Оглавление

Задача «Уравнение Теневого Древа»

Постановка задачи

Есть $N = 2^n - 1$ вершин, соединенных дорогами в виде полного бинарного дерева. Дороги задаются правилом: i-я дорога соединяет вершины $(i+1)$ и $\lceil i/2 \rceil$ для $i = 1, 2, \ldots, N-1$. Вершины с номерами от $\lceil N/2 \rceil$ до N имеют фиксированные параметры a_i, которые могут изменяться при запросах, а параметры вершин с номерами от 1 до $\lfloor N/2 \rfloor$ имеют можно выбрать произвольно.

Требуется минимизировать сумму квадратов разностей параметров вдоль всех дорог:

$$\text{ans} = \min_{a_1,\ldots,a_{\lfloor N/2 \rfloor} \in \mathbb{R}} \sum_{i=1}^{N-1} \left(a_{i+1} - a_{\lceil i/2 \rceil}\right)^2$$

После каждого изменения параметра листа нужно выводить новое оптимальное значение ans по модулю 998244353.

Структура дерева и идея

Перенумеруем вершины так, чтобы дерево представляло собой полное бинарное дерево с корнем в вершине 1; листья - вершины с номерами от $L = 2^{n-1}$ до $N = 2^n - 1$. Используем ДП по поддеревьям: для каждой вершины u определим функцию $f_u(x)$, которая возвращает оптимальную стоимость поддерева в зависимости от параметра x вершины u ($a_u = x$).

Для листа u с известным параметром a_u стоимость определяется просто: $f_u(x) = (x - a_u)^2$, ведь у него нет детей, а дорога к родителю будет учтена на уровне родителя. Раскрывая квадрат получим: $f_u(x) = x^2 - 2a_u x + a_u^2$. Определим для листа коэффициенты квадратичной функции: $p_u = 1$, $q_u = -2a_u$, $r_u = a_u^2$.

Для вершины u с детьми l и r: $f_u(x) = \min_y \left[(x-y)^2 + f_l(y)\right] + \min_z \left[(x-z)^2 + f_r(z)\right]$. x - параметр текущей вершины u. Мы рассматриваем, как изменится оптимальная стоимость поддерева, если параметр вершины u равен x. y - параметр левого ребенка вершины u (вершины l). Мы перебираем все возможные параметры левого ребенка, чтобы найти оптимальный. z - параметр правого ребенка вершины u. Мы перебираем все возможные параметры правого ребенка, чтобы найти оптимальный. $(x-y)^2$ - стоимость дороги между вершиной u (параметр x) и её левым ребенком (параметр y). $(x-z)^2$ - стоимость дороги между вершиной u (параметр x) и её правым ребенком (параметр z). $f_l(y)$ - минимальная стоимость левого поддерева при условии, что параметр левого ребенка равен y. Это уже вычисленная рекурсивно функция для вершины l. $f_r(z)$ - минимальная стоимость правого поддерева при условии, что параметр правого ребенка равен z. Это уже вычисленная рекурсивно функция для вершины r.

Чтобы упростить рекуррентное соотношение, введём вспомогательную функцию.

$$g(x) = \min_y \left[(x-y)^2 + f(y)\right]$$

x - параметр родителя, y - параметр ребенка, $f(y)$ - стоимость поддерева ребенка при параметре y. Таким образом функция $g(x)$ представляет минимальную стоимость поддерева ребенка плюс стоимость дороги от родителя к ребенку.

Вычисление $g(x)$

Пусть $f(y) = py^2 + qy + r$. Вычислим $g(x)$.

$$g(x) = \min_y \left[(x-y)^2 + py^2 + qy + r\right]$$

Раскрываем квадрат:

$$\begin{aligned}(x-y)^2 + py^2 + qy + r &= x^2 - 2xy + y^2 + py^2 + qy + r \\ &= x^2 - 2xy + (1+p)y^2 + qy + r\end{aligned}$$

Обозначим: $A = 1+p$, $B = -2x+q$, $C = x^2 + r$. Минимум квадратичной функции $Ay^2 + By + C$ при $A > 0$ достигается в точке: $y^* = -\frac{B}{2A} = \frac{2x-q}{2(1+p)}$. Значение минимума: $C - \frac{B^2}{4A} = x^2 + r - \frac{(-2x+q)^2}{4(1+p)}$. Тогда:

$$\begin{aligned}g(x) &= x^2 + r - \frac{4x^2 - 4qx + q^2}{4(1+p)} \\ &= x^2 + r - \frac{4x^2}{4(1+p)} + \frac{4qx}{4(1+p)} - \frac{q^2}{4(1+p)} \\ &= x^2 - \frac{x^2}{1+p} + \frac{qx}{1+p} - \frac{q^2}{4(1+p)} + r \\ &= \frac{p}{1+p}x^2 + \frac{q}{1+p}x + \left(r - \frac{q^2}{4(1+p)}\right)\end{aligned}$$

Таким образом, новые коэффициенты: $p' = \frac{p}{1+p}$, $q' = \frac{q}{1+p}$, $r' = r - \frac{q^2}{4(1+p)}$. И $g(x) = p'x^2 + q'x + r'$. Для вершины u с детьми l и r: $f_u(x) = g_l(x) + g_r(x)$, где $g_l(x)$ и $g_r(x)$ вычисляются по формулам выше.

Таким образом, коэффициенты для $f_u(x)$: $p_u = p'_l + p'_r$, $q_u = q'_l + q'_r$, $r_u = r'_l + r'_r$. Функция $g(x)$ предоставляет родителю информацию о том, сколько будет стоить выбрать параметр x с учетом оптимального выбора параметров в поддереве ребенка.

Вычисление ответа и обработка обновлений

Для корня дерева (вершина 1) нам нужно найти:

$$\text{ans} = \min_x f_1(x)$$

Поскольку $f_1(x) = p_1x^2 + q_1x + r_1$ - квадратичная функция, её минимум достигается в точке: $x^* = -\frac{q_1}{2p_1}$. Значение минимума: $ans = f_1(x^*) = r_1 - \frac{q_1^2}{4p_1}$.

При изменении параметра листа x на значение w: обновляем коэффициенты листа. $p[x] = 1$, $q[x] = -2w$, $r[x] = w^2$. Пересчитываем всех предков: начиная с родителя $\lfloor x/2 \rfloor$ и двигаясь вверх до корня, для каждой вершины u на пути пересчитываем коэффициенты по формулам:

Пусть $l = 2u$, $r = 2u+1$ - дети вершины u. Тогда: $p_u = \frac{p_l}{1+p_l} + \frac{p_r}{1+p_r}$, $q_u = \frac{q_l}{1+p_l} + \frac{q_r}{1+p_r}$, $r_u = \left(r_l - \frac{q_l^2}{4(1+p_l)}\right) + \left(r_r - \frac{q_r^2}{4(1+p_r)}\right)$

Вычисляем новый ответ для корня по формуле: $\text{ans} = r_1 - \frac{q_1^2}{4p_1}$

Так как глубина дерева $O(n)$, то и время на одно обновление: $O(n)$. А значит общее время для m обновлений: $O(m \cdot n)$. Для $n \le 18$ и $m \le 2 \times 10^5$ это укладывается в TL 2sec.

Примечания

Поскольку параметры могут быть до 10^8, а ответы могут быть дробными, все значения выводятся по модулю 998244353. Обратный элемент вычисляется по малой теореме Ферма: $a^{-1} \equiv a^{998244351} \pmod{998244353}$, а деление заменяется умножением на обратный элемент.

Реализация

```
void Solve() {
    int MOD = 998244353;
    auto POW = [&](int a, int b) {
        int ans = 1; a %= MOD; b %= MOD;
        while (b) {
            if (b & 1) {
                ans = (ans * a) % MOD;
            } a = (a * a) % MOD; b >>= 1;
        } return ans;
    };
    int n, m; cin >> n >> m;
    int N = (1LL << n) - 1;
    int L = 1LL << (n - 1);
    vector<int> a(L);
    for (int i = 0; i < L; i++) {
        cin >> a[i];
    }
    vector<int> p(N + 1), q(N + 1), r(N + 1);
    for (int i = 0; i < L; i++) {
        p[L + i] = 1;
        q[L + i] = (-2 * a[i] + 2 * MOD) % MOD;
        r[L + i] = (a[i] * a[i]) % MOD;
    }
    for (int i = L - 1; i >= 2; i--) {
        int A = (p[2 * i] + p[2 * i + 1]) % MOD;
        int B = (q[2 * i] + q[2 * i + 1]) % MOD;
        int C = (r[2 * i] + r[2 * i + 1]) % MOD;
        p[i] = A * POW(1 + A, MOD - 2) % MOD;
        q[i] = A * POW(1 + A, MOD - 2) % MOD;
        r[i] = (C - (B * B) % MOD * (POW(4, MOD - 2)) % MOD * POW(1 + A, MOD - 2) %
            ↪ MOD + MOD) % MOD;
    }
    auto F = [&]() {
        int A = (p[2] + p[3]) % MOD;
        int B = (q[2] + q[3]) % MOD;
        int C = (r[2] + r[3]) % MOD;
        return (C - (B * B % MOD) * POW(4 * A % MOD, MOD - 2) % MOD + MOD) % MOD;
    };
    cout << F() << '\n';
    for (int z = 0; z < m; z++) {
        int x, w; cin >> x >> w;
        p[x] = 1;
        q[x] = (-2 * w + 2 * MOD) % MOD;
        r[x] = (w * w) % MOD;
        int i = x / 2;
        while (i > 1) {
            int A = (p[2 * i] + p[2 * i + 1]) % MOD;
            int B = (q[2 * i] + q[2 * i + 1]) % MOD;
            int C = (r[2 * i] + r[2 * i + 1]) % MOD;
            p[i] = A * POW(1 + A, MOD - 2) % MOD;
            q[i] = B * POW(1 + A, MOD - 2) % MOD;
            r[i] = (C - (B * B) % MOD * (POW(4, MOD - 2)) % MOD * POW(1 + A, MOD - 2)
                ↪  % MOD + MOD) % MOD;
            i = i / 2;
        }
        cout << F() << '\n';
    }
}
```

Задача «Расстояние Левенштейна»

Постановка задачи

Расстояние Левенштейна между двумя строками — это минимальное количество односимвольных операций (вставка, удаление, замена), необходимых для превращения одной строки в другую.

Дана исходная строка s (длина $1 \le |s| \le 10$, только заглавные латинские буквы) и целое число d ($0 \le d \le 10$). Требуется найти количество различных строк в алфавите ‘A‘–‘Z‘, расстояние Левенштейна от которых до s равно в точности d. Пустая строка также считается допустимой. Ответ необходимо вывести по модулю 998 244 353.

Основная идея

Наивный перебор всех строк невозможен, так как уже при длине 10 существует 26^{10} вариантов. Однако длина s и d не превосходят 10. Это позволяет применить динамическое программирование, которое будет строить все возможные строки посимвольно, одновременно поддерживая текущее расстояние Левенштейна до префиксов s.

Рассмотрим две строки: исходную s (длины n) и строящуюся t (переменной длины m). Редакционное расстояние $D(i, j)$ между префиксом $s[1..i]$ и префиксом $t[1..j]$ вычисляется по известному динамическому программированию Вагнера–Фишера:

$$D(i, 0) = i, \qquad D(0, j) = j,$$

$$D(i, j) = \min\big(D(i-1, j) + 1,\ D(i, j-1) + 1,\ D(i-1, j-1) + [s_i \neq t_j]\big).$$

Значения $D(i, j)$ образуют таблицу размера $(n + 1) \times (m + 1)$.

Представьте, что мы строим строку t слева направо, символ за символом. В каждый момент, когда уже построен префикс t длины j, нам не нужна вся таблица — достаточно знать последний столбец:

$$\text{profile} = \big(D(0, j),\ D(1, j),\ D(2, j),\ \ldots,\ D(n, j)\big).$$

Почему? Потому что при дописывании одного символа c к t новый столбец для длины $j + 1$ вычисляется исключительно по предыдущему столбцу и символу c. Формула перехода:

$$D(0, j + 1) = j + 1,$$

$$D(i, j + 1) = \min\big(D(i-1, j+1) + 1,\ \ D(i, j) + 1,\ \ D(i-1, j) + [s_i \neq c]\big), \quad i = 1 \ldots n.$$

Таким образом, профиль — это вектор $p_0, p_1, \ldots, p_n$, где $p_i = D(i, j)$. Он содержит всю информацию, необходимую для продолжения построения строки t без знания того, какой именно префикс t был построен раньше. Разные строки одной длины могут приводить к одному и тому же профилю, и с точки зрения дальнейших продолжений они неразличимы.

Следовательно, задачу можно решать, храня для каждой длины j количество строк, которые дают тот или иной профиль. Это классический приём «ДП по профилю».

Ограничение на длину t

Редакционное расстояние удовлетворяет неравенству треугольника. Для любых двух строк расстояние d не может быть меньше разности их длин: $d \geq ||s| - |t||$. Кроме того, оно не может превышать сумму длин (если все символы разные, расстояние равно максимуму из длин, но в общем случае можно оценить $d \leq n + m$). Значит, если мы ищем строки t, расстояние от которых до s ровно d, длина m обязана лежать в пределах

$$\max(0, n - d) \leq m \leq n + d.$$

Выходить за эти пределы — значит гарантированно получить расстояние, большее или меньшее d, либо невозможность получить в точности d. Поэтому будем перебирать длины m только в этом диапазоне.

ДП: база, переход, ответ

Начальный профиль (пустая t)

Когда t пуста ($j = 0$), расстояние до префикса $s[1..i]$ равно i (нужно удалить все i символов). Профиль:

$$(0, 1, 2, \ldots, n).$$

Единственная строка длины 0 — пустая, и ей соответствует этот профиль с количеством 1.

Переход к следующей длине

Пусть для некоторой длины j у нас есть множество профилей и для каждого известно, сколько строк дают этот профиль (по модулю). Чтобы получить все возможные профили для длины $j + 1$, мы для каждого профиля перебираем все 26 букв латинского алфавита, применяем формулу перехода к новому столбцу, и прибавляем количество строк-исходников к соответствующему новому профилю.

После обработки всех профилей и всех букв получаем новую «картину» для длины $j + 1$. Это повторяется, пока длина не достигнет максимума $n + d$.

Когда фиксировать ответ

Ответ — это сумма количеств тех профилей, у которых последний элемент p_n равен d (т.е. расстояние между всей s и всей t равно d), и при этом длина j принадлежит разрешённому диапазону $[\mathrm{MIN}, \mathrm{MAX}]$. Мы можем проверять это после каждого увеличения длины (включая длину 0).

Таким образом, алгоритм постепенно наращивает длину t, и по пути собирает строки с нужным финальным расстоянием.

Почему количество профилей невелико

Профиль состоит из $n + 1$ числа, каждое в диапазоне от 0 до $n + d \leq 20$. Кроме того, эти числа не произвольны: всегда $p_0 = j$, и для $i \geq 1$ выполняется $|p_i - p_{i-1}| \leq 1$. Это следует из того, что расстояние между соседними префиксами s отличается не более чем на 1 при добавлении одного символа к t. Действительно, $D(i, j) \leq D(i - 1, j) + 1$ и $D(i - 1, j) \leq D(i, j) + 1$. Значит, профиль — это почти «плавная» последовательность, которая может расти, убывать или оставаться на месте, но не скачет больше чем на 1 за шаг.

Перебор всех таких последовательностей показывает, что их количество при $n \leq 10$ не превышает нескольких тысяч. Это позволяет хранить их в словаре (map) и обрабатывать за разумное время.

Реализация

```cpp
void Solve() {
    int MOD = 998244353;

    string s; int d;
    cin >> s >> d;

    int n = s.size();
    int MIN = max(0, n - d);
    int MAX = n + d;

    map<vector<int>, int> M;
    vector<int> a(n + 1);
    for (int i = 0; i <= n; i++) {
        a[i] = i;
    }
    M[a] = 1;

    int ans = 0;
    if (0 >= MIN && 0 <= MAX) {
        for (auto& [st, cnt] : M) {
            if (st[n] == d) {
                ans = (ans + cnt) % MOD;
            }
        }
    }

    for (int j = 0; j < MAX; j++) {
        map<vector<int>, int> M_;
        for (auto& [st, cnt] : M) {
            for (char c = 'A'; c <= 'Z'; c++) {
                vector<int> nst(n + 1);
                nst[0] = j + 1;
                for (int i = 1; i <= n; i++) {
                    nst[i] = min({ nst[i - 1] + 1, st[i] + 1, st[i - 1] +
                        (s[i - 1] != c) });
                }
                M_[nst] = (M_[nst] + cnt) % MOD;
            }
        }
        M = M_;
        if (j + 1 >= MIN && j + 1 <= MAX) {
            for (auto& [st, cnt] : M) {
                if (st[n] == d) {
                    ans = (ans + cnt) % MOD;
                }
            }
        }
    }

    cout << ans;
}
```

Задача «Хайтей Раоюэ»

Постановка задачи

Учитель может применять к начальному баллу x ($1 \le x \le 100$) четыре типа операций:

- A (не более a_1 раз): $x \leftarrow \lfloor 10\sqrt{x} \rfloor$;
- B (не более a_2 раз): $x \leftarrow \lfloor 0.7x + 30 \rfloor$;
- C (не более a_3 раз): $x \leftarrow \lfloor 1.2x \rfloor$;
- D (не более a_4 раз): $x \leftarrow x + 5$.

Требуется для каждого из T тестов ($T \le 3 \cdot 10^5$, $0 \le a_i \le 400$) найти максимально возможный итоговый балл.

Анализ операций

- **Операции A и B** не являются монотонно возрастающими, но для малых x (до ≈ 100) они могут увеличивать значение, а при больших — уменьшают. Использовать более 100 операций A и B не имеет смысла — дальнейшие применения не улучшат результат.
- **Операция C** (умножение на 1.2) выгодна в конце: чем больше текущее число, тем больший прирост она даёт. Поскольку она не имеет отрицательных эффектов, все a_3 операций C следует применять после всех остальных.
- **Операция D** (прибавление 5) также только увеличивает балл. Её можно частично применить до A и B, чтобы поднять x перед обработкой, но тогда нужно следить, чтобы x не превысил 100 (иначе A и B могут начать уменьшать значение). Оставшиеся операции D применяются после A и B, но до C.

Таким образом, нужно выбрать число i ($0 \le i \le \min(a_4, \lfloor (100 - x)/5 \rfloor)$) операций D, которые будут применены до A и B. При этом начальное значение станет $x_0 = x + 5i \le 100$. Далее применить к x_0 не более $c_1 = \min(a_1, 100)$ операций A и $c_2 = \min(a_2, 100)$ операций B в некотором оптимальном порядке. Результат обозначим $y = f(x_0, c_1, c_2)$. Ко всем оставшимся операциям D (их $a_4 - i$) добавить их к y: $y' = y + 5(a_4 - i)$. Далее применить все a_3 операций C: итог $= \lfloor 1.2^{a_3} \cdot y' \rfloor$ (с округлением вниз на каждом шаге). Максимум по всем допустимым i даст ответ.

Предподсчёт функции f (операции A и B)

Для каждого начального $x \in [1, 100]$ и количеств $c_1, c_2 \in [0, 100]$ определим

$$dp_1[x][c_1][c_2] = \text{максимальный балл после } c_1 \text{ операций A и } c_2 \text{ операций B.}$$

База: $dp_1[x][0][0] = x$. Переходы (при $c_1 > 0$ или $c_2 > 0$):

$$dp_1[x][c_1][c_2] = \max\Big(dp_1\big[\,\lfloor 10\sqrt{x} \rfloor\,\big][c_1 - 1][c_2], dp_1\big[\,\lfloor 0.7x + 30 \rfloor\,\big][c_1][c_2 - 1]\Big).$$

Предподсчёт операции C

Пусть $dp_2[s][k]$ — результат применения k операций C к начальному числу s. Рекуррентно:

$$dp_2[s][0] = s \qquad dp_2[s][k] = \lfloor 1.2 \cdot dp_2[s][k-1] \rfloor$$

Поскольку $1.2 = \frac{6}{5}$, целочисленно: $dp_2[s][k] = (dp_2[s][k-1] \cdot 6) \,/\, 5$. Максимальное возможное s (после всех D и A/B) не превосходит $100 + 5 \cdot 400 = 2100$. Количество операций C — до 400. Размер таблицы 2101×401. Все значения могут быть огромны (порядка $x \cdot 1.2^{400} \approx 10^{30}$), поэтому используем 128-битный целый тип (`__int128` в C++).

