вычислить минимальное количество баллов, которое нужно дать ему, чтобы он стал новым лидером и не нарушал порядок неубывания.

Перед началом массив p сортируется по возрастанию. Это упрощает обработку условия «при равенстве баллов — по p_i»: порядок объявления должен быть совместим с сортировкой. Для маски размера 1: инициализируются только те певцы, которые не являются исходным лидером (последний в отсортированном массиве). При переходе от маски без i к маске с i вычисляется минимальный дополнительный вклад:

$$\texttt{need} = \max(0, p[\texttt{prev}] + 1 - p[i]) \cdot (n - \texttt{have} + 1),$$

где множитель $(n - \texttt{have} + 1)$ учитывает, что баллы должны быть неубывающими, и минимальное значение можно «растянуть» на оставшиеся шаги. В конце суммируются все состояния с полной маской.

Количество масок 2^n, для каждой маски — n вариантов последнего итоговая сложность: $O(2^n \cdot n^2 \cdot sum)$.