Итоговый перебор

Для одного теста (x, a_1, a_2, a_3, a_4): устанавливаем $c_1 = \min(a_1, 100)$, $c_2 = \min(a_2, 100)$ и перебираем i от 0 до $mx = \min(a_4, \lfloor (100 - x)/5 \rfloor)$. Для каждого i:

$$\text{cur} = dp_2\big[\, dp_1[x + 5i][c_1][c_2] + 5(a_4 - i) \,\big][a_3].$$

Если $a_4 > mx$ (т.е. нельзя применить все D до достижения границы 100), вычисляем вариант без A и B:

$$\text{cur} = dp_2[\, x + 5a_4 \,][a_3].$$

Ответ — максимум среди всех cur.

Реализация

```
using i128 = __int128_t;

int mas[10001];
int dp1[101][101][101];
i128 dp2[2101][401];

void print_i128(i128 x) {
    if (x == 0) {
        cout << '0' << '\n';
        return;
    }
    string s;
    while (x > 0) {
        s += char('0' + (x % 10));
        x /= 10;
    }
    reverse(s.begin(), s.end());
    cout << s << '\n';
}

void Solve() {
    int z = 0;
    for (int i = 0; i <= 10000; i++) {
        while (z * z <= i) {
            z++;
        }
        mas[i] = z - 1;
    }

    for (int x = 1; x <= 100; x++) {
        dp1[x][0][0] = x;
    }

```

```
    for (int c1 = 0; c1 <= 100; c1++) {
        for (int c2 = 0; c2 <= 100; c2++) {
            if (c1 == 0 && c2 == 0) {
                continue;
            }
            for (int x = 1; x <= 100; x++) {
                int y = x;
                if (c1 > 0) {
                    y = max(y, dp1[mas[100 * x]][c1 - 1][c2]);
                }
                if (c2 > 0) {
                    y = max(y, dp1[(7 * x + 300) / 10][c1][c2 - 1]);
                }
                dp1[x][c1][c2] = y;
            }
        }
    }

    for (int s = 0; s <= 2100; s++) {
        dp2[s][0] = s;
    }
    for (int k = 1; k <= 400; k++) {
        for (int s = 0; s <= 2100; s++) {
            dp2[s][k] = (dp2[s][k - 1] * 6) / 5;
        }
    }

    int t; cin >> t;
    while (t--) {
        int x, a1, a2, a3, a4;
        cin >> x >> a1 >> a2 >> a3 >> a4;

        i128 best = 0;
        int mx = min(a4, (100 - x) / 5);

        int c1 = min(a1, 100);
        int c2 = min(a2, 100);

        for (int i = 0; i <= mx; i++) {
            best = max(best, dp2[dp1[x + 5 * i][c1][c2] + 5 * (a4 - i)][a3]);
        }

        if (a4 > mx) {
            best = max(best, dp2[x + 5 * a4][a3]);
        }

        print_i128(best);
    }
}
```

Задача «Строгое возрастание»

Постановка задачи

Даны n хомяков, упорядоченных по росту (от 1 до n). Для каждого известен вес a_i ($1 \le a_i \le 10^9$). Отряд называется гармоничным, если для любой пары хомяков в нём более высокий имеет строго больший вес. Иными словами, отряд — это возрастающая подпоследовательность массива a. Пусть m — длина наибольшей возрастающей подпоследовательности (НВП) исходного массива. Требуется найти количество способов выбрать непустой отрезок подряд идущих хомяков, чтобы после его удаления длина НВП оставшегося массива осталась равной m.

Предварительные вычисления

Сначала стандартным образом за $O(n \log n)$ находим длину НВП m и для каждого индекса i:

- $END[i]$ — макс. длина возрастающей подпоследовательности, заканчивающейся в позиции i.
- $BEGIN[i]$ — макс. длина возрастающей подпоследовательности, начинающейся в позиции i.

При этом $m = \max_i END[i]$. Введём два важных индекса:

- fme — первый (самый левый) индекс i, для которого $END[i] = m$.
- lmb — последний (самый правый) индекс j, для которого $BEGIN[j] = m$.

Интуиция

Ключевая идея состоит в том, что удаление отрезка разрывает массив на левую и правую части. Чтобы НВП длины m уцелела, должна существовать возрастающая последовательность, которая либо целиком лежит в левой части, либо целиком в правой, либо «перепрыгивает» через удалённый отрезок. Именно этот прыжок и обеспечивается парой (i, j), где i находится левее удаляемого отрезка, j — правее, и они стыкуются по значениям ($a_i < a_j$) и по длинам ($END[i] + BEGIN[j] = m$).

Геометрически это означает, что мы можем взять лучшую левую половинку НВП, заканчивающуюся в i, и немедленно продолжить её лучшей правой половинкой, начинающейся в j. Если самый правый такой j (максимальный по индексу) всё ещё лежит внутри удаляемого отрезка (или левее его правой границы), то никакой мостик не перекрывает разрыв, и НВП длины m теряется.

Для фиксированного i существует, вообще говоря, много позиций $j > i$ с подходящими свойствами. Если удаляемый отрезок покрывает какой-то из них, но другой j' (правее) остаётся, то мостик через i и j' всё равно сработает. Поэтому для проверки достаточно знать самого правого кандидата $mx[i]$; если удаляемый отрезок не доходит до него, он не опасен.

Когда удаление отрезка не вредит НВП

Рассмотрим удаление отрезка $[l, r]$ ($0 \le l \le r < n$, индексация с нуля). Длина НВП не уменьшится, если выполняется хотя бы одно из условий:

1. Слева от отрезка существует полная НВП длины m. Это возможно, если $l > fme$, потому что тогда хомяк fme (и вся НВП, заканчивающаяся в нём) остаётся.
2. Справа от отрезка существует полная НВП длины m. Это возможно, если $r < lmb$, потому что тогда хомяк lmb (и вся НВП, начинающаяся в нём) остаётся.
3. Существует пара элементов $i < l \le r < j$, такая что $a_i < a_j$ и $END[i] + BEGIN[j] = m$. Тогда можно взять НВП, которая проходит «над» удалённым отрезком.

Заметим, что первые два случая охватывают все отрезки, у которых $l > fme$ или $r < lmb$. Потенциально опасными остаются лишь отрезки с $l \le fme$ и $r \ge lmb$.

Построение мостиков

Для каждой позиции i с $END[i] < m$ мы хотим знать самый правый индекс $j > i$, такой что $a_j > a_i$ и $BEGIN[j] = m - END[i]$. Именно такой j позволяет «перепрыгнуть» через удалённый отрезок, пристыковав НВП, заканчивающуюся в i, к НВП, начинающейся в j. Обозначим $mx[i]$ — максимальный такой индекс j. Если подходящих j нет, положим $mx[i] = -1$.

Для эффективного вычисления $mx[i]$ предварительно сгруппируем все индексы j по значению $BEGIN[j]$ (длине начинающейся НВП). Для каждой длины $L = 1 \dots m$ создадим вектор пар (a_j, j), отсортированный по значению a_j. Далее для каждого значения a_j оставляем только максимальный индекс j, а затем строим суффиксный максимум по индексам. Теперь для запроса «найти максимальный j с $a_j > X$ и $BEGIN[j] = L$» достаточно выполнить бинарный поиск по первому элементу пары, большему X, и взять предпосчитанный максимум индекса на суффиксе.

Таким образом, для каждой позиции i, вычислив $L = m - END[i]$ (при $L > 0$), находим $mx[i]$ бинарным поиском в подготовленном массиве для длины L.

Подсчёт плохих отрезков

Назовём отрезок плохим, если его удаление уменьшает максимальную длину НВП. С учётом предыдущего анализа, отрезок $[l, r]$ плох тогда и только тогда, когда одновременно:

- $l \le fme$ (слева нет целой НВП),
- $r \ge lmb$ (справа нет целой НВП),
- $\max\limits_{i<l} mx[i] \le r$ (не существует мостика через отрезок; все потенциальные j лежат не правее r).

Третье условие можно переписать как $r \ge \max(l, lmb, pref\_mx[l])$, где $pref\_mx[l] = \max_{i<l} mx[i]$.

Следовательно, для фиксированного $l \in [0, fme]$ все r, удовлетворяющие $r \ge M = \max(l, lmb, pref\_mx[l])$, делают отрезок плохим. Количество таких r равно $\max(0,\ n - M)$.

Просуммировав по $l \le fme$, получим число плохих отрезков bad. Тогда ответ на задачу есть общее число непустых непрерывных подотрезков $\frac{n(n+1)}{2}$ минус bad.

Реализация

```
void Solve() {
    int n; cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    vector<int> sa = a;
    sort(sa.begin(), sa.end());
    sa.erase(unique(sa.begin(), sa.end()), sa.end());

    vector<int> comp(n);
    for (int i = 0; i < n; i++) {
        comp[i] = lower_bound(sa.begin(), sa.end(), a[i]) - sa.begin();
    }

    vector<int> END(n);
    vector<int> dp;
    for (int i = 0; i < n; i++) {
        auto it = lower_bound(dp.begin(), dp.end(), a[i]);
        if (it == dp.end()) {
            dp.push_back(a[i]);
            END[i] = (int)dp.size();
        }
        else {
            *it = a[i];
            END[i] = (int)(it - dp.begin() + 1);
        }
    }

    int m = (int)dp.size();

    vector<int> BEGIN(n);
    vector<int> dpr;
    for (int i = n - 1; i >= 0; i--) {
        auto it = lower_bound(dpr.begin(), dpr.end(), -a[i]);
        if (it == dpr.end()) {
            dpr.push_back(-a[i]);
            BEGIN[i] = (int)dpr.size();
        }
        else {
            *it = -a[i];
            BEGIN[i] = (int)(it - dpr.begin() + 1);
        }
    }

    int fme = -1;
    for (int i = 0; i < n; i++) {
        if (END[i] == m && fme == -1) {
            fme = i;
        }
    }
    int lmb = -1;
    for (int i = n - 1; i >= 0; i--) {
        if (BEGIN[i] == m && lmb == -1) {
            lmb = i;
        }
    }

```

```
    vector<vector<int>> G(m + 1);
    for (int i = 0; i < n; i++) {
        G[BEGIN[i]].push_back(i);
    }

    vector<vector<pair<int, int>>> lend(m + 1);
    for (int i = 0; i < n; i++) {
        lend[BEGIN[i]].push_back({ comp[i], i });
    }

    for (int i = 1; i <= m; i++) {
        if (lend[i].empty()) {
            continue;
        }
        sort(lend[i].begin(), lend[i].end());
        vector<pair<int, int>> C;
        for (auto& p : lend[i]) {
            int c = p.first;
            int idx = p.second;
            if (!C.empty() && C.back().first == c) {
                C.back().second = max(C.back().second, idx);
            }
            else {
                C.push_back({ c, idx });
            }
        }
        for (int j = (int)C.size() - 2; j >= 0; j--) {
            C[j].second = max(C[j].second, C[j + 1].second);
        }
        lend[i] = move(C);
    }

    vector<int> mx(n, -1);
    for (int i = 0; i < n; i++) {
        int need = m - END[i];
        if (need > 0 && need <= m) {
            auto it = upper_bound(lend[need].begin(), lend[need].end(), make_pair(
                ↪ comp[i], 1e9));
            if (it != lend[need].end()) {
                mx[i] = it->second;
            }
        }
    }

    vector<int> prefmx(n + 1, -1);
    for (int i = 0; i < n; i++) {
        prefmx[i + 1] = max(prefmx[i], mx[i]);
    }

    long long bad = 0;
    for (int l = 0; l <= fme; l++) {
        int M = max({ l, lmb, prefmx[l] });
        if (M < n) {
            bad += n - M;
        }
    }

    cout << (long long)n * (n + 1) / 2 - bad;
}
```

Задача «XOR порталы»

Постановка задачи

Есть дерево из n вершин, рёбра имеют вес w ($0 \le w < 2^{30}$). При прохождении пути состояние путешественника изменяется по правилу XOR: начальное состояние 0, после прохождения ребра с весом w состояние s становится $s \oplus w$. Стоимость пути между i и j — итоговое состояние после прохождения кратчайшего пути (XOR всех весов на нём). Если $i = j$, стоимость 0.

Требуется обрабатывать q запросов двух типов:

1. `x` — магическая буря: веса всех рёбер изменяются $w := w \oplus x$ ($0 \le x < 2^{30}$).
2. `s` — вычислить $\sum_{i=1}^{n} d(s, i)$, где $d(s, i)$ — стоимость пути между s и i ($1 \le s \le n$).

Потенциалы XOR и влияние бури

Выберем произвольную вершину (например, 1) в качестве корня и вычислим для каждой вершины u потенциал V_u — XOR весов на пути от корня до u. Тогда для любых двух вершин s и i имеем:

$$d(s, i) = V_s \oplus V_i$$

Действительно, путь $s \leftrightarrow i$ через LCA даёт $V_s \oplus V_i$ (веса выше LCA сокращаются). Задача свелась к: по запросу `2 s` выдать $\sum_{i=1}^{n}(V_s \oplus V_i)$, где V_u могут изменяться при буре. Пусть начальные потенциалы V_u. При применении бури `x` ко всем рёбрам каждый вес w заменяется на $w \oplus x$. Новый потенциал вершины u с глубиной d_u (количество рёбер на пути от корня):

$$V'_u = V_u \oplus \underbrace{(x \oplus x \oplus \cdots \oplus x)}_{d_u \text{ раз}} = V_u \oplus \big(x \cdot (d_u \bmod 2)\big)$$

XOR накапливается: x участвует d_u раз, итог равен 0 при чётном d_u и x при нечётном. Таким образом, все накопленные бури можно заменить одним параметром X — XOR всех x из запросов типа 1. Тогда текущий потенциал:

$$V_u^{\text{cur}} = V_u \oplus \big(X \cdot (d_u \bmod 2)\big)$$

Побитовый подсчёт ответа

Стоимость $V_s \oplus V_i$ можно разложить по битам ($0 \le k < 30$):

$$\sum_{i=1}^{n}(V_s \oplus V_i) = \sum_{k=0}^{29} 2^k \cdot \big(\text{число } i : \ k\text{-й бит } V_s \ne k\text{-й бит } V_i\big)$$

Обозначим $cnt_1[k]$ — количество вершин i, у которых k-й бит текущего потенциала V_i^{cur} равен 1. Тогда

$$ans(s) = \sum_{k=0}^{29} 2^k \cdot \begin{cases} cnt_1[k], & \text{если } k\text{-й бит } V_s^{\text{cur}} = 0 \\ n - cnt_1[k], & \text{если } k\text{-й бит } V_s^{\text{cur}} = 1 \end{cases}$$

При изменении X необходимо быстро пересчитывать $cnt_1[k]$ и узнавать бит V_s^{cur}. Это можно делать без тяжёлых перестроек, если заметить, что для каждого бита k буря либо действует ($X_k = 1$), либо нет ($X_k = 0$). При $X_k = 0$ потенциалы не меняются, при $X_k = 1$ инвертируется бит потенциала у всех вершин нечётной глубины.

Предподсчёт для двух конфигураций бита

Для каждого бита k построим два массива ответов $ans_0[s]$ и $ans_1[s]$:

- $ans_0[s]$ — ответ $\sum(V_s \oplus V_i)$ при условии, что $X_k = 0$ (исходные веса).
- $ans_1[s]$ — ответ при условии, что $X_k = 1$ (все биты k на рёбрах инвертированы).

Тогда при накопленном X ответ для вершины s равен

$$\sum_{k=0}^{29} 2^k \cdot \big(X_k = 1 \ ? \ ans_1[s]_k : ans_0[s]_k\big)$$

Здесь $ans_0[s]_k$ — количество i с отличающимся k-м битом в исходном дереве, $ans_1[s]_k$ — в дереве с инвертированными рёбрами. Остаётся эффективно вычислить ans_0 и ans_1 для всех s. Для одного бита и фиксированного типа дерева (исходное или инвертированное) задача формулируется так: "Дано дерево, на каждом ребре написано 0 или 1 (бит k). Для каждой вершины s найти количество вершин i, для которых XOR на пути $s \leftrightarrow i$ равен 1". Поскольку $d(s,i) = V_s \oplus V_i$, это просто количество вершин i с $V_i \neq V_s$.

Эффективный подсчёт $ans_0[s]_k$ и $ans_1[s]_k$

Для одного бита k и фиксированного состояния «инвертированы все рёбра или нет» потенциал каждой вершины V_u равен 0 или 1. Тогда число вершин i с $V_i \neq V_s$ равно

$$cnt_1, \text{ если } V_s = 0, \qquad cnt_0, \text{ если } V_s = 1,$$

где cnt_1 и cnt_0 — общее количество вершин с потенциалом 1 и 0 в данном состоянии. Следовательно, достаточно для каждого бита k отдельно знать:

- для исходного дерева ($X_k = 0$) — количества $cnt_0[k]$, $cnt_1[k]$ и потенциал V_u для всех вершин;
- для инвертированного дерева ($X_k = 1$) — количества $cnt'_0[k]$, $cnt'_1[k]$ и потенциал V'_u для всех вершин.

Потенциал инвертированного дерева легко выражается через исходный. Инверсия всех битов k на рёбрах означает, что вес каждого ребра изменился с w_k на $w_k \oplus 1$. Тогда XOR на пути до вершины u изменится на $depth[u] \bmod 2$ (число рёбер на пути, взятое по модулю 2), поэтому

$$V'_u = V_u \oplus (depth[u] \bmod 2),$$

где $depth[u]$ — количество рёбер от корня до u (глубина). Теперь предподсчёт делается за $O(30 \cdot n)$:

1. Запускаем DFS от корня (вершина 1). Передаём текущий XOR весов и глубину. Для каждой вершины u сохраняем $pot[u]$ (исходный потенциал) и $depth[u]$.
2. Для каждого бита $k = 0 \dots 29$:
   - Проходим по всем вершинам, извлекаем $b = (pot[u] \gg k) \,\&\, 1$. Увеличиваем $cnt_1[k]$, если $b = 1$, иначе $cnt_0[k]$.
   - Для инвертированного: $b' = b \oplus (depth[u] \,\&\, 1)$. Увеличиваем $cnt'_1[k]$, если $b' = 1$, иначе $cnt'_0[k]$.

Обработка запросов

Чтобы эффективно обрабатываться запросы нужно глобально хранить X (изначально 0) - накопление операций `x`.

Запрос `x`:

При поступлении запроса типа `x` с параметром x мы не изменяем веса рёбер явно. Вместо этого мы обновляем глобальную переменную X, которая накапливает суммарный эффект всех бурь. Поскольку последовательное применение бурь эквивалентно XOR их параметров (порядок не важен в силу ассоциативности и коммутативности XOR), достаточно выполнить $X := X \oplus x$. Позже, при обработке запроса `s`, мы учтём влияние X на потенциалы вершин, используя формулу с учётом чётности глубины.

Запрос `s`:

Для вершины s вычисляем текущий k-й бит её потенциала:

$$b_k(s) = \big((pot[s] \gg k)\ \&\ 1\big) \oplus \big(((X \gg k)\ \&\ 1) \cdot (depth[s]\ \&\ 1)\big).$$

Число единиц во всём дереве для бита k при текущем X_k равно

$$c_1(k) = \begin{cases} cnt_1[k], & \text{если } X_k = 0, \\ cnt'_1[k], & \text{если } X_k = 1. \end{cases}$$

Количество вершин с битом, противоположным $b_k(s)$, составляет

$$\text{diff}_k = \begin{cases} c_1(k), & b_k(s) = 0, \\ n - c_1(k), & b_k(s) = 1. \end{cases}$$

Итоговый ответ:

$$ans(s) = \sum_{k=0}^{29} 2^k \cdot \text{diff}_k.$$

Каждый запрос обрабатывается за $O(30)$, что легко укладывается в ограничения при $n, q \le 2 \cdot 10^5$.

Примечание: другое решение

Для каждого бита k и для каждого из двух состояний графа (исходный, где $X_k = 0$, и инвертированный, где все рёбра заменены на противоположные по k-му биту) задача решается независимо. Для фиксированного бита и фиксированного состояния графа выполняется два обхода дерева:

- Первый обход (снизу вверх). Для каждой вершины u вычисляется пара чисел (f_1, f_0) в её поддереве: сколько вершин i имеют потенциал, отличный от потенциала u (f_1), и сколько — совпадающий (f_0). При переходе от ребёнка v к родителю u эти величины корректируются с учётом бита ребра $t \in \{0, 1\}$. Если $t = 0$, то потенциалы родителя и ребёнка по данному биту совпадают, иначе — противоположны.
- Второй обход (сверху вниз). Зная ответ для корня, мы спускаемся к его детям и для каждого ребёнка v вычисляем его ответ уже для всего дерева. Для этого из ответа родителя вычитается вклад поддерева v, а оставшаяся часть комбинируется с уже известными значениями внутри поддерева v с учётом бита ребра между ними. Таким образом, после второго обхода для каждой вершины s становится известно итоговое количество ans_diff_s — число вершин i во всём дереве, для которых k-й бит $V_i \neq V_s$.

После того как для каждого бита k предподсчитаны два массива: $t_0[k][s] = ans\_diff_s$ для исходного графа и $t_1[k][s] = ans\_diff_s$ для инвертированного графа, ответ на запрос `2 s` собирается простым суммированием:

$$\text{ans} = \sum_{k=0}^{29} 2^k \cdot \big(X_k = 1\ ?\ t_1[k][s] : t_0[k][s]\big).$$

Реализация

```
void Solve() {
    int n, q; cin >> n >> q;
    vector<vector<pair<int, int>>> g(n + 1);
    for (int i = 0; i < n - 1; i++) {
        int u, v, w; cin >> u >> v >> w;
        g[u].push_back({ v, w });
        g[v].push_back({ u, w });
    }

    vector<int> pot(n + 1), depth(n + 1);
    function<void(int, int, int, int)> dfs = [&](int u, int p, int cur, int d) {
        pot[u] = cur;
        depth[u] = d;
        for (auto [v, w] : g[u]) {
            if (v != p) {
                dfs(v, u, cur ^ w, d + 1);
            }
        }
    };
    dfs(1, 0, 0, 0);

    vector<int> cnt1(30), cnt1_inv(30);
    for (int k = 0; k < 30; k++) {
        for (int u = 1; u <= n; u++) {
            int b = (pot[u] >> k) & 1;
            if (b == 1)
                cnt1[k]++;
            int b_inv = b ^ (depth[u] & 1);
            if (b_inv == 1)
                cnt1_inv[k]++;
        }
    }

    int X = 0;
    while (q--) {
        int t; cin >> t;
        if (t == 1) {
            int x; cin >> x; X ^= x;
        }
        else {
            int s; cin >> s;
            long long ans = 0;
            for (int k = 0; k < 30; k++) {
                int c1, diff;
                if (((X >> k) & 1) == 1)
                    c1 = cnt1_inv[k];
                else
                    c1 = cnt1[k];
                if ((((pot[s] >> k) & 1) ^ (((X >> k) & 1) & (depth[s] & 1))) == 0)
                    diff = c1;
                else
                    diff = n - c1;
                ans += (1LL << k) * diff;
            }
            cout << ans << '\n';
        }
    }
}
```

Задача «Массив Бесси»

Постановка задачи

У фермера Джона есть массив a длины n. Бесси разбивает массив на n/k групп по k элементов (подряд идущих). Затем она выполняет операции двух типов: сначала все операции типа 1, потом все операции типа 2.

- Тип 1 (циклический сдвиг групп влево): массив становится $a_{k+1}, a_{k+2}, \ldots, a_n, a_1, \ldots, a_k$.
- Тип 2 (обмен двух соседних групп): можно поменять местами группы i и $i+1$ (нельзя менять первую и последнюю).

Необходимо найти минимальное количество операций (по всем допустимым k и последовательностям операций с соблюдением порядка типов), за которое можно отсортировать массив.

Решение

Зафиксируем размер блока k (он обязан делить n, иначе деление невозможно). Обозначим $L = n/k$ — количество блоков. Заметим, что операции типа 1 действуют на все блоки одновременно (циклический сдвиг всего массива блоков), а операции типа 2 — это перестановки соседних блоков. Чтобы массив стал отсортированным, необходимо и достаточно:

1. Каждый блок i (подмассив с $a_{(i-1)k+1}$ по a_{ik}) сам по себе был неубывающим.
2. Если отсортировать блоки по значениям, то они не должны «пересекаться»: максимум предыдущего блока не должен превышать минимум следующего.

Если это выполнено, то задача сводится к сортировке массива b длины L, где b_i — некоторая метка блока i (например, пара $(\min, \max)$). Операции над блоками становятся операциями над элементами b:

- Тип 1 — циклический сдвиг массива b влево на 1.
- Тип 2 — обмен соседних элементов в b.

Чтобы минимизировать общее число операций, нужно выбрать количество циклических сдвигов t ($0 \le t < L$), выполнить их, а затем отсортировать полученный массив обменами соседних элементов. Минимальное число обменов равно числу инверсий в массиве после сдвигов.

Таким образом, для каждого допустимого k мы должны вычислить

$$\min_{0 \le t < L} \big(t + \text{inversions}(\text{rotate}_t(b))\big).$$

Мы умеем за $O(L \log L)$ находить начальное число инверсий. Затем, перебирая t от 1 до L, пересчитываем число инверсий после циклического сдвига на t. Для эффективного пересчёта: пусть мы знаем число инверсий для текущего состояния. При удалении первого элемента и добавлении его в конец число инверсий изменяется на

$$-\text{less} + \text{more},$$

где less — количество элементов в массиве, строго меньших удаляемого, а more — количество элементов, строго больших его. Эти величины легко получить через предподсчитанные частоты элементов.

Обновляя текущий счётчик инверсий cnt, для каждого t (начиная с 1) записываем кандидата $t + \text{cnt}$ в ответ. Исходный случай $t = 0$ даёт $\text{inversions}(b)$. Берём минимум по всем таким кандидатам $t + \text{cnt}$.

Реализация

```cpp
long long mergeinverse(vector<long long>& a) {
    long long inv_count = 0;
    if (a.size() > 1) {
        int mid = a.size() / 2;
        vector<long long> left(a.begin(), a.begin() + mid);
        vector<long long> right(a.begin() + mid, a.end());
        inv_count += mergeinverse(left);
        inv_count += mergeinverse(right);
        int i = 0, j = 0, k = 0;
        while (i < (int)left.size() && j < (int)right.size()) {
            if (left[i] <= right[j]) {
                a[k] = left[i]; i++;
            }
            else {
                a[k] = right[j]; j++;
                inv_count += (int)left.size() - i;
            }
            k++;
        }
        while (i < (int)left.size()) {
            a[k] = left[i]; i++; k++;
        }
        while (j < (int)right.size()) {
            a[k] = right[j]; j++; k++;
        }
    }
    return inv_count;
}

long long GET(vector<long long> a) {
    if (a.size() == 1) {
        return 0;
    }
    unordered_map<long long, int> M;
    for (int i = 0; i < (int)a.size(); i++) {
        M[a[i]] += 1;
    }
    vector<long long> s = a;
    long long cnt = mergeinverse(s);
    long long ans = cnt;
    unordered_map<long long, int> Less;
    for (int i = 1; i < (int)s.size(); i++) {
        if (s[i] != s[i - 1]) {
            Less[s[i]] = i;
        }
    }
    unordered_map<long long, int> More;
    for (int i = (int)s.size() - 2; i >= 0; i--) {
        if (s[i] != s[i + 1]) {
            More[s[i]] = (int)s.size() - 1 - i;
        }
    }
    for (int i = 0; i < (int)a.size(); i++) {
        cnt -= Less[a[i]];
        cnt += More[a[i]];
        ans = min(ans, cnt + i + 1);
    }
    return ans;
}
```

```cpp
vector<long long> FORM(vector<int>& a, int d) {
    int i = 0;
    vector<long long> ans;
    vector<pair<long long, long long>> p;
    while (i < (int)a.size()) {
        int l = i;
        int r = i + d - 1;
        vector<int> b;
        for (int j = l; j <= r; j++) {
            b.push_back(a[j]);
        }
        bool ok = true;
        for (int j = 1; j < (int)b.size(); j++) {
            if (b[j] < b[j - 1]) {
                ok = false;
            }
        }
        if (ok) {
            ans.push_back((long long)a[l] * 1000000000 + a[r]);
            p.push_back({ a[l], a[r] });
        }
        else {
            return {};
        }
        i += d;
    }
    sort(p.begin(), p.end());
    for (int i = 1; i < (int)p.size(); i++) {
        if (p[i].first < p[i - 1].second) {
            return {};
        }
    }
    return ans;
}

void Solve() {
    int n; cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    vector<int> D;
    for (int d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            D.push_back(d);
            if (n / d != d) {
                D.push_back(n / d);
            }
        }
    }
    long long ans = (long long)1e18;
    for (int d : D) {
        vector<long long> b = FORM(a, d);
        if (b.size() != 0) {
            ans = min(ans, GET(b));
        }
    }
    cout << ans << '\n';
}
```

Задача «Двойная звездная система»

Постановка задачи

Две одинаковые сферические планеты радиуса r вращаются по круговой орбите вокруг общего центра M, который находится в середине отрезка между их центрами. Планеты всегда расположены на противоположных концах одного диаметра орбиты. Наблюдатель находится в начале координат $O(0,0,0)$, лежащем в той же плоскости, что и орбита. Расстояние от O до M строго больше радиуса орбиты R, планеты не пересекаются.

Требуется провести прямую линию, проходящую через O, которая одновременно пересекает или касается обеих планет. Прямая считается допустимой, если она имеет хотя бы одну общую точку с каждой из двух сфер.

Планеты вращаются с постоянной угловой скоростью ω (направление вращения можно выбрать произвольно). Начальные координаты центров планет заданы. Необходимо найти минимальное время, через которое искомая прямая станет возможной, или определить, что это невозможно (вывести -1). Ответ требуется с точностью 10^{-4}.

Геометрический анализ

Обе планеты и точка O лежат в одной плоскости. Задача сводится к следующей: даны два круга радиуса r, центры которых симметричны относительно точки M и вращаются вокруг неё по окружности радиуса R. Нужно провести луч из O, пересекающий оба круга.

В качестве кандидата на искомую прямую рассмотрим прямую OM, проходящую через начало координат и центр вращения. Эта прямая обладает симметрией: если она пересекает одну планету, то автоматически пересекает и вторую.

Прямая OM пересекает планету (сферу) с центром C_1, если расстояние от C_1 до прямой OM не превышает радиуса r. Обозначим через α острый угол между векторами $\overrightarrow{OM}$ и $\overrightarrow{MC_1}$. Тогда расстояние от C_1 до прямой OM равно $R\sin\alpha$ (так как M лежит на прямой). Условие пересечения:

$$R\sin\alpha \le r \quad\Longleftrightarrow\quad \alpha \le \arcsin\frac{r}{R}.$$

Ключевой факт: если существует хоть какая-нибудь прямая ℓ из O, пересекающая обе планеты, то обязательно прямая OM тоже их пересекает. Это можно доказать, используя выпуклость расстояния или свойства симметрии. В результате задача сводится к наблюдению за углом между $\overrightarrow{MC_1}$ и $\overrightarrow{MO}$ и ожиданию момента, когда он станет не больше $\arcsin(r/R)$.

Вычисление угла α

Пусть C_1, C_2 — текущие координаты центров планет. Найдём $M = \frac{C_1+C_2}{2}, V = C_1 - M$. Тогда $R = |V|, D = |OM|$. Косинус угла между векторами $\overrightarrow{OM}$ и V равен

$$\cos\angle(\overrightarrow{OM}, V) = \frac{\overrightarrow{OM}\cdot V}{D\cdot R}.$$

Острый угол α получается как arccos абсолютного значения этого косинуса (поскольку расстояние зависит от синуса, а $\sin\theta = \sin(\pi - \theta)$):

$$\alpha = \arccos \frac{|\overrightarrow{OM} \cdot V|}{D \cdot R}.$$

Учёт вращения

Планеты вращаются вокруг M с угловой скоростью ω. При вращении вектор V поворачивается, и угол α изменяется со скоростью ω. Выбирая направление вращения, мы можем как увеличивать, так и уменьшать α. Если в начальный момент уже выполнено $\alpha \le \arcsin(r/R)$, то искомая прямая доступна мгновенно, время $t = 0$.

В противном случае нужно дождаться момента, когда α уменьшится до критического значения $\varphi = \arcsin(r/R)$. Минимальное время достигается при вращении в сторону уменьшения угла и равно

$$t = \frac{\alpha - \varphi}{\omega}.$$

Существование решения

Поскольку планеты могут свободно вращаться на полный оборот, вектор V может принять любое направление в плоскости. В частности, можно совместить планету с лучом OM (тогда $\alpha = 0$), что заведомо удовлетворяет условию. Следовательно, требуемое положение всегда достижимо, и ответ -1 никогда не потребуется при заданных ограничениях.

Реализация

```cpp
double F(double x) {
    if (x > 1.0) {
        return 1.0;
    }
    if (x < -1.0) {
        return -1.0;
    }
    return x;
}

void Solve() {
    double x1, y1, z1, x2, y2, z2, r, w;
    cin >> x1 >> y1 >> z1 >> x2 >> y2 >> z2 >> r >> w;

    double mx = (x1 + x2) / 2, my = (y1 + y2) / 2, mz = (z1 + z2) / 2;
    double vx = x1 - mx, vy = y1 - my, vz = z1 - mz;
    double D = sqrt(mx * mx + my * my + mz * mz);
    double R = sqrt(vx * vx + vy * vy + vz * vz);

    double a = acos(fabs(F((vx * mx + vy * my + vz * mz) / (R * D))));

    if (a <= asin(min(r / R, 1.0))) {
        cout << fixed << setprecision(4) << 0.0 << '\n';
    }
    else {
        cout << fixed << setprecision(4) << (a - asin(min(r / R, 1.0))) / w << '\n';
    }
}
```

Задача «Игра со строками»

Постановка задачи

В магазине продаются строки n видов, i-й вид описывается строкой s_i и ценой c_i за одну копию. Можно купить любое количество экземпляров каждого вида, заплатив c_i за каждый экземпляр. Требуется найти минимальный суммарный бюджет, при котором из купленных строк можно составить хотя бы один палиндром, конкатенируя их в некотором порядке. Если это невозможно ни при каком наборе, вывести -1.

Динамическое построение палиндрома

Палиндром можно строить одновременно с двух сторон: на каждом шаге имеется уже построенная центральная часть, которая является палиндромом, а слева и справа к ней добавляются одинаковые (с точностью до разворота) строки. Однако в обратном порядке процесс выглядит так: начинаем с некоторой строки, которая будет центром, и постепенно «раскрываем» её, добавляя к краям пары купленных строк (одна — в прямом направлении, другая — в обратном). Формально, пусть мы хотим получить палиндром P. Разрежем его мысленно на три части: левую L, центральную C и правую $R = \mathrm{rev}(L)$, где $\mathrm{rev}(X)$ — развёрнутая строка. Тогда $P = L + C + \mathrm{rev}(L)$. Если C также можно представить в виде аналогичной конструкции, то весь палиндром раскладывается в последовательность вложенных пар строк.

Удобно рассмотреть процесс с точки зрения «невязки» между левой и правой половинами. Представим, что мы строим палиндром, добавляя строки по одной то к левому, то к правому концу, и на каждом этапе одна из половин может быть длиннее другой. Разность между ними можно описать одной строкой — остатком более длинной половины. Если левая половина длиннее на $|v|$ символов, то этот остаток равен последним $|v|$ символам левой половины; если правая длиннее — остатком является развёрнутая «лишняя» часть правой половины. При добавлении очередной купленной строки к одной из сторон остаток определённым образом изменяется.

Таким образом, состояние системы естественно задавать строкой остатка и указанием на то, с какой стороны он образовался (левая половина длиннее или правая). Состояние будем обозначать парой (v, dir), где $\mathrm{dir} \in \{L, R\}$. Если $\mathrm{dir} = L$, это означает, что левая часть длиннее правой на $|v|$ символов и её «хвост» равен v; если $\mathrm{dir} = R$ — правая часть длиннее, и её «хвост» после разворота равен v. Пустое состояние (полный баланс, $|v| = 0$) естественно считать одним и тем же для обоих направлений.

Начальное состояние — пустой остаток, то есть полный баланс. Это соответствует тому, что у нас ещё нет никакой построенной части. Цель — добиться того, чтобы после добавления некоторой заключительной строки остаток вновь стал пустым. Более конкретно, заключительной строкой может выступать любая из исходных строк s_i (купленная последней), которая будет помещена в центр. Тогда непосредственно перед её добавлением остаток должен быть равен s_i (если она добавляется к более короткой стороне) или $\mathrm{rev}(s_i)$ (если к более длинной). В силу симметрии достаточно рассмотреть случай, когда остаток равен s_i и он находится на левой стороне. Поэтому ответом будет минимальная сумма стоимостей купленных строк, позволяющая перейти из пустого остатка в состояние (s_i, L), плюс стоимость самой s_i.

Построение графа состояний

Остаток может принимать не произвольные значения, а лишь те, которые получаются как суффиксы или префиксы исходных строк при попытке «сократить» их. Поэтому множество всех возможных остатков V — это все строки, которые можно получить, взяв любую исходную строку s_i и вырезав из неё любой префикс или любой суффикс (включая пустую строку и саму s_i). Формально,

$$V = \bigcup_{i=1}^{n} \{\, s_i[0 \dots j] \mid 0 \le j \le |s_i| \,\} \cup \{\, s_i[j \dots] \mid 0 \le j \le |s_i| \,\}.$$

(Здесь $s[0 \dots j]$ – префикс длины j, $s[j \dots]$ – суффикс, начинающийся с позиции j.)

Для каждой строки $v \in V$ введём две вершины графа: v_L (остаток v на левой стороне) и v_R (остаток v на правой стороне). Вершину, соответствующую пустой строке, можно считать общей для обоих типов.

Теперь для каждой вершины и каждой исходной строки s с ценой c определим, в какие вершины можно перейти, добавив s к текущей конструкции. Рассмотрим случай, когда текущий остаток — это v на левой стороне (т.е. левая половина длиннее на $|v|$). Чтобы сократить разницу, мы можем добавить s справа. При этом необходимо, чтобы добавляемая строка «покрывала» остаток v в обратном направлении. Возможны два подслучая.

1. **Строка s не длиннее остатка ($|s| \le |v|$).** Тогда $\mathrm{rev}(s)$ должна совпадать с префиксом v длины $|s|$. Если это так, то после добавления s справа остаток уменьшится: новый остаток будет равен v без этого префикса, и он по-прежнему останется на левой стороне. Получаем переход

$$v_L \xrightarrow{c} (v_{|s|\dots})_L.$$

2. **Строка s длиннее остатка ($|s| > |v|$).** Тогда $\mathrm{rev}(s)$ должна начинаться с v. После добавления s справа левый остаток полностью компенсируется, и появляется новый остаток на правой стороне — это «избыток» строки s, который не понадобился для покрытия. А именно, если $\mathrm{rev}(s) = v + u$ (приписывание), то новый остаток на правой стороне будет равен $\mathrm{rev}(u)$. Получаем переход

$$v_L \xrightarrow{c} (\mathrm{rev}(u))_R.$$

Симметрично, если текущий остаток v на правой стороне, можно добавлять s слева. Тогда условия и переходы зеркальны:

- При $|s| \le |v|$: нужно, чтобы s совпадала с префиксом v (длины $|s|$), и остаток сокращается до $v_{|s|\dots}$ на правой стороне.
- При $|s| > |v|$: нужно, чтобы s начиналась с v; тогда остаток переходит на левую сторону и становится равным $\mathrm{rev}(s_{|v|\dots})$.

Кроме того, отдельно обрабатывается ситуация, когда текущий остаток пуст. В этом случае добавление строки s с любой стороны немедленно создаёт остаток, равный s (на противоположной стороне) или $\mathrm{rev}(s)$ (на той же стороне). Однако для целей нашей модели удобнее поступить иначе: пустое состояние будем считать нейтральным, а переходы из него можно получить как частный случай описанных выше правил.

Бесплатные переходы в палиндромы

Если некоторая строка v сама является палиндромом, то её можно поместить в центр «бесплатно» в том смысле, что она не требует дополнительной балансировки: в состоянии, когда остаток равен v на любой стороне, мы можем сразу же использовать этот остаток как центральную часть. Формально, для любого палиндрома $v \in V$ разрешим переход из начальной (пустой) вершины в v_L и в v_R с нулевой стоимостью. Это соответствует тому, что мы можем выбрать в качестве центральной части палиндром, который позже будет оплачен (как последняя строка s_i, если $v = s_i$), либо, возможно, составлен из других строк и поэтому не требует отдельной покупки.

Поиск минимальной стоимости

Построенный ориентированный граф имеет $O(|V|)$ вершин и $O(n \cdot |V|)$ рёбер. Поскольку длины строк не превышают 20, общее число состояний невелико (не более нескольких тысяч). Веса всех рёбер неотрицательны. С помощью алгоритма Дейкстры находим кратчайшие расстояния от начальной вершины (пустой остаток) до всех остальных. Обозначим через $\mathrm{dist}[v_L]$ минимальную стоимость достижения состояния v_L.

Ответом будет

$$\min_{i=1}^{n}\bigl(\mathrm{dist}[(s_i)_L] + c_i\bigr),$$

при условии, что вершина $(s_i)_L$ существует в графе (т.е. $s_i \in V$). Если ни одна из таких величин не конечна, ответ -1.

Реализация

```
bool pal(string s) {
    int n = s.size();
    for (int i = 0; i < n / 2; i++) {
        if (s[i] != s[n - 1 - i]) {
            return false;
        }
    }
    return true;
}

void Solve() {
    int n; cin >> n;
    vector<string> S(n);
    vector<long long> C(n);
    for (int i = 0; i < n; i++) {
        cin >> S[i] >> C[i];
    }

    unordered_set<string> vs; vs.insert("");
    for (int i = 0; i < n; i++) {
        int len = S[i].size();
        for (int j = 0; j <= len; j++) {
            vs.insert(S[i].substr(0, j));
            vs.insert(S[i].substr(j));
        }
    }

    vector<string> vv(vs.begin(), vs.end());
    unordered_map<string, int> lb;
    int m = vv.size();
    for (int i = 0; i < m; i++) {
        lb[vv[i]] = i;
    }

    int st = lb[""];
    vector<vector<pair<int, long long>>> G(2 * m);

    for (auto p : lb) {
        string v = p.first;
        int i = p.second;
        if (pal(v)) {
            G[st].push_back({ i, 0 });
            G[st].push_back({ i + m, 0 });
        }
    }
```

```
46      for (auto p : lb) {
47          string v = p.first;
48          int i = p.second;
49          int a = v.size();
50
51          for (int id = 0; id < n; id++) {
52              string s = S[id];
53              long long c = C[id];
54              int b = s.size();
55
56              if (a < b) {
57                  string rs = s;
58                  reverse(rs.begin(), rs.end());
59                  if (rs.substr(0, a) == v) {
60                      string k = rs.substr(a);
61                      reverse(k.begin(), k.end());
62                      if (lb.count(k)) {
63                          int j = lb[k];
64                          G[j + m].push_back({ i, c });
65                      }
66                  }
67                  string sp = s.substr(0, a);
68                  reverse(sp.begin(), sp.end());
69                  if (sp == v) {
70                      string k = s.substr(a);
71                      if (lb.count(k)) {
72                          int j = lb[k];
73                          G[j].push_back({ i + m, c });
74                      }
75                  }
76              }
77              else {
78                  string vp = v.substr(0, b);
79                  reverse(vp.begin(), vp.end());
80                  if (vp == s) {
81                      string k = v.substr(b);
82                      if (lb.count(k)) {
83                          int j = lb[k];
84                          G[j].push_back({ i, c });
85                      }
86                  }
87                  string vr = v;
88                  reverse(vr.begin(), vr.end());
89                  string vrp = vr.substr(0, b);
90                  if (vrp == s) {
91                      string k = vr.substr(b);
92                      reverse(k.begin(), k.end());
93                      if (lb.count(k)) {
94                          int j = lb[k];
95                          G[j + m].push_back({ i + m, c });
96                      }
97                  }
98              }
99          }
100     }
101
102     vector<long long> dst(2 * m, 1e18);
103     dst[st] = 0;
104     priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>> pq;
105     pq.push({ 0, st });
106
```

```
107     while (!pq.empty()) {
108         auto [d, u] = pq.top();
109         pq.pop();
110         if (d != dst[u]) {
111             continue;
112         }
113         for (auto [v, w] : G[u]) {
114             if (dst[v] > d + w) {
115                 dst[v] = d + w;
116                 pq.push({ dst[v], v });
117             }
118         }
119     }
120 
121     long long ans = 1e18;
122     for (int i = 0; i < n; i++) {
123         if (lb.count(S[i])) {
124             int id = lb[S[i]];
125             if (dst[id] != 1e18) {
126                 ans = min(ans, dst[id] + C[i]);
127             }
128         }
129     }
130 
131     if (ans == 1e18) {
132         cout << -1;
133     }
134     else {
135         cout << ans;
136     }
137 }
```

Задача «Яблоневый сад»

Постановка задачи

В ряду растут n яблонь, на i-й яблоне a_i яблок. Сбор фруктов с i-го дерева занимает время, равное сумме яблок на соседних деревьях в момент сбора:

$$t_i = a_{i-1} + a_{i+1},$$

где $a_0 = a_{n+1} = 0$. Сразу после сбора a_i становится равным нулю.

Требуется собрать все яблоки, потратив наименьшее суммарное время, и найти количество различных порядков сбора (перестановок $1 \dots n$), при которых достигается это минимальное время. Ответ выводится по модулю $10^9 + 7$.

Выражение общего времени через порядок

Рассмотрим два соседних дерева i и $i+1$. Относительный порядок их сбора полностью определяет вклад этой пары в общее время. Если i собирается раньше $i+1$, то на момент сбора i сосед $i+1$ ещё содержит a_{i+1} яблок, и время t_i включает a_{i+1}. Позднее, при сборе $i+1$, дерево i уже пусто, поэтому t_{i+1} не содержит a_i. Суммарный вклад пары в этом случае равен a_{i+1}.

Если же $i+1$ собирается раньше i, то время сбора $i+1$ включает a_i, а позднее при сборе i дерево $i+1$ уже пусто. Вклад равен a_i.

Таким образом, для каждого ребра $(i, i+1)$ общий вклад в целевую функцию составляет

$$\begin{cases} a_{i+1}, & \text{если } i \text{ раньше } i+1, \\ a_i, & \text{если } i+1 \text{ раньше } i. \end{cases}$$

Никакие другие пары деревьев не влияют на эту величину. Следовательно, общее время сбора равно сумме по всем $i = 1 \dots n-1$ выбранного вклада, и разные пары независимы.

Минимизация времени

Для каждого ребра мы можем выбрать направление, дающее меньшее из двух чисел. Если $a_i < a_{i+1}$, выгоднее заплатить a_i, т.е. выбрать порядок $i+1$ раньше i. Если $a_i > a_{i+1}$, выгоднее заплатить a_{i+1}, т.е. порядок i раньше $i+1$. При $a_i = a_{i+1}$ оба варианта дают одинаковый вклад a_i, и порядок на этом ребре может быть любым.

Таким образом, минимальное возможное время однозначно и равно

$$T_{\min} = \sum_{i=1}^{n-1} \min(a_i, a_{i+1}).$$

Чтобы достичь этого времени, необходимо и достаточно для каждого ребра выполнить соответствующее условие предшествования:

- если $a_i > a_{i+1}$, то i должно идти раньше $i+1$;
- если $a_i < a_{i+1}$, то $i+1$ должно идти раньше i;
- если $a_i = a_{i+1}$, ограничений нет.

Подсчёт числа подходящих перестановок

Мы получили частичный порядок на множестве элементов $\{1, \ldots, n\}$. Граф отношений представляет собой ориентированные рёбра вдоль линии: для каждого i либо $i \to i+1$, либо $i+1 \to i$, либо ребро отсутствует. Этот граф не содержит циклов (направления согласованы с перепадами значений a_i). Задача свелась к нахождению количества линейных расширений (топологических сортировок) данного частичного порядка.

Решим её динамическим программированием, добавляя деревья по порядку их номеров и поддерживая распределение позиции самого правого на данный момент элемента.

Пусть $\mathrm{dp}[i][p]$ — количество перестановок первых i элементов, удовлетворяющих всем условиям на рёбрах $1 \ldots i-1$, в которых элемент i стоит на позиции p $(1 \le p \le i)$. Изначально для $i = 1$ имеем $\mathrm{dp}[1][1] = 1$.

При добавлении элемента $i+1$ с известным отношением к i возможны переходы:

- Если $a_i > a_{i+1}$: элемент $i+1$ обязан идти после i. Он может занять любую позицию правее p. Если старый i был на позиции p, то допустимы новые позиции $q \in \{p+1, \ldots, i+1\}$. Суммируем способы:

$$\mathrm{dp}[i+1][q] = \sum_{p=1}^{q-1} \mathrm{dp}[i][p].$$

  Это префиксная сумма.

- Если $a_i < a_{i+1}$: элемент $i+1$ обязан идти раньше i. Он может занять позиции от 1 до p включительно.

$$\mathrm{dp}[i+1][q] = \sum_{p=q}^{i} \mathrm{dp}[i][p].$$

  Это суффиксная сумма.

- Если $a_i = a_{i+1}$: допустима любая позиция q $(1 \le q \le i+1)$, и все значения $\mathrm{dp}[i][p]$ равномерно распределяются:

$$\mathrm{dp}[i+1][q] = \sum_{p=1}^{i} \mathrm{dp}[i][p] = \text{общая сумма}.$$

После обработки всех n элементов ответом будет сумма $\mathrm{dp}[n][p]$ по всем позициям $p = 1 \ldots n$.

Реализация

```
void Solve() {
    int MOD = 1000000007;

    int n; cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    if (n == 1) {
        cout << 1;
        return;
    }

    vector<long long> cur(2);
    cur[1] = 1;


```

```
    for (int i = 0; i < n - 1; i++) {
        int len = i + 1;
        vector<long long> nxt(len + 2);

        if (a[i] > a[i + 1]) {
            long long pf = 0;
            for (int k = 1; k <= len; k++) {
                pf = (pf + cur[k]) % MOD;
                nxt[k + 1] = pf;
            }
        }
        else if (a[i] < a[i + 1]) {
            long long sf = 0;
            for (int k = len; k >= 1; k--) {
                sf = (sf + cur[k]) % MOD;
                nxt[k] = sf;
            }
        }
        else {
            long long tt = 0;
            for (int k = 1; k <= len; k++) {
                tt = (tt + cur[k]) % MOD;
            }
            for (int k = 1; k <= len + 1; k++) {
                nxt[k] = tt;
            }
        }
        cur = move(nxt);
    }

    long long ans = 0;
    for (int k = 1; k <= n; k++) {
        ans = (ans + cur[k]) % MOD;
    }
    cout << ans;
}
```

Задача «Комплексный обед»

Постановка задачи

Даны два набора по n точек на плоскости (координаты целые, $n \le 100$). Первый набор — (x_i, y_i), второй — (xx_i, yy_i), причём внутри каждого набора все точки различны. Над одним из наборов можно совершать две операции:

- Rotate(p) — повернуть все точки вокруг начала координат на p градусов по часовой стрелке ($0 \le p \le 360$);
- Move(r, d) — сдвинуть все точки на r по оси Ox и на d по оси Oy.

Операции можно применять в любом порядке и любое количество раз. Два набора называются "R-равными если можно применить несколько таких операций к одному из них и получить набор, в котором для каждого i координаты i-й точки совпадают с i-й точкой другого набора. Требуется по заданным наборам определить, являются ли они R-равными.

Решение

Поворот вокруг начала координат и последующий сдвиг — это движение плоскости, сохраняющее расстояния между точками. Поэтому если два набора переводятся друг в друга такими операциями, то расстояния между соответствующими парами точек должны сохраняться.

Случай с одной точкой весьма скучный - можно показать, что ответ всегда Yes. При $n \ge 2$ рассмотрим первые две точки каждого набора. Возьмём разности:

- В первом наборе: вектор от первой точки ко второй.
- Во втором наборе: вектор от первой точки ко второй.

Поворот, переводящий один набор в другой, обязан переводить первый вектор во второй. Значит, длины этих векторов должны совпадать. Если длины разные — ответ No.

Если длины совпадают, можно однозначно определить поворот, переводящий первый вектор во второй, и затем проверить, переводит ли этот же поворот все остальные точки первого набора в соответствующие точки второго (после учёта сдвига).

Формально, пусть (x_1, y_1), (x_2, y_2) — первые две точки первого набора, а (xx_1, yy_1), (xx_2, yy_2) — первые две точки второго. Образуем векторы:

$$(x_u, y_u) = (x_2 - x_1,\ y_2 - y_1), \qquad (x_v, y_v) = (xx_2 - xx_1,\ yy_2 - yy_1).$$

Квадрат длины векторов: $L = x_u^2 + y_u^2 = x_v^2 + y_v^2$.

Теперь для каждой точки i ($i = 1 \dots n$) образуем её вектор относительно первой точки своего набора. Для первого набора это $(x_i - x_1,\ y_i - y_1)$, для второго — $(xx_i - xx_1,\ yy_i - yy_1)$. Поворот, переводящий (x_u, y_u) в (x_v, y_v), действует на произвольный вектор (w_x, w_y) по формулам:

$$(\text{новый } x) = \frac{(x_u x_v + y_u y_v)\, w_x - (x_u y_v - y_u x_v)\, w_y}{L},$$

$$(\text{новый } y) = \frac{(x_u y_v - y_u x_v)\, w_x + (x_u x_v + y_u y_v)\, w_y}{L}.$$

Подставляя сюда $w_x = x_i - x_1$, $w_y = y_i - y_1$ и сравнивая с $(xx_i - xx_1,\ yy_i - yy_1)$, получаем условия:

$$(x_u x_v + y_u y_v)(x_i - x_1) - (x_u y_v - y_u x_v)(y_i - y_1) = L\,(xx_i - xx_1),$$
$$(x_u y_v - y_u x_v)(x_i - x_1) + (x_u x_v + y_u y_v)(y_i - y_1) = L\,(yy_i - yy_1).$$

Обе части должны быть целыми, причём левая часть обязана делиться на L нацело.

Если для всех i эти равенства выполнены, то после поворота останется только сдвинуть весь набор на вектор (xx_1, yy_1) минус повёрнутое (x_1, y_1), и наборы совпадут. Ответ Yes.

Если хотя бы для одного i левая часть не делится на L или частное не совпадает с правой частью — ответ <No.

Реализация

```
void Solve() {
    int n; cin >> n;
    vector<pair<int, int>> A(n), B(n);
    for (int i = 0; i < n; i++) {
        cin >> A[i].first >> A[i].second;
    }
    for (int i = 0; i < n; i++) {
        cin >> B[i].first >> B[i].second;
    }

    if (n == 1) {
        cout << "Yes"; return;
    }

    int dxa = A[1].first - A[0].first;
    int dya = A[1].second - A[0].second;
    int dxb = B[1].first - B[0].first;
    int dyb = B[1].second - B[0].second;

    int LA = dxa * dxa + dya * dya;
    int LB = dxb * dxb + dyb * dyb;
    if (LA != LB) {
        cout << "No"; return;
    }

    int dot = dxa * dxb + dya * dyb;
    int crs = dxa * dyb - dya * dxb;
    int L = LA;

    for (int i = 0; i < n; i++) {
        int dx = A[i].first - A[0].first;
        int dy = A[i].second - A[0].second;
        int X = dot * dx - crs * dy;
        int Y = crs * dx + dot * dy;
        if (X % L != 0 || Y % L != 0) {
            cout << "No"; return;
        }
        int ex = X / L;
        int ey = Y / L;
        if (ex != B[i].first - B[0].first || ey != B[i].second - B[0].second) {
            cout << "No"; return;
        }
    }

    cout << "Yes";
}
```

Задача «Симметричный элемент»

Постановка задачи

Дан массив $a_1, \ldots, a_n$. Для каждой позиции i требуется найти минимальное натуральное t, такое что $1 \le i - t \le i + t \le n$ и $a_{i-t} = a_{i+t}$. Если такого t нет, вывести -1. Ограничения: $1 \le n \le 10^5$, $|a_i| \le 10^5$.

Разбиение на группы

Условие $a_{i-t} = a_{i+t}$ означает, что позиции $p = i - t$ и $q = i + t$ содержат одинаковые значения. При этом $p + q = 2i$. Следовательно, для каждого значения v и любой пары позиций $p < q$ с $a_p = a_q = v$ индекс $i = (p + q)/2$ получает кандидат $t = (q - p)/2$ для ответа (если i целое). Нам нужно для каждого i найти минимальное такое t среди всех пар с одинаковым значением, симметричных относительно i. Заметим, что i целое тогда и только тогда, когда p и q имеют одинаковую чётность. Поэтому позиции одного значения естественно разбить на две группы: чётные и нечётные. Для каждой группы кандидатов будем искать отдельно.

Для пары чётных позиций $p = 2x$, $q = 2y$ $(x < y)$ имеем $i = x + y$ и $t = y - x$. Для пары нечётных позиций $p = 2x - 1$, $q = 2y - 1$ $(x < y)$ получаем $i = x + y - 1$ и $t = y - x$.

Таким образом, для каждого значения v выделим все его позиции. Разобьём их на две подгруппы по чётности индекса. В каждой подгруппе построим массив X преобразованных координат:

- для чётной подгруппы: $X = \{\, p/2 \mid p$ — чётная позиция $\}$;
- для нечётной подгруппы: $X = \{\, (p + 1)/2 \mid p$ — нечётная позиция $\}$.

Тогда для любой пары элементов $x, y \in X$ с $x < y$ получаем кандидат:

$$i = x + y - \tau, \qquad t = y - x,$$

где $\tau = 0$ для чётной подгруппы и $\tau = 1$ для нечётной. Задача свелась к следующей: для каждой подгруппы (конкретных τ и множества X) нужно для всех пар (x, y) с $x < y$ обновить ответ для индекса $i = x + y - \tau$ значением $t = y - x$, если оно меньше уже записанного. Изначально все ответы равны $+\infty$.

Размер множества X обозначим m. Если действовать перебором всех пар, время для одной подгруппы будет $O(m^2)$, что в сумме по всем подгруппам может достигать $O(n^2)$ в худшем случае (например, все элементы равны). Чтобы уложиться в ограничения, введем параметр B.

- Если $m \le B$, подгруппа считается лёгкой.
- Если $m > B$, подгруппа тяжёлая.

Обработка лёгких подгрупп

Для лёгкой подгруппы просто перебираем все пары индексов $u < v$ в массиве X и обновляем $ans[x_u + x_v - \tau]$ значением $x_v - x_u$. Суммарное время всех лёгких подгрупп оценивается следующим образом. Пусть $m_1, \ldots, m_k$ — размеры лёгких подгрупп, $\sum m_i = N_{\text{лег}} \le n$. Так как каждый $m_i \le B$, максимум суммы квадратов $\sum m_i^2$ при фиксированной сумме достигается, когда как можно больше подгрупп имеют размер B. Тогда $\sum m_i^2 \le (n/B) \cdot B^2 = nB$.

Обработка тяжёлых подгрупп с помощью bitset

Тяжёлая подгруппа имеет размер $m > B$. Для неё перебор пар за $O(m^2)$ уже слишком велик. Вместо этого воспользуемся тем, что все элементы X — целые числа из ограниченного диапазона. Тогда множество всевозможных сумм $x_u + x_v$ также ограничено: максимальная сумма не превосходит $2 \cdot \max X \le n + 1$. Это позволяет применить bitset.

Рассмотрим массив X в порядке возрастания (это естественный порядок, так как мы заносили позиции исходного массива последовательно). Пройдём по элементам с конца (от наибольшего к наименьшему). Будем поддерживать bitset tail, в котором отмечены уже просмотренные элементы (т.е. элементы, бóльшие текущего). При обработке текущего элемента x_{id} (id убывает) все возможные суммы $S = x_{id}+y$ для $y \in tail$ (где $y > x_{id}$) могут быть получены сдвигом $tail$ влево на x_{id}: ($tail \ll x_{id}$) содержит биты на позициях $S = x_{id} + y$.

Важно, что одна и та же сумма S может быть получена из разных пар (x, y) с различным расстоянием $t = y - x$. Для фиксированной суммы S и фиксированного текущего $x = x_{id}$ значение $t = S - 2x$. Поскольку мы идём от больших x к меньшим, величина $2x$ убывает, следовательно $t = S - 2x$ возрастает при движении по убыванию x. Это означает, что первая же пара, породившая сумму S при обходе с конца, даст минимальное возможное t для этой суммы. Действительно, если для той же S позже (т.е. при меньшем x') будет найдена другая пара (x', y'), то $t' = S - 2x' > S - 2x = t$. Значит, минимальное расстояние достигается при максимальном x, участвующем в сумме S.

Поэтому мы можем завести bitset seen, в котором будем отмечать уже найденные суммы S. При обработке $x = X[id]$ вычислим $new\_sums = (tail \ll x)$ & $\sim seen$ — множество сумм, порождённых с текущим x и ещё не встречавшихся ранее. Для каждой такой суммы S обновим ответ для индекса $i = S - \tau$ значением $t = S - 2x$ и отметим эти суммы в seen: $seen \mid = new\_sums$, после чего добавим текущий x в $tail$. Таким образом, каждая возможная сумма S будет обработана ровно один раз (когда она впервые появляется), и при этом ей будет сопоставлено минимальное t среди всех пар, дающих эту сумму.

Пусть подгруппа имеет размер m. Для каждого из m элементов выполняется сдвиг и операция над bitset размера $O(n)$. Это даёт $O(m \cdot n/64)$ битовых операций. Кроме того, итерация по установленным битам в new_sums суммарно потребует $O(n)$ действий на всю подгруппу (каждый бит будет просмотрен ограниченное число раз). Итоговая сложность для одной тяжёлой подгруппы $O(m \cdot n/64 + n)$.

Суммируя по всем тяжёлым подгруппам, получаем общее время

$$\sum_{\text{тяж}} \big(O(m_i \cdot n/64) + O(n)\big) = O\big((n/64) \sum m_i + Kn\big),$$

где K — количество тяжёлых подгрупп. Поскольку $\sum m_i = N_{\text{тяж}} \le n$, первое слагаемое есть $O(n^2/64)$. Так как каждая тяжёлая подгруппа имеет размер $m_i > B$, их число $K \le n/B$. Второе слагаемое $Kn \le n^2/B$.

Таким образом, итоговая сложность для тяжёлой части составляет

$$O\left(\frac{n^2}{64} + \frac{n^2}{B}\right).$$

Выбор параметра B

Суммарная сложность алгоритма складывается из трёх частей: обработка лёгких подгрупп: $O(nB)$; обработка тяжёлых подгрупп, первое слагаемое: $O(n^2/64)$; обработка тяжёлых подгрупп, второе слагаемое: $O(n^2/B)$.

Первое слагаемое тяжёлой части не зависит от B и составляет $O(n^2/64)$ — это неизбежная плата за использование bitset. Оставшиеся два слагаемых зависят от B: с ростом B растёт время лёгкой части, но убывает второе слагаемое тяжёлой.

Чтобы сбалансировать эти величины, приравняем nB и n^2/B:

$$nB \approx \frac{n^2}{B} \quad \Longrightarrow \quad B^2 \approx n \quad \Longrightarrow \quad B \approx \sqrt{n}.$$

При таком выборе слагаемые nB и n^2/B оба имеют порядок $n\sqrt{n}$, а итоговая сложность составляет

$$O\left(\frac{n^2}{64} + n\sqrt{n}\right).$$

Реализация

```
int B = 300; int MAXS = 100005;

void go(vector<int>& X, int tp, int n, vector<int>& ans) {
    int m = X.size();
    if (m <= B) {
        for (int i = 0; i < m; i++) {
            for (int j = i + 1; j < m; j++) {
                int mid = X[i] + X[j] - (tp == 1);
                if (mid >= 1 && mid <= n && X[j] - X[i] < ans[mid]) {
                    ans[mid] = X[j] - X[i];
                }
            }
        }
    }
    else {
        bitset<100005> tail, seen;
        for (int id = m - 1; id >= 0; id--) {
            bitset<100005> nw = (tail << X[id]) & ~seen;
            for (int S = nw._Find_first(); S != nw.size(); S = nw._Find_next(S)) {
                int mid = S - (tp == 1);
                if (mid >= 1 && mid <= n) {
                    ans[mid] = min(ans[mid], S - 2 * X[id]);
                }
            }
            seen |= nw; tail.set(X[id]);
        }
    }
}

void Solve() {
    int n; cin >> n;
    vector<pair<int, int>> pos;
    for (int i = 1; i <= n; i++) {
        int v; cin >> v;
        pos.push_back({ v, i });
    }
    sort(pos.begin(), pos.end());

    vector<int> ans(n + 1, 1e9);
    int i = 0;
    while (i < n) {
        int j = i;
        vector<int> P;
        while (j < n && pos[j].first == pos[i].first) {
            P.push_back(pos[j].second); j++;
        }
        vector<int> ev, od;
        for (int p : P) {
            if (p % 2 == 0) ev.push_back(p / 2);
            else od.push_back((p + 1) / 2);
        }
        if (!ev.empty()) go(ev, 0, n, ans);
        if (!od.empty()) go(od, 1, n, ans);
        i = j;
    }

    for (int i = 1; i <= n; i++) cout << (ans[i] == 1e9 ? -1 : ans[i]) << ' ';
}
```

Задача «Участие в раундах»

Постановка задачи

Игрок начинает с рейтингом $x = 0$. Дано n раундов, i-й раунд задаётся парой (a_i, b_i):

- a_i — максимальный рейтинг, с которым можно войти в раунд;
- b_i — новое значение рейтинга после раунда: $x \leftarrow \max(x, b_i)$.

Участвовать в раунде можно только если $x \le a_i$. Каждый раунд можно пройти не более одного раза, порядок выбирает игрок. Требуется найти максимальное количество раундов, которое можно пройти.

Классификация раундов и порядок прохождения

Разобьём раунды на два типа:

- Слабые: $a_i \ge b_i$.
- Сильные: $a_i < b_i$.

Сильный раунд всегда повышает рейтинг до b_i (так как $x \le a_i < b_i$). Слабый раунд может быть пройден как нейтральный (если $b_i \le x$) или как повышающий (если $x < b_i$), но в любом случае после него рейтинг не превысит b_i.

Оптимально проходить сильные раунды в порядке возрастания b_i, чтобы меньшие a_i не оказались недоступными из-за слишком высокого рейтинга. Слабые раунды можно «вставлять» между сильными. Отсортируем все раунды по возрастанию b_i (при равных b порядок произволен). Пусть в этом порядке раунды имеют номера $1, 2, \ldots, n$.

Динамическое программирование

Обозначим через $dp[i]$ максимальное число раундов, которое можно пройти, используя только раунды с номерами $1, \ldots, i$, причём i-й раунд обязательно пройден (и является последним). Тогда ответ равен $\max_{1 \le i \le n} dp[i]$.

Если последний раунд i, то рейтинг после всей последовательности равен b_i (так как b_i максимально среди всех пройденных). Пусть предыдущий раунд был $j < i$. После раунда j рейтинг равен b_j. Чтобы войти в раунд i, необходимо $b_j \le a_i$. Между j и i можно пройти некоторое количество слабых раундов k ($j < k < i$). Какие именно слабые раунды можно добавить?

Рассмотрим слабый раунд k ($j < k < i$). Его можно пройти, только если на момент входа текущий рейтинг $x \le a_k$. После раунда j рейтинг равен b_j. Если мы проходим несколько слабых раундов подряд, рейтинг увеличивается до их b (в порядке возрастания). Чтобы все эти раунды были доступны и не помешали входу в i-й, необходимо, чтобы максимальный рейтинг среди них не превышал a_i. Следовательно, можно безопасно пройти все слабые раунды k, для которых $b_k \le a_i$. При этом условие $b_j \le a_k$ для первого из них выполняется автоматически, так как $b_j \le a_i \le a_k$ (последнее неравенство верно, поскольку k слабый $\implies a_k \ge b_k \ge b_j$).

Таким образом, при фиксированном j в промежуток (j, i) можно включить все слабые раунды k с $b_k \le a_i$. Они не помешают друг другу, и рейтинг после них останется $\le a_i$, что позволит пройти i-й раунд.

По мере увеличения i и, соответственно, b_i, некоторые слабые раунды перестают быть доступными для будущих шагов. Слабый раунд k требует для входа $x \le a_k$. Когда текущий рейтинг становится строго больше a_k, такой раунд уже никогда не сможет быть пройден (его «окно возможностей» закрывается). В нашем процессе рейтинг повышается до значений b_i. Значит, как только встречается раунд i с $b_i > a_k$, слабый раунд k «сгорает» — его уже нельзя будет использовать в дальнейшем, и если мы хотим его пройти, это нужно было сделать раньше.

Поэтому при обработке очередного раунда i мы фиксируем, что все слабые раунды k с $a_k < b_i$ теперь недоступны. В динамике их вклад нужно добавить к тем состояниям $j < i$, которые предшествуют текущему i (то есть они могли быть пройдены до i).

Структура данных для быстрого пересчёта

Формально для каждого i:

$$dp[i] = \max_{\substack{0 \le j < i \\ b_j \le a_i}} \Big(dp[j] + cnt(j, i) \Big) + 1,$$

где $cnt(j, i)$ — количество слабых раундов k между j и i, которые «сгорели» к моменту i (т.е. $a_k < b_i$) и, следовательно, могли быть пройдены в промежутке.

Чтобы вычислять это быстро, будем поддерживать множество ещё не «сгоревших» слабых раундов, упорядоченных по a_k, общее количество уже сгоревших слабых раундов на текущий момент и для каждого индекса j величину

$$S[j] = dp[j] - (\text{число сгоревших слабых раундов с индексом } \le j).$$

Изначально $A = 0$, $S[0] = dp[0] = 0$, множество несгоревших раундов содержит все слабые раунды. При обработке раунда i (с параметрами b_i, a_i):

1. **Закрываем окна для слабых раундов.** Из множества несгоревших извлекаем все слабые раунды k, для которых $a_k < b_i$. Каждый такой раунд объявляется сгоревшим: для всех $j \ge k$ значение $S[j]$ уменьшается на 1 (так как число сгоревших раундов $\le j$ увеличилось).

2. **Вычисляем** $dp[i]$**.** Среди индексов $1 \dots i-1$ находим те, для которых $b_j \le a_i$. Пусть pos — наибольший такой индекс. Для любого допустимого $j \le pos$ выполнено

$$dp[j] + cnt(j, i) = dp[j] + \big(A - (\text{сгоревшие } \le j) \big) = S[j] + A.$$

   Следовательно,

$$dp[i] = \max_{j \le pos} \big(S[j] \big) + A + 1.$$

   После вычисления $dp[i]$ мы должны занести в массив S новое значение $S[i] = dp[i] - (\text{сгоревшие } \le i)$. Если раунд i слабый ($a_i \ge b_i$), он добавляется в множество несгоревших (его «окно» пока открыто).

Для эффективной реализации операций «уменьшить суффикс $S[j]$ на 1» и «найти максимум $S[j]$ на префиксе» подходит дерево отрезков, хранящее разности соседних элементов S и позволяющее выполнять оба действия за логарифмическое время от n. Множество несгоревших раундов удобно хранить как очередь с приоритетом (по a_k).

Замечание об одинаковых b

При равных b_i порядок обработки не важен. Если подряд идут несколько раундов с одинаковым b, то условие сгорания слабых $a_k < b_i$ для первого из них не активирует слабые с таким же b (они ещё не обработаны). Ожидающие в очереди слабые раунды будут активированы, как только b_i станет строго больше их a_k.

Реализация

```cpp
const int mx = 1 << 20;
int STsum[4 * mx], STmax[4 * mx];

void STadd(int p, int x) {
    p += mx; STsum[p] += x; STmax[p] += x;
    while (p != 1) {
        p /= 2;
        STsum[p] = STsum[2 * p] + STsum[2 * p + 1];
        STmax[p] = max(STmax[2 * p], STsum[2 * p] + STmax[2 * p + 1]);
    }
}
int STget(int p) {
    p += mx; int ans = 0;
    while (p > 0) {
        if (p % 2 == 1) {
            --p; ans = max(ans + STsum[p], STmax[p]);
        }
        p /= 2;
    } return ans;
}
int F(vector<pair<long long, long long>>& a, long long x, int p) {
    int l = -1, r = p;
    while (l + 1 != r) {
        int m = (l + r) / 2;
        if (a[m].first <= x) l = m;
        else r = m;
    } return l + 1;
}

void Solve() {
    int n; cin >> n;
    vector<pair<long long, long long>> ab(n);
    for (int i = 0; i < n; i++) {
        cin >> ab[i].second >> ab[i].first;
    } sort(ab.begin(), ab.end());
    for (int i = 0; i < 4 * mx; i++) {
        STsum[i] = 0; STmax[i] = 0;
    }
    vector<long long> dp(n + 1);
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>> Q;
    long long cur = 0;
    for (int i = 1; i <= n; i++) {
        while (!Q.empty() && Q.top().first < ab[i - 1].first) {
            STadd(Q.top().second, 1);
            ++cur;
            Q.pop();
        }
        dp[i] = STget(F(ab, ab[i - 1].second, i - 1) + 1) + 1 - cur;
        if (ab[i - 1].second >= ab[i - 1].first) {
            --cur;
            Q.push({ ab[i - 1].second, i });
        }
        STadd(i, dp[i] - dp[i - 1] - (ab[i - 1].second >= ab[i - 1].first));
    }
    long long ans = 0;
    for (int i = 1; i <= n; i++) ans = max(ans, dp[i]);
    cout << ans << '\n';
}
```

Задача «Посадка томатов»

Постановка задачи

Профессор начинает в точке $(0, 0)$ и может перемещаться по плоскости, делая шаги длины $\sqrt{n}$. Каждый шаг обязан заканчиваться в точке с целыми координатами, то есть вектор шага (dx, dy) — целочисленный и удовлетворяет $dx^2 + dy^2 = n$. Такие шаги можно повторять произвольное число раз.

Сад представляет собой прямоугольник со сторонами, параллельными осям координат:

$$[x,\, x + w] \times [y,\, y + h],$$

где x, y, w, h — целые числа, $w, h \geq 1$. Точки внутри сада (включая границу) имеют целые координаты. Требуется определить, сколько целочисленных точек сада достижимы из $(0, 0)$ описанными шагами.

Решение

Прежде всего, чтобы вообще существовал вектор (dx, dy) с длиной $\sqrt{n}$, необходимо, чтобы число n представлялось в виде суммы двух квадратов целых чисел. Критерий представления числа суммой двух квадратов (теорема Ферма–Эйлера):

$$n = 2^a \prod_{p_i \equiv 1 \pmod 4} p_i^{e_i} \prod_{q_j \equiv 3 \pmod 4} q_j^{2f_j},$$

то есть все простые вида $4k + 3$ должны входить в разложение с чётными показателями. Степень двойки a может быть любой. В условии задачи дополнительно оговорено, что a нечётно. Это гарантирует, что после возможного сокращения на общие множители останется двойка в нечётной степени, что наложит определённую структуру на множество достижимых точек.

Если n не удовлетворяет условию (то есть найдётся простое $q \equiv 3 \pmod 4$ с нечётным показателем), то не существует ни одного целочисленного вектора длины $\sqrt{n}$, а значит, никакая точка, кроме начала координат, не достижима. В этом случае ответ равен 1, если $(0, 0)$ лежит внутри прямоугольника, и 0 иначе.

Пусть n представимо суммой двух квадратов и удовлетворяет условию нечётности степени двойки. Тогда все достижимые точки образуют двумерную подрешётку, которую можно описать явно. Рассмотрим каноническое разложение n:

$$n = 2^a \prod_{p_i \equiv 1 \pmod 4} p_i^{e_i} \prod_{q_j \equiv 3 \pmod 4} q_j^{2f_j},$$

где a нечётно. Положим

$$C = 2^{(a-1)/2} \prod_{q_j \equiv 3 \pmod 4} q_j^{f_j}.$$

Простые $p_i \equiv 1 \pmod 4$ не влияют на C. Тогда множество достижимых из $(0, 0)$ точек — это в точности все целые точки (X, Y), для которых

$$X \equiv Y \pmod{2C} \quad \text{и} \quad X, Y \text{ кратны } C.$$

Иными словами, координаты (X, Y) представимы в виде $(C \cdot u,\ C \cdot v)$, где $u, v \in \mathbb{Z}$ и $u \equiv v \pmod 2$ (т.е. u и v имеют одинаковую чётность). Эквивалентно, после деления на C координаты становятся целыми и имеют одинаковую чётность.

Обоснование: любой допустимый вектор (dx, dy) длины $\sqrt{n}$ можно записать как $dx = C \cdot du$, $dy = C \cdot dv$, где du, dv — целые числа, $du^2 + dv^2 = n/C^2$. При a нечётном и отсутствии «плохих» простых число n/C^2 есть произведение степеней простых $1 \pmod 4$ и, возможно, одной двойки (если a нечётно). Можно показать, что все достижимые комбинации таких векторов дают в точности описанную решётку. Наличие простых $4k+3$ в чётных степенях добавляет множитель q^f в C, сжимая решётку в q раз по каждой оси, но не меняя условие на чётность.

Подсчёт точек в прямоугольнике

Итак, нам нужно посчитать число целых точек (X, Y) в прямоугольнике $[x, x+w] \times [y, y+h]$, для которых X и Y кратны C и $X/C \equiv Y/C \pmod 2$.

Перейдём к новым координатам: $u = X/C$, $v = Y/C$. Точка (X, Y) принадлежит прямоугольнику тогда и только тогда, когда

$$x \le Cu \le x + w, \qquad y \le Cv \le y + h.$$

Из этих неравенств выражаем целочисленные границы для u и v:

$$U_{\min} = \left\lceil \frac{x}{C} \right\rceil, \qquad U_{\max} = \left\lfloor \frac{x+w}{C} \right\rfloor,$$
$$V_{\min} = \left\lceil \frac{y}{C} \right\rceil, \qquad V_{\max} = \left\lfloor \frac{y+h}{C} \right\rfloor.$$

Если $U_{\min} > U_{\max}$ или $V_{\min} > V_{\max}$, прямоугольник не содержит ни одной точки решётки, ответ 0. В противном случае все пары (u, v) с $u \in [U_{\min}, U_{\max}]$, $v \in [V_{\min}, V_{\max}]$ дают целые точки (Cu, Cv) внутри сада. Осталось отобрать те из них, для которых u и v имеют одинаковую чётность. Пусть:

- $L_u = U_{\max} - U_{\min} + 1$ — количество целых u в отрезке.
- $L_v = V_{\max} - V_{\min} + 1$ — количество целых v в отрезке.
- E_u — количество чётных чисел среди этих u.
- E_v — количество чётных чисел среди этих v.

Числа E_u, E_v легко вычисляются по формулам: если $U_{\min}$ чётно, то чётных будет $\lceil L_u/2 \rceil$, иначе $\lfloor L_u/2 \rfloor$ (аналогично для v). Тогда количество пар с одинаковой чётностью равно

$$\text{Ans} = E_u \cdot E_v \; + \; (L_u - E_u) \cdot (L_v - E_v).$$

Первое слагаемое — пары (чётный u, чётный v), второе — пары (нечётный u, нечётный v). Это и есть искомое число достижимых точек в саду.

Реализация

```
void Solve() {
    long long n, x, y, w, h;
    cin >> n >> x >> y >> w >> h;

    int a = 0;
    while (n % 2 == 0) {
        n /= 2;
        a++;
    }

    long long C = 1;
    bool ok = true;
    if (a % 2 == 0) {
        ok = false;
    }
```

```
    else {
        int k = (a - 1) / 2;
        C <<= k;
    }

    for (long long p = 3; p * p <= n; p += 2) {
        if (n % p == 0) {
            int e = 0;
            while (n % p == 0) {
                n /= p; e++;
            }
            if (p % 4 == 3) {
                if (e % 2 != 0) {
                    ok = false; break;
                }
                else {
                    for (int i = 0; i < e / 2; i++) {
                        C *= p;
                    }
                }
            }
        }
    }

    if (ok && n > 1 && n % 4 == 3) {
        ok = false;
    }

    if (!ok) {
        if (0 >= x && 0 <= x + w && 0 >= y && 0 <= y + h) { cout << 1; }
        else { cout << 0; }
        return;
    }

    long long Lx = x, Rx = x + w;
    long long Ly = y, Ry = y + h;
    long long U1, U2, V1, V2;

    if (Lx >= 0) { U1 = (Lx + C - 1) / C; }
    else { U1 = Lx / C; }
    if (Rx >= 0) { U2 = Rx / C; }
    else { U2 = (Rx - C + 1) / C; }
    if (Ly >= 0) { V1 = (Ly + C - 1) / C; }
    else { V1 = Ly / C; }
    if (Ry >= 0) { V2 = Ry / C; }
    else { V2 = (Ry - C + 1) / C; }

    if (U1 > U2 || V1 > V2) {
        cout << 0;
        return;
    }

    long long eu;
    if (U1 % 2 == 0) { eu = (U2 - U1 + 1 + 1) / 2; }
    else { eu = (U2 - U1 + 1) / 2; }
    long long ev;
    if (V1 % 2 == 0) { ev = (V2 - V1 + 1 + 1) / 2; }
    else { ev = (V2 - V1 + 1) / 2; }

    cout << eu * ev + (U2 - U1 + 1 - eu) * (V2 - V1 + 1 - ev);
}
```

Задача «Софиты»

Постановка задачи

На плоскости имеется n софитов, центр i-го софита находится в точке $C_i = (x_i, y_i)$. Он освещает круглую область с границей, задаваемую неравенством

$$(X - x_i)^2 + (Y - y_i)^2 \;\le\; x_i^2 + y_i^2. \tag{1}$$

Актёр в начальный момент находится в начале координат $O = (0, 0)$ и должен всё время оставаться в освещённой области, принадлежащей всем софитам одновременно. Требуется найти максимальное расстояние от O, на которое он может удалиться, то есть

$$\max\Big\{\, |OT| \;\Big|\; T \in \bigcap_{i=1}^{n} \text{круг}_i \,\Big\},$$

где $|OT| = \sqrt{X^2 + Y^2}$. Если пересечение кругов не содержит точек, кроме O (или содержит только O), ответ равен 0.

Геометрическое преобразование неравенства

Раскроем скобки в неравенстве (1):

$$X^2 + Y^2 - 2x_iX - 2y_iY + x_i^2 + y_i^2 \le x_i^2 + y_i^2,$$

откуда

$$X^2 + Y^2 \;\le\; 2\,(x_iX + y_iY). \tag{2}$$

Заметим, что левая часть — квадрат расстояния от начала координат, а правая — удвоенное скалярное произведение радиус-вектора точки T и центра софита. Это неравенство можно «вывернуть» с помощью подходящей замены переменных. Для любой точки $T \neq O$ определим новую точку

$$U = \frac{T}{|T|^2},$$

то есть точка U лежит на том же луче из начала координат, что и T, но её расстояние до начала равно $1/|T|$. Тогда

$$|T| = \frac{1}{|U|}, \qquad T = \frac{U}{|U|^2}.$$

Подставим T в неравенство (2):

$$\frac{1}{|U|^2} \;\le\; 2\Big(x_i\frac{u_x}{|U|^2} + y_i\frac{u_y}{|U|^2}\Big).$$

Домножая обе части на положительное $|U|^2$, получаем простейшее линейное неравенство

$$1 \;\le\; 2\,(x_iu_x + y_iu_y) \quad\Longleftrightarrow\quad x_iu_x + y_iu_y \ge \frac{1}{2}. \tag{3}$$

Таким образом, в координатах U каждый круг превратился в полуплоскость, граница которой — прямая, задаваемая равенством $x_i u_x + y_i u_y = 1/2$. Расстояние от начала координат до этой прямой равно

$$d_i = \frac{1}{2\sqrt{x_i^2 + y_i^2}},$$

а нормальный вектор прямой — это единичный вектор в направлении центра софита: $\vec{n}_i = \dfrac{(x_i, y_i)}{\sqrt{x_i^2 + y_i^2}}$.

Искомая задача максимизации $|T|$ эквивалентна минимизации $|U|$ при ограничениях (3). Действительно, чем дальше точка T от начала координат, тем ближе к началу координат её образ U. Следовательно,

$$\max |T| = \frac{1}{\min\limits_{U \in \mathcal{P}} |U|},$$

где $\mathcal{P}$ — пересечение полуплоскостей (3). Заметим, что оно не содержит начало координат, так как оно не удовлетворяет неравенству (3).

Пересечение полуплоскостей

Множество допустимых точек $\mathcal{P}$ описывается системой линейных неравенств

$$x_i u_x + y_i u_y \geq \frac{1}{2}, \qquad i = 1, \ldots, n.$$

Каждое неравенство задаёт полуплоскость, граница которой — прямая

$$\ell_i : \; x_i u_x + y_i u_y = \frac{1}{2}.$$

Для каждой прямой определим единичный вектор нормали, направленный внутрь допустимой области:

$$\vec{n}_i = \frac{(x_i, y_i)}{\sqrt{x_i^2 + y_i^2}}.$$

Тогда прямая ℓ_i состоит из точек U, для которых $\vec{n}_i \cdot U = d_i$, где

$$d_i = \frac{1}{2\sqrt{x_i^2 + y_i^2}}.$$

Точка $P_i = d_i \vec{n}_i$ лежит на прямой, а направляющий вектор прямой $\vec{v}_i = (n_{i,y}, -n_{i,x})$ ориентирован так, что допустимая полуплоскость лежит слева от прямой при движении вдоль $\vec{v}_i$. Таким образом, i-я полуплоскость задаётся парой $(P_i, \vec{v}_i)$.

Чтобы гарантировать ограниченность пересечения, необходимо добавить четыре дополнительные полуплоскости, образующие большой квадрат со сторонами, параллельными осям координат (например, $x = \pm M$, $y = \pm M$ с $M = 10^6$). Эти полуплоскости не меняют искомую область $\mathcal{P}$, так как она заведомо лежит внутри такого квадрата.

После этого применим стандартный алгоритм пересечения полуплоскостей, который возвращает выпуклый многоугольник $\Pi = P_1 P_2 \ldots P_k$ — искомую область $\mathcal{P}$. Если число вершин $k < 3$, то пересечение пусто или вырождено, что означает отсутствие общих точек у исходных кругов вне начала координат; в этом случае ответ равен 0.

Поиск ближайшей к началу координат точки многоугольника

Итак, мы получили выпуклый многоугольник $\Pi = P_1 P_2 \ldots P_k$, который представляет собой область $\mathcal{P}$ в преобразованных координатах. Начало координат O лежит вне этого многоугольника, так как не удовлетворяет ни одному из неравенств (3). Наша цель — найти точку многоугольника, ближайшую к O, то есть

$$R_{\min} = \min_{U \in \Pi} |OU|.$$

Поскольку многоугольник выпуклый, а расстояние от точки до выпуклого множества — выпуклая функция, минимум достигается либо в вершине многоугольника, либо на одном из его рёбер. Поэтому достаточно перебрать все рёбра P_iP_{i+1} и для каждого найти ближайшую к O точку этого отрезка.

Рассмотрим отрезок AB (где $A = P_i$, $B = P_{i+1}$). Векторы:

$$\vec{AB} = B - A, \qquad \vec{AO} = O - A.$$

Вычислим скалярное произведение $t = \vec{AO} \cdot \vec{AB}$. Возможны три случая.

1. $t \le 0$**.** Проекция точки O на прямую, содержащую отрезок, лежит левее точки A (или совпадает с ней). Следовательно, ближайшая точка отрезка — его левый конец A. Расстояние равно $|AO|$.

2. $t \ge \vec{AB} \cdot \vec{AB}$**.** Проекция точки O лежит правее точки B. Ближайшая точка отрезка — B, расстояние равно $|BO|$.

3. $0 < t < \vec{AB} \cdot \vec{AB}$**.** Проекция попадает строго внутрь отрезка. Ближайшая точка — основание перпендикуляра, опущенного из O на прямую AB. Расстояние равно длине этого перпендикуляра:

$$\frac{|\vec{AO} \times \vec{AB}|}{|\vec{AB}|},$$

где $\vec{AO} \times \vec{AB} = AO_x \cdot AB_y - AO_y \cdot AB_x$ — модуль векторного произведения.

Пройдя по всем рёбрам многоугольника и вычислив минимальное из найденных расстояний, получим $R_{\min}$. После нахождения $R_{\min}$ ответ исходной задачи вычисляется как

$$\text{Ans} = \frac{1}{R_{\min}}.$$

Действительно, точка U, ближайшая к началу координат, соответствует точке $T = U/|U|^2$ в исходной плоскости, которая является наиболее удалённой от начала координат точкой пересечения кругов. Её расстояние до начала координат равно $1/|U| = 1/R_{\min}$.

Реализация

```cpp
//https://github.com/rewatlok/Prepare-for-ICPC/blob/main/Theory/Math/namespace%20GEOMETRY.cpp

using namespace GEOMETRY;

void Solve() {
    int n; cin >> n;
    vector<pair<r<double>, r<double>>> ls;

    for (int i = 0; i < n; i++) {
        double x, y; cin >> x >> y;
        double l = sqrtl(x * x + y * y);

        double nx = x / l;
        double ny = y / l;

        ls.push_back({ r<double>(0.5 / l * nx, 0.5 / l * ny),
                       r<double>(ny, -nx) });
    }

    ls.push_back({ r<double>(-1e6, 0), r<double>(0, -1) });
    ls.push_back({ r<double>(1e6, 0), r<double>(0, 1) });
    ls.push_back({ r<double>(0, -1e6), r<double>(1, 0) });
    ls.push_back({ r<double>(0, 1e6), r<double>(-1, 0) });
```

```
    vector<r<double>> pl = halfplane_intersection(ls);

    if (pl.size() < 3) {
        cout << fixed << setprecision(6) << 0.0 << '\n';
    }
    else {
        double mn = 1e18;

        for (int i = 0; i < (int)pl.size(); i++) {
            r<double> o(0, 0);
            r<double> a = pl[i];
            r<double> b = pl[(i + 1) % pl.size()];

            r<double> ab = b - a;
            r<double> ao = o - a;

            if (ao * ab <= 0) {
                mn = min(mn, (double)ao.sqrt_len());
            }
            else if (ao * ab >= ab * ab) {
                mn = min(mn, (double)(b - o).sqrt_len());
            }
            else {
                mn = min(mn, fabs(ao ^ ab) / (double)ab.sqrt_len());
            }
        }

        cout << fixed << setprecision(6) << 1.0 / mn << '\n';
    }
}
```

Задача «Диапазонные запросы»

Постановка задачи

Имеется массив a длины n, изначально заполненный нулями. Дана последовательность из m запросов прибавления: i-й запрос ($1 \le i \le m$) характеризуется числами l_i, r_i, x_i и означает прибавление значения x_i ко всем элементам a с индексами от l_i до r_i включительно. После того как все m запросов были записаны, поступает q итоговых запросов. Каждый итоговый запрос задаётся парой чисел L, R ($1 \le L \le R \le m$) и требует:

1. взять только запросы прибавления с номерами от L до R включительно;
2. применить их к исходному нулевому массиву (то есть каждый итоговый запрос обрабатывается независимо, без накопления эффекта);
3. вывести получившийся массив a длины n.

Ограничения: $n, m, q \le 10^5$, но гарантируется, что $n \cdot q \le 10^6$.

Корневая декомпозиция запросов

Разобьём все m запросов прибавления на блоки фиксированного размера B. Количество блоков $K = \lceil m/B \rceil$. Для каждого блока мы можем заранее вычислить его суммарное влияние на массив, если применить все запросы этого блока к изначально нулевому массиву.

Зная влияние каждого блока, ответ на итоговый запрос, охватывающий несколько блоков, можно собрать из трёх частей:

- полностью входящие блоки (их суммарное влияние берётся из предподсчитанных данных);
- неполный левый блок (обрабатывается напрямую);
- неполный правый блок (обрабатывается напрямую).

Занумеруем блоки с нуля: блок b ($0 \le b < K$) содержит запросы с индексами от $b \cdot B + 1$ до $\min((b+1) \cdot B,\ m)$ (в 1-индексации). Для каждого блока b вычислим массив block_effect_b длины n, в котором $\text{block\_effect}_b[j]$ — суммарное приращение, которое получает элемент a_j после применения всех запросов данного блока (к изначально нулевому массиву). Удобный способ вычислить block_effect_b — использование разностного массива: заведём массив $diff$ длины $n+2$, заполненный нулями и для каждого запроса (l, r, x) из блока выполняем $diff[l] \mathrel{+}= x$, $diff[r+1] \mathrel{-}= x$. После обработки всех запросов блока, проходим по массиву с 1 до n, поддерживая текущую сумму cur. Имеем $\text{block\_effect}_b[j] = cur$ (после прибавления $diff[j]$).

Далее строим массив префиксных сумм по блокам $\text{pref}[k][j]$ — значение в позиции j после применения всех запросов из первых k блоков (блоки $0, 1, \ldots, k-1$). $\text{pref}[0][j] = 0$ для всех j. Для $k = 1 \ldots K$:

$$\text{pref}[k][j] = \text{pref}[k-1][j] + \text{block\_effect}_{k-1}[j].$$

Фактически, $\text{pref}[k]$ — это просто накопительный массив, в котором учтены блоки до $k-1$ включительно.

Ответ на итоговый запрос

Пусть поступил запрос с границами L, R (1-индексация номеров запросов прибавления). Перейдём к 0-индексации: $L' = L - 1$, $R' = R - 1$. Определим номера блоков, в которые попадают левая и правая границы:

$$bl = \left\lfloor \frac{L'}{B} \right\rfloor, \qquad br = \left\lfloor \frac{R'}{B} \right\rfloor.$$

Возможны два случая.

Случай 1: $bl = br$ (все запросы внутри одного блока)

Если левая и правая границы принадлежат одному блоку, то никакие полные блоки не фигурируют. Мы применяем все запросы с L' по R' напрямую с помощью разностного массива: инициализируем $diff[1..n{+}1]$ нулями и для каждого i от L' до R' выполняем $diff[l_{i+1}]$ += x_{i+1}, $diff[r_{i+1} + 1]$ −= x_{i+1}. После чего вычисляем префиксные суммы по $diff$, получаем итоговый массив.

Случай 2: $bl < br$ (запросы охватывают несколько блоков)

Тогда запросы из полных блоков между bl и br можно учесть с помощью предподсчитанных префиксных сумм:

- блоки, полностью входящие в диапазон, имеют индексы от $bl + 1$ до $br - 1$. Их суммарный вклад равен разности массивов:
$$ans[j] = \text{pref}[br][j] - \text{pref}[bl + 1][j],$$
так как pref$[br]$ содержит вклад блоков $0 \dots br - 1$, а pref$[bl + 1]$ — блоков $0 \dots bl$. Вычитание даёт вклад блоков $bl + 1 \dots br - 1$.

Остаётся добавить вклад от неполных блоков — левого (часть блока bl от L' до конца блока) и правого (часть блока br от начала блока до R'). Это снова делается через разностный массив $diff$:

- для левого неполного блока: перебираем i от L' до $\min((bl + 1) \cdot B - 1,\ m - 1)$ и применяем запросы в $diff$;
- для правого неполного блока: перебираем i от $br \cdot B$ до R' и применяем запросы в $diff$.

После этого проходим по массиву, считаем текущую сумму по $diff$ и прибавляем её к $ans[j]$. В итоге получаем искомый массив.

Выбор параметра B

Общее время работы алгоритма складывается из:

- предподсчёта: $O\big(m + \frac{m}{B} \cdot n\big)$;
- ответов на q запросов: $O\big(q \cdot (n + B)\big)$.

Однако благодаря гарантии $n \cdot q \le 10^6$, суммарное время, потраченное на формирование и вывод массивов во всех запросах (слагаемое $q \cdot n$), не превышает 10^6 операций, что пренебрежимо мало. Поэтому доминирующими становятся:

- предподсчёт: $O(m + \frac{m}{B} \cdot n)$;
- обработка краевых запросов: $O(q \cdot B)$;

Чтобы сбалансировать $\frac{m}{B} \cdot n$ и $q \cdot B$, выберем $B \approx \sqrt{m}$. Тогда итоговая асимптотика:

$$O\big(m + \sqrt{m} \cdot n + q \cdot \sqrt{m}\big) = O\big(m + \sqrt{m} \cdot (n + q)\big).$$

Реализация

```
void Solve() {
    int n, m, q; cin >> n >> m >> q;
    vector<int> l(m + 1), r(m + 1), x(m + 1);
    for (int i = 1; i <= m; i++) {
        cin >> l[i] >> r[i] >> x[i];
    }

    int B = sqrt(m);
    int blocks = (m + B - 1) / B;

    vector<vector<int>> pref(blocks + 1, vector<int>(n));

    for (int b = 0; b < blocks; b++) {
        vector<int> diff(n + 2);
        for (int i = b * B; i <= min((b + 1) * B, m) - 1; i++) {
            int idx = i + 1;
            diff[l[idx]] += x[idx];
            if (r[idx] + 1 <= n) {
                diff[r[idx] + 1] -= x[idx];
            }
        }

        int cur = 0;
        for (int j = 1; j <= n; j++) {
            cur += diff[j];
            pref[b + 1][j - 1] = pref[b][j - 1] + cur;
        }
    }

    for (int t = 0; t < q; t++) {
        int L, R; cin >> L >> R;
        L--; R--;

        int bl = L / B;
        int br = R / B;

        vector<int> ans(n);

        if (bl == br) {
            vector<int> diff(n + 2);
            for (int i = L; i <= R; i++) {
                int idx = i + 1;
                diff[l[idx]] += x[idx];
                if (r[idx] + 1 <= n) {
                    diff[r[idx] + 1] -= x[idx];
                }
            }
            int cur = 0;
            for (int j = 1; j <= n; j++) {
                cur += diff[j];
                ans[j - 1] = cur;
            }
        }
        else {
            if (bl + 1 <= br - 1) {
                for (int j = 0; j < n; j++) {
                    ans[j] = pref[br][j] - pref[bl + 1][j];
                }
            }
```

```
60
61              vector<int> diff(n + 2);
62
63              for (int i = L; i <= min(m - 1, (bl + 1) * B - 1); i++) {
64                  int idx = i + 1;
65                  diff[l[idx]] += x[idx];
66                  if (r[idx] + 1 <= n) {
67                      diff[r[idx] + 1] -= x[idx];
68                  }
69              }
70
71              for (int i = br * B; i <= R; i++) {
72                  int idx = i + 1;
73                  diff[l[idx]] += x[idx];
74                  if (r[idx] + 1 <= n) {
75                      diff[r[idx] + 1] -= x[idx];
76                  }
77              }
78
79              int cur = 0;
80              for (int j = 1; j <= n; j++) {
81                  cur += diff[j];
82                  ans[j - 1] += cur;
83              }
84          }
85
86          for (int j = 0; j < n; j++) {
87              cout << ans[j] << ' ';
88          }
89          cout << '\n';
90      }
91  }
```

Задача «Заморозка»

Постановка задачи

На соревновании участвуют n команд и предложено p задач. До заморозки результатов известна полная таблица попыток: для каждой команды и каждой задачи отмечен один из четырёх статусов:

- `+` — задача сдана до заморозки,
- `-` — были неудачные попытки до заморозки, задача не сдана,
- `?` — попытки были во время заморозки, исход неизвестен,
- `.` — попыток не было.

После заморозки каждая задача, помеченная `?`, может стать либо решённой (`+`), либо нерешённой (`-`). Дополнительно известна (возможно, частичная) информация:

- Число K — количество открытых задач (задач, по которым есть хотя бы одно успешное решение). Это число может быть неизвестно.
- Для некоторых команд известно точное число решённых ими задач.

Нужно найти количество итоговых таблиц (после разморозки) по модулю 998 244 353, совместимых со всей предоставленной информацией. Если информация противоречива — ответ 0.

Формализация в виде подмножеств

Занумеруем задачи от 1 до p. Итоговый результат команды можно представить в виде подмножества (маски) задач, которые она решила. Для каждой команды i по её строке статусов можно определить множество допустимых масок. Маска M (биты $0, \dots, p-1$) допустима для команды i, если выполнены три условия:

1. Если задача j решена (бит j установлен), то символ в строке команды для этой задачи обязан быть `+` или `?`.
2. Если задача j не решена (бит j сброшен), то символ обязан быть `-`, `.` или `?`.
3. Если для команды известно точное число решённых задач c_i, то $\mathrm{popcount}(M) = c_i$.

Множество всех допустимых масок для команды i обозначим через $\mathcal{M}_i$. Пусть $O \subseteq \{1, \dots, p\}$ — множество открытых задач. Тогда для каждой команды i её итоговая маска решённых задач должна быть подмножеством O (она не может решить задачу, которая не открыта).

Решение

Зафиксируем множество открытых задач O. Для одной команды i количество допустимых масок, являющихся подмножеством O, равно

$$C_i[O] = \sum_{\substack{M \in \mathcal{M}_i \\ M \subseteq O}} 1.$$

Поскольку команды независимы, количество способов выбрать для всех команд допустимые маски, лежащие в O, есть

$$F[O] = \prod_{i=1}^{n} C_i[O] \pmod{998\,244\,353}.$$

Величина $F[O]$ подсчитывает все варианты, в которых множество открытых задач содержится в O. Чтобы получить количество вариантов, при которых множество открытых задач в точности равно O, обозначим его через $dp[O]$. По определению, для любого O выполняется

$$F[O] = \sum_{S \subseteq O} dp[S].$$

Это классическая свёртка по подмножествам. Зная все $F[O]$, можно восстановить $dp[O]$ с помощью обратного преобразования SOS (sum over subsets):

$$dp[O] = \sum_{S \subseteq O} (-1)^{|O|-|S|} F[S],$$

что эффективно реализуется динамикой по битам за $O(p \cdot 2^p)$. После вычисления $dp[O]$ ответ на задачу получается суммированием $dp[O]$ по всем O, для которых количество открытых задач совпадает с заданным K (если K неизвестно — суммирование по всем O).

Для каждой команды i создадим массив can_i размера 2^p, где $can_i[M] = 1$, если маска M совместима со строкой команды и (если задано) количеством решённых задач, и 0 иначе. Совместимость проверяется побитово: для каждого j, если бит j в M равен 1, то символ в позиции j должен быть `+` или `?`; если бит j равен 0, то символ должен быть `-`, `.` или `?`. При наличии известного числа решённых задач c_i дополнительно требуем, чтобы количество единичных битов в M равнялось c_i.

Далее для каждой команды нужно вычислить $C_i[M]$ — сумму $can_i[S]$ по всем подмножествам $S \subseteq M$. Для каждого бита $j = 0 \dots p-1$ перебираем все маски M от 0 до $2^p - 1$; если M содержит бит j (т.е. $(M \,\&\, (1 \ll j)) \neq 0$), то прибавляем к $can_i[M]$ значение $can_i[M \oplus (1 \ll j)]$. После обработки всех битов $can_i[M]$ становится равным $\sum_{S \subseteq M} can_i[S]$, то есть искомой величиной $C_i[M]$.

Теперь объединяем команды. Заводим массив F размера 2^p, изначально заполненный единицами. Для каждой команды (после её SOS-преобразования) поэлементно домножаем $F[M]$ на $C_i[M]$ по модулю $998\,244\,353$. После обработки всех команд $F[M]$ будет равно $\prod_i C_i[M]$ — числу способов, при котором множество открытых задач является подмножеством M.

Осталось перейти от величин $F[M]$, которые считают все варианты с множеством открытых задач, содержащимся в M, к величинам $dp[M]$, которые считают варианты с множеством открытых задач, в точности равным M. Связь между ними даётся формулой включения-исключения:

$$dp[M] = \sum_{S \subseteq M} (-1)^{|M|-|S|} F[S].$$

Копируем F в массив dp. Для каждого бита $j = 0 \dots p-1$ перебираем маски в убывающем порядке (от $2^p - 1$ до 0); если M содержит бит j, выполняем $dp[M] = (dp[M] - dp[M \oplus (1 \ll j)] + MOD) \bmod MOD$. После этой процедуры $dp[M]$ станет числом способов, при которых множество открытых задач в точности равно M.

Реализация

```
void Solve() {
    int mod = 998244353;

    int n, p; string s;

    cin >> n >> p;
    cin >> s;
    cin >> s;

    int opens;
    if (s == "unknown") { opens = 400; }
    else { opens = stoll(s); }

    vector<int> known(n);
    vector<string> P(n);

    for (int i = 0; i < n; i++) {
        cin >> s;
        cin >> s;
        if (s == "unknown") {
            known[i] = 400;
        }
        else {
            known[i] = stoll(s);
        }
        cin >> P[i];
    }

    int t_size = 1 << p;
    vector<int> F(t_size, 1);

    for (int k = 0; k < n; k++) {
        vector<int> can(t_size);

        for (int mask = 0; mask < t_size; mask++) {
            if (known[k] != 400) {
                if (__builtin_popcount(mask) != known[k]) {
                    continue;
                }
            }
            bool f = true;
            for (int j = 0; j < p; j++) {
                if (mask & (1 << j)) {
                    if (P[k][j] != '+' && P[k][j] != '?') {
                        f = false;
                        break;
                    }
                }
                else {
                    if (P[k][j] != '-' && P[k][j] != '.' && P[k][j] != '?') {
                        f = false;
                        break;
                    }
                }
            }
            if (f) {
                can[mask] = 1;
            }
        }
```

```
60          vector<int> C = can;
61          for (int i = 0; i < p; i++) {
62              for (int mask = 0; mask < t_size; mask++) {
63                  if (mask & (1 << i)) {
64                      C[mask] = C[mask] + C[mask ^ (1 << i)];
65                      if (C[mask] >= mod) {
66                          C[mask] -= mod;
67                      }
68                  }
69              }
70          }
71
72          for (int mask = 0; mask < t_size; mask++) {
73              F[mask] = ((long long)F[mask] * C[mask]) % mod;
74          }
75      }
76
77      vector<int> dp_ = F;
78      for (int i = 0; i < p; i++) {
79          for (int mask = t_size - 1; mask >= 0; mask--) {
80              if (mask & (1 << i)) {
81                  dp_[mask] = dp_[mask] - dp_[mask ^ (1 << i)];
82                  if (dp_[mask] < 0) {
83                      dp_[mask] += mod;
84                  }
85              }
86          }
87      }
88
89      int ans = 0;
90      for (int mask = 0; mask < t_size; mask++) {
91          if (opens == 400 || __builtin_popcount(mask) == opens) {
92              ans += dp_[mask];
93              if (ans >= mod) {
94                  ans -= mod;
95              }
96          }
97      }
98
99      cout << ans;
100 }
```

Задача «Министерство налогов»

Постановка задачи

Есть n компаний с доходами $a_i \geq 0$. Министр может подписать некоторые из k указов, j-й указ облагает налогом компании с номерами от l_j до r_j включительно. Если компания i попадает ровно в один подписанный указ, она платит a_i бурлей. Если она не попадает ни в один указ или попадает в два и более, она не платит ничего. Требуется выбрать множество указов так, чтобы максимизировать суммарный налог.

Решение

Каждый указ можно рассматривать как отрезок $[l, r]$, а выбор множества указов — как подмножество этих отрезков. Для каждого элемента i его вклад равен a_i, если число покрывающих его выбранных отрезков равно 1, и 0 иначе.

Отсортируем отрезки по правому концу. Пусть $\texttt{pref}[i] = a_1 + \cdots + a_i$ — префиксная сумма доходов компаний (считаем $\texttt{pref}[0] = 0$). Определим $dp[i]$ как максимальный налог, который можно собрать с компаний $1 \dots i$ (префикс длины i). Инициализируем $dp[0] = 0$. Для каждого $r = 1 \dots n$:

$$dp[r] = dp[r-1]$$

(не берём никакой новый отрезок с концом r). Затем перебираем все указы, оканчивающиеся в r. Для каждого такого указа $[l, r]$ возможны два принципиально разных способа его использования.

1. Отрезок не перекрывается с уже выбранными

Если выбранные ранее отрезки не заходят правее $l-1$, то добавление отрезка $[l, r]$ даёт прирост $\sum_{i=l}^{r} a_i$. В этом случае

$$dp[r] = \max\big(dp[r],\ dp[l-1] + \texttt{pref}[r] - \texttt{pref}[l-1]\big).$$

2. Отрезок перекрывает некоторый предыдущий отрезок

Предположим, что в оптимальном решении на префиксе j ($l \leq j < r$) последний выбранный отрезок заканчивается в j и целиком покрывает интервал $[l, j]$ (это может быть результат цепочки перекрытий). Тогда если мы добавим новый отрезок $[l, r]$, он пересечётся с тем отрезком на $[l, j]$, и вклад перекрытия пропадёт. Итоговая сумма для префикса r будет равна $dp[j] + \sum_{i=l}^{r} a_i \ - \ 2\sum_{i=l}^{j} a_i$. Выражая через префиксные суммы, получаем:

$$dp[j] + \big(\texttt{pref}[r] - \texttt{pref}[l-1]\big) - 2\big(\texttt{pref}[j] - \texttt{pref}[l-1]\big) = dp[j] - 2\texttt{pref}[j] + \texttt{pref}[r] + \texttt{pref}[l-1].$$

Таким образом, для фиксированного указа $[l, r]$ выгодно выбрать такое $j \in [l, r-1]$, которое максимизирует $dp[j] - 2\texttt{pref}[j]$. Обозначим:

$$M(l, r) = \max_{j \in [l,\, r-1]} \big(dp[j] - 2\texttt{pref}[j]\big).$$

Тогда переход имеет вид:

$$dp[r] = \max\Big(dp[r],\ M(l, r) + \texttt{pref}[r] + \texttt{pref}[l-1]\Big)$$

Структура данных для быстрых запросов

Для эффективного вычисления $M(l, r)$ построим дерево отрезков (или другое RMQ) над массивом значений $val[j] = dp[j] - 2\texttt{pref}[j]$. При обработке очередного r мы сначала вычисляем $dp[r]$ по всем указам, заканчивающимся в r, используя запрос максимума на отрезке $[l, r-1]$. После того как $dp[r]$ найдено, мы обновляем позицию r в дереве, устанавливая $val[r] = dp[r] - 2\texttt{pref}[r]$.

Заметим, что переход без перекрытия также можно интерпретировать как частный случай $j = l - 1$ (если считать $val[l-1] = dp[l-1] - 2\texttt{pref}[l-1]$), но для единообразия его проще обработать отдельно.

Корректность

Почему достаточно рассматривать только перекрытие с одним предыдущим отрезком, а не с несколькими? Если новый отрезок $[l, r]$ перекрывается с несколькими ранее выбранными отрезками, то все они должны лежать внутри $[l, r]$ и не могут перекрываться между собой (иначе их пересечения уже были бы обнулены и не могли бы быть оптимальными). Можно показать индукцией по длине, что оптимальное множество можно представить в виде последовательности отрезков, в которой каждый следующий отрезок начинается не позже начала предыдущего и перекрывает его правый конец. В такой цепочке каждый новый отрезок взаимодействует только с непосредственно предыдущим, и формула с одним предыдущим индексом j полностью описывает переход.

Реализация

```
class SegmentTree {
public:
    SegmentTree(int size) {
        n = size; tree.assign(4 * n, -2e18);
    }
    void change(int p, long long x) {
        U(1, 0, n - 1, p, x);
    }
    long long query(int L, int R) {
        return Q(1, 0, n - 1, L, R);
    }
private:
    int n;
    vector<long long> tree;
    void U(int v, int tl, int tr, int p, long long x) {
        if (tl == tr) {
            tree[v] = x;
            return;
        }
        int m = (tl + tr) / 2;
        if (p <= m)
            U(v * 2, tl, m, p, x);
        else
            U(v * 2 + 1, m + 1, tr, p, x);
        tree[v] = max(tree[v * 2], tree[v * 2 + 1]);
    }
    long long Q(int v, int tl, int tr, int l, int r) {
        if (l > r)
            return -2e18;
        if (l == tl && r == tr)
            return tree[v];
        int m = (tl + tr) / 2;
        return max(Q(v * 2, tl, m, l, min(r, m)),
            Q(v * 2 + 1, m + 1, tr, max(l, m + 1), r));
    }
};
```

```
long long get(vector<long long>& p, int l, int r) {
    if (l == 0)
        return p[r];
    else
        return p[r] - p[l - 1];
}

void Solve() {
    int n, q; cin >> n >> q;
    vector<long long> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    vector<long long> p(n);
    long long s = 0;
    for (int i = 0; i < n; i++) {
        s += a[i];
        p[i] = s;
    }

    map<int, vector<int>> M;
    for (int i = 0; i < q; i++) {
        int l, r; cin >> l >> r;
        l--; r--;
        M[r].push_back(l);
    }
    for (auto& [u, v] : M) {
        sort(v.begin(), v.end());
    }

    vector<long long> dp(n);
    SegmentTree ST(n);

    for (int r = 0; r < n; r++) {
        if (r != 0) {
            dp[r] = dp[r - 1];
        }
        if (!M[r].empty()) {
            for (int l : M[r]) {
                if (l != 0) {
                    dp[r] = max(dp[r], dp[l - 1] + get(p, l, r));
                }
                else {
                    dp[r] = max(dp[r], get(p, l, r));
                }
                if (l != r) {
                    if (l != 0) {
                        dp[r] = max(dp[r], ST.query(l, r - 1) + p[r] + p[l - 1]);
                    }
                    else {
                        dp[r] = max(dp[r], ST.query(l, r - 1) + p[r]);
                    }
                }
            }
        }
        ST.change(r, dp[r] - 2 * p[r]);
    }

    cout << dp[n - 1];
}
```

Задача «Треугольники с началом координат»

Постановка задачи

Дано множество A из n точек на плоскости и q точек $p_1, \ldots, p_q$. Никакие три из точек $A \cup \{p_1, \ldots, p_q\} \cup \{O\}$ не коллинеарны, где $O = (0, 0)$. Для каждой точки p_i требуется подсчитать количество треугольников Oa_xa_y $(1 \le x < y \le n)$ таких, что p_i лежит строго внутри этого треугольника.

Решение

Рассмотрим треугольник OAB с вершинами $O(0,0)$, $A, B \in A$. Точка p лежит внутри OAB тогда и только тогда, когда она находится по ту же сторону от прямой OA, что и B, от прямой OB — что и A, и от прямой AB — что и O. В частности, проведём прямую через O и p. Поскольку отрезок AB соединяет точки, лежащие по разные стороны от этой прямой (иначе весь треугольник лежал бы в одной полуплоскости, и p не могла бы быть внутри), справедливо:

$$\text{Точка } p \text{ внутри } \triangle OAB \implies A \text{ и } B \text{ лежат по разные стороны от прямой } Op.$$

Обратное, разумеется, неверно, но это условие задаёт необходимое разбиение: для фиксированной p все точки A разделяются на два множества:

$$\mathcal{A} = \{\, a_i \mid \operatorname{cross}(a_i,\, p) > 0 \,\}, \qquad \mathcal{B} = \{\, a_i \mid \operatorname{cross}(a_i,\, p) < 0 \,\},$$

где $\operatorname{cross}(u, v) = u_x v_y - u_y v_x$ — ориентированная площадь. Точки с нулевым значением отсутствуют по условию неколлинеарности. Любой треугольник Oa_xa_y, содержащий p, обязан иметь одну вершину в $\mathcal{A}$, другую в $\mathcal{B}$.

Зафиксируем p. Поместим начало координат в p: каждой точке X сопоставим вектор $X - p$. Точка O переходит в $-p$. Для двух точек $A \in \mathcal{A},\ B \in \mathcal{B}$ треугольник OAB содержит p тогда и только тогда, когда луч pO (направленный противоположно вектору p) лежит строго внутри угла, образованного векторами $A - p$ и $B - p$. Иными словами, при обходе по кругу от направления на A до направления на B против часовой стрелки мы должны пройти через направление на O.

Пусть угол вектора V (измеренный от положительного направления оси Ox) равен $\operatorname{atan2}(V_y, V_x)$, приведённый к $[0, 2\pi)$. Обозначим угол точки X относительно p:

$$\varphi_X = \operatorname{atan2}(X_y - p_y,\ X_x - p_x), \qquad \varphi_X \in [0, 2\pi).$$

Угол направления на O равен $(\varphi_p + \pi) \bmod 2\pi$, где $\varphi_p = \operatorname{atan2}(p_y, p_x)$. Для фиксированной точки $A \in \mathcal{A}$ множество направлений на точки $B \in \mathcal{B}$, которые вместе с A дают треугольник, содержащий p, есть в точности открытый интервал углов $(\varphi_A,\ \varphi_A + \pi)$. Действительно, векторы $A - p$ и $B - p$ должны лежать по разные стороны от прямой, проходящей через p и O, причём O (точнее, $-p$) должен находиться внутри угла, образованного этими векторами. Угол между $A - p$ и $B - p$, измеренный против часовой стрелки, обязан быть меньше π и содержать направление на $-p$. Поскольку A и B принадлежат множествам $\mathcal{A}$ и $\mathcal{B}$ соответственно, их углы φ_A и φ_B лежат в противоположных полуплоскостях относительно направления φ_p. Из этого следует, что φ_B обязана попадать в интервал $(\varphi_A, \varphi_A + \pi)$ при измерении в положительном направлении; в противном случае угол между ними превышал бы π или луч на $-p$ оказывался бы вне угла.

Таким образом, для каждой точки $A \in \mathcal{A}$ допустимые точки $B \in \mathcal{B}$ — в точности те, чей угол φ_B лежит в интервале $(\varphi_A,\ \varphi_A + \pi)$. Если $\varphi_A + \pi \geq 2\pi$, интервал «перехлёстывает» через 0 и принимает вид $(\varphi_A,\ 2\pi) \cup [0,\ \varphi_A + \pi - 2\pi)$.

Чтобы эффективно подсчитать количество таких B сразу для всех A, мы сначала вычисляем углы всех точек из $\mathcal{B}$ и сортируем их по возрастанию, получая массив Φ_B. Затем для каждой точки $A \in \mathcal{A}$ определяем левую границу $L = \varphi_A$ и правую границу $R = \varphi_A + \pi$. Если $R \geq 2\pi$, заменяем R на $R - 2\pi$, сводя интервал к стандартному виду. Дальнейший подсчёт распадается на два случая. Если $L \leq R$, интервал не проходит через 2π, и искомое количество равно числу элементов Φ_B, строго больших L и строго меньших R. С помощью бинарного поиска это выражается как $\text{lower\_bound}(R) - \text{upper\_bound}(L)$. Если же $L > R$, интервал циклически оборачивается вокруг 2π и состоит из двух частей: $(\varphi_A, 2\pi)$ и $[0, \varphi_A + \pi - 2\pi)$. Тогда нужное число элементов равно сумме $(|\Phi_B| - \text{upper\_bound}(L)) + \text{lower\_bound}(R)$. Просуммировав полученные величины по всем $A \in \mathcal{A}$, мы получаем количество треугольников, содержащих точку p.

Временная сложность для одной запросной точки составляет $O(n \log n)$: $O(n)$ уходит на разделение точек по знаку векторного произведения, $O(n \log n)$ — на сортировку углов множества $\mathcal{B}$, и $O(|\mathcal{A}| \log n)$ — на бинарные поиски для каждой точки из $\mathcal{A}$. Общая сложность по всем q запросам равна $O(q\, n \log n)$, что с запасом укладывается в отведённые временные рамки.

Реализация

```
struct Point {
    long long x, y;
    Point() {}
    Point(long long x, long long y) :
        x(x),
        y(y)
    {}
};

long long cross(Point& a, Point& b) {
    return a.x * b.y - a.y * b.x;
}

void Solve() {
    long double PI = acosl(-1.0L);
    int n, q; cin >> n >> q;
    vector<Point> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i].x >> a[i].y;
    }

    vector<Point> p(q);
    for (int i = 0; i < q; i++) {
        cin >> p[i].x >> p[i].y;
    }

    for (int idx = 0; idx < q; idx++) {
        Point& pp = p[idx];

        vector<Point> A, B;
        for (Point& ai : a) {
            long long cr = cross(ai, pp);
            if (cr > 0) {
                A.push_back(ai);
            }
            else if (cr < 0) {
                B.push_back(ai);
            }
        }
```

```
40
41          vector<long double> Bang;
42          for (Point& b : B) {
43              long double ang = atan2l(b.y - pp.y, b.x - pp.x);
44              if (ang < 0) {
45                  ang += 2 * PI;
46              }
47              Bang.push_back(ang);
48          }
49          sort(Bang.begin(), Bang.end());
50
51          long long cnt = 0;
52          for (Point& ai : A) {
53              long double ang = atan2l(ai.y - pp.y, ai.x - pp.x);
54              if (ang < 0) {
55                  ang += 2 * PI;
56              }
57              long double L = ang;
58              long double R = ang + PI;
59              if (R >= 2 * PI) {
60                  R -= 2 * PI;
61                  cnt += (Bang.end() - upper_bound(Bang.begin(), Bang.end(), L)) +
62                      (lower_bound(Bang.begin(), Bang.end(), R) - Bang.begin());
63              }
64              else {
65                  cnt += (lower_bound(Bang.begin(), Bang.end(), R) -
66                      upper_bound(Bang.begin(), Bang.end(), L));
67              }
68          }
69          cout << cnt << '\n';
70      }
71  }
```

Задача «Сеть ЭВМ»

Постановка задачи

Дано дерево из n вершин, рёбра имеют вес $w_i \in \{-1, 0, 1\}$. Из каждой вершины i во все вершины $j \geq i$ отправляется сигнал. Начальная сила всех сигналов одинакова (можно считать равной нулю). При прохождении по ребру сила сигнала изменяется на вес этого ребра. Сигнал движется по кратчайшему пути.

Таким образом, каждому сигналу, идущему из i в j ($i \leq j$), соответствует сумма весов на единственном пути между i и j. Эта сумма и является итоговой силой сигнала.

Поступают m запросов, каждый задаёт две вершины a и b. Требуется найти, сколько различных значений силы имеют сигналы, прошедшие и через a, и через b (то есть такие сигналы, путь которых содержит обе указанные вершины).

Непрерывность множества значений

Рассмотрим все сигналы, путь которых проходит через фиксированное множество вершин. Ключевой факт состоит в том, что если существуют сигналы, проходящие через заданные вершины, с силами $s_{\min}$ и $s_{\max}$, то для любого целого s между ними также найдётся сигнал с силой s.

Это свойство обусловлено тем, что веса рёбер принимают значения $-1, 0, 1$, а меняя начало и конец пути внутри доступных областей, можно «сдвигать» итоговую сумму на $-1, 0, +1$ и получать все промежуточные значения.

Следовательно, для ответа на запрос достаточно найти минимальную и максимальную силу среди сигналов, проходящих через обе заданные вершины. Тогда количество различных значений равно

$$\max(0,\ s_{\max} - s_{\min} + 1).$$

Структура сигналов и экстремальные значения

Зафиксируем две вершины a и b. Сигнал из i в j проходит через обе тогда и только тогда, когда обе вершины лежат на простом пути $i \leftrightarrow j$ и при этом $i \leq j$. Поскольку порядок $i \leq j$ фиксирован, путь является направленным. Можно показать, что минимальная и максимальная суммы достигаются на путях, которые «уходят» из a и b в наиболее выгодные стороны, захватывая при необходимости всё дерево.

Обозначим через $\mathrm{up}(u)$ и $\mathrm{down}(u)$ экстремальные отклонения суммы от некоторого базового уровня при движении от вершины u вверх (к корню) или вниз (в поддеревья). Более точно, для каждой вершины u можно определить:

- $mn_{\mathrm{down}}(u)$ — минимальный прирост при движении вниз от u
- $mx_{\mathrm{down}}(u)$ — максимальный прирост при движении вниз от u
- $mn_{\mathrm{up}}(u)$ — минимальный прирост при движении вверх от u
- $mx_{\mathrm{up}}(u)$ — максимальный прирост при движении вверх от u

Все эти величины вычисляются так, что соответствующие пути лежат целиком в допустимом множестве (с учётом $i \leq j$), а приросты ≤ 0 для минимумов и ≥ 0 для максимумов.

Теперь, имея запрос (a, b), экстремальные силы сигналов, проходящих через обе вершины, можно собрать из двух независимых частей: одна отвечает за «ответвление» в сторону от одной вершины, другая — за ответвление от другой. В зависимости от расположения a и b есть три случая:

Случай 1: $a = b$. Путь, проходящий через данную вершину, может уходить концами в двух разных направлениях относительно неё. А именно, один конец может спускаться в какое-либо поддерево вершины a, а другой — в другое поддерево. Или же один конец спускается в поддерево, а второй уходит наверх (выше a по дереву). Чтобы получить минимальную возможную силу сигнала, нужно взять два наименьших значения среди: всех величин $mn_{\text{down\_child}}(a)$ для детей a, а также величины $mn_{\text{up}}(a)$. Это даёт mn_1 (самое маленькое) и mn_2 (второе по величине). Аналогично, для максимума берутся два наибольших значения из $mx_{\text{down\_child}}(a)$ и $mx_{\text{up}}(a)$ — это mx_1 и mx_2. Итоговая сила пробегает все целые значения от $mn_1 + mn_2$ до $mx_1 + mx_2$, поэтому ответ: $(mx_1 + mx_2) - (mn_1 + mn_2) + 1$.

Случай 2: одна из вершин является предком другой. Без ограничения общности, пусть a — предок b. Тогда любой сигнал, проходящий через обе вершины, выглядит так: его путь начинается где-то в поддереве b, поднимается до b, затем продолжает подниматься до a, после чего либо уходит наверх от a, либо спускается в какое-то поддерево a, отличное от того, в котором лежит b. Следовательно, один из двух минимальных приростов берётся из поддерева b: это $mn_{\text{down}}(b)$ (минимальный прирост от b вниз). Второй минимальный прирост формируется из «оставшейся» части пути от a: нужно выбрать минимум из $mn_{\text{up}}(a)$ и всех $mn_{\text{down\_child}}(a)$ для детей, не содержащих b. Это значение, обозначим его $mn_{\text{up\_except\_b}}(a)$, получается предподсчитанным методом, исключающим вклад ветки b. Тогда $mn_1 = mn_{\text{down}}(b)$, $mn_2 = mn_{\text{up\_except\_b}}(a)$. Аналогично, $mx_1 = mx_{\text{down}}(b)$, $mx_2 = mx_{\text{up\_except\_b}}(a)$. Итоговая сила пробегает все целые значения от $mn_1 + mn_2$ до $mx_1 + mx_2$, поэтому ответ: $(mx_1 + mx_2) - (mn_1 + mn_2) + 1$.

Случай 3: вершины a и b лежат в разных ветвях (ни одна не является предком другой). Тогда любой путь, проходящий через обе, проходит через их наименьшего общего предка l и должен заходить в поддерево a и в поддерево b. Экстремальные приросты берутся независимо из каждой ветви: $mn_1 = mn_{\text{down}}(a)$, $mn_2 = mn_{\text{down}}(b)$; $mx_1 = mx_{\text{down}}(a)$, $mx_2 = mx_{\text{down}}(b)$. Итоговая сила пробегает все целые значения от $mn_1 + mn_2$ до $mx_1 + mx_2$, поэтому ответ: $(mx_1 + mx_2) - (mn_1 + mn_2) + 1$.

Предподсчёт величин up и down

Выберем корень дерева – вершину 1. Для каждой вершины u вычислим $dist[u]$ – сумму весов рёбер на пути от корня до u. Это делается одним обходом в глубину. Выполним эйлеров обход дерева, записывая вершину при входе и при выходе. Для каждой вершины u запомним $in[u]$ – позицию первого входа в эйлеровом массиве, и $out[u]$ – позицию последнего выхода. Тогда поддерево вершины u соответствует отрезку $[in[u], out[u]]$ в этом массиве.

По эйлеровому массиву значений $dist$ построим две разреженные таблицы (Sparse Table) – для минимума и для максимума. Это позволит за $O(1)$ получать минимальное и максимальное значение $dist$ на любом отрезке, а значит, и в любом поддереве.

Определим для каждой вершины u:

$$mn_{\text{down}}(u) = \min_{v \in \text{поддерево } u} \big(dist[v] - dist[u]\big),$$
$$mx_{\text{down}}(u) = \max_{v \in \text{поддерево } u} \big(dist[v] - dist[u]\big).$$

Это минимальный и максимальный прирост расстояния при движении от u вниз по дереву. Заметим, что $mn_{\text{down}}(u) \le 0$, $mx_{\text{down}}(u) \ge 0$ (так как сама вершина u даёт разность 0).

Теперь для каждой вершины u необходимо иметь возможность быстро получать наилучшие значения по её сыновьям (веткам) и отбрасывать одну конкретную ветку. Для этого в каждой вершине u заводятся два мультимножества:

- $mns[u]$ — содержит для каждого сына s вершины u значение $mn_{\text{down}}(s) + w(u, s)$, то есть минимальный прирост при движении из u в поддерево сына s.
- $mxs[u]$ — содержит для каждого сына s вершины u значение $mx_{\text{down}}(s) + w(u, s)$, то есть максимальный прирост при движении из u в поддерево сына s.

Мультимножества автоматически поддерживают порядок, так что за $O(\log n)$ можно получить минимальный (максимальный) элемент и, если нужно, следующий за ним, исключая заданную ветвь.

Теперь научимся вычислять величины «вверх». Определим для вершины u:

$mn_{\text{up}}(u)$ = минимальный прирост пути, начинающегося из u и идущего в «остальную» часть дерева (не в поддерево
$mx_{\text{up}}(u)$ = максимальный прирост такого пути.

Будем считать, что путь может быть пустым (длины 0), поэтому при необходимости $mn_{\text{up}}(u) \le 0$, $mx_{\text{up}}(u) \ge 0$.

Для корня дерева (вершины 1) положим $mn_{\text{up}}(1) = 0$, $mx_{\text{up}}(1) = 0$ – выход за пределы дерева невозможен, прирост нулевой.

Для остальных вершин выполняется рекурсивный обход сверху вниз. Пусть мы находимся в родителе p и хотим вычислить величины для его сына s. При движении из s вверх путь может:

- Пойти из s в p (добавляя вес ребра $w(p, s)$), а затем продолжиться вверх от p в «оставшуюся» часть (уже посчитанную как $mn_{\text{up}}(p)$ и $mx_{\text{up}}(p)$), либо
- Из p уйти вниз в какое-либо другое поддерево (не s), выбрав наилучший прирост среди оставшихся детей p.

Таким образом, получаем:

$$mn_{\text{up}}(s) = \min\Big(mn_{\text{up}}(p) + w(p,s),\ \text{второй минимум (или первый, если ветка } s \text{ не лучшая) из } mns[p] + w(p,s)\Big),$$
$$mx_{\text{up}}(s) = \max\Big(mx_{\text{up}}(p) + w(p,s),\ \text{второй максимум (или первый) из } mxs[p] + w(p,s)\Big).$$

Заметим, что вторые минимумы и максимумы легко извлекаются из мультимножеств: берём первый элемент; если он соответствует запрещённой ветке s (т.е. его источник – сын s), то берём следующий элемент. Таким образом, после завершения обхода мы для любой вершины u можем мгновенно узнать:

- $mn_{\text{down}}(u)$, $mx_{\text{down}}(u)$ – минимум и максимум в поддереве относительно $dist[u]$;
- $mn_{\text{up}}(u)$, $mx_{\text{up}}(u)$ – минимум и максимум вне поддерева u относительно $dist[u]$;
- а также для каждого сына мы можем получить «второй» минимум/максимум, исключая этого сына, что используется в запросах при случае a – предок b.

Обработка запроса

При поступлении запроса (a, b) сначала определяем наименьшего общего предка $l = \text{lca}(a, b)$ (с помощью двоичного подъёма). Затем возможны случаи:

- $a = b$. Минимальные и максимальные величины берутся из предподсчитанных списков для вершины: два минимума из mns[a] и up(a), два максимума из mxs[a] и up(a).
- a — предок b. Тогда один минимум — $mn_{\text{down}}(b)$, второй — $mn_{\text{up}}(a)$ с исключением той ветки, в которой лежит b (чтобы не учесть её же дважды). Аналогично для максимумов.
- иначе (a и b в разных ветвях). Тогда оба минимума — $mn_{\text{down}}(a)$ и $mn_{\text{down}}(b)$, оба максимума — $mx_{\text{down}}(a)$ и $mx_{\text{down}}(b)$.

После получения двух минимумов и двух максимумов ответ даётся формулой $(mx_1 + mx_2) - (mn_1 + mn_2) + 1$.

Реализация

```
class SparseTablemin {
private:
    vector<vector<int>> st;
    vector<int> logTable;
    int neutral = 2e9;

    void buildLogTable(int n) {
        logTable.resize(n + 1);
        logTable[0] = logTable[1] = 0;
        for (int i = 2; i <= n; i++) {
            logTable[i] = logTable[i / 2] + 1;
        }
    }

public:
    SparseTablemin(const vector<int>& a_) {
        vector<int> a = a_;
        while ((a.size() & (a.size() - 1)) != 0) {
            a.push_back(neutral);
        } int n = a.size();

        buildLogTable(n);

        int K = logTable[n] + 1;
        st.resize(K, vector<int>(n, neutral));

        for (int i = 0; i < n; i++) {
            st[0][i] = a[i];
        }

        for (int k = 1; k < K; k++) {
            for (int i = 0; i + (1 << k) <= n; i++) {
                st[k][i] = min(st[k - 1][i], st[k - 1][i + (1 << (k - 1))]);
            }
        }
    }

    int query(int l, int r) {
        int k = logTable[r - l + 1];
        return min(st[k][l], st[k][r - (1 << k) + 1]);
    }
};



class SparseTablemax {
private:
    vector<vector<int>> st;
    vector<int> logTable;
    int neutral = -2e9;

    void buildLogTable(int n) {
        logTable.resize(n + 1);
        logTable[0] = logTable[1] = 0;
        for (int i = 2; i <= n; i++) {
            logTable[i] = logTable[i / 2] + 1;
        }
    }

```

```
public:
    SparseTablemax(const vector<int>& a_) {
        vector<int> a = a_;
        while ((a.size() & (a.size() - 1)) != 0) {
            a.push_back(neutral);
        } int n = a.size();

        buildLogTable(n);

        int K = logTable[n] + 1;
        st.resize(K, vector<int>(n, neutral));

        for (int i = 0; i < n; i++) {
            st[0][i] = a[i];
        }

        for (int k = 1; k < K; k++) {
            for (int i = 0; i + (1 << k) <= n; i++) {
                st[k][i] = max(st[k - 1][i], st[k - 1][i + (1 << (k - 1))]);
            }
        }
    }

    int query(int l, int r) {
        int k = logTable[r - l + 1];
        return max(st[k][l], st[k][r - (1 << k) + 1]);
    }
};

vector<vector<pair<int, int>>> g;
map<pair<int, int>, int> W;
vector<int> a;
vector<int> dist;

const int LOGN = 23;

vector<vector<int>> parent;
vector<int> depth;

void DFS(int v, int p, int d) {
    depth[v] = d;
    parent[v][0] = p;
    for (int i = 1; i < LOGN; ++i) {
        parent[v][i] = parent[parent[v][i - 1]][i - 1];
    }
    for (auto [u, w] : g[v]) {
        if (u != p) {
            DFS(u, v, d + 1);
        }
    }
}

int lca(int u, int v) {
    if (depth[u] < depth[v]) {
        swap(u, v);
    }
    for (int i = LOGN - 1; i >= 0; --i) {
        if (depth[u] - (1 << i) >= depth[v]) {
            u = parent[u][i];
        }
    }
```

```
    if (u == v) {
        return u;
    }
    for (int i = LOGN - 1; i >= 0; --i) {
        if (parent[u][i] != parent[v][i]) {
            u = parent[u][i];
            v = parent[v][i];
        }
    }
    return parent[u][0];
}

vector<vector<pair<int, int>>> tree;

void dfs(int u, int p) {
    a.push_back(u);
    for (auto [v, w] : g[u]) {
        if (v != p) {
            dist[v] = dist[u] + w;
            tree[u].push_back({ v, 0 });
            dfs(v, u);
        }
    }
    a.push_back(u);
}

vector<multiset<int>> mns;
vector<multiset<int, greater<int>>> mxs;

vector<int> mn_up;
vector<int> mx_up;

vector<int> in;
vector<int> out;

int get_mn_down(int u, SparseTablemin& ST) {
    return ST.query(in[u], out[u]);
}

int get_mx_down(int u, SparseTablemax& ST) {
    return ST.query(in[u], out[u]);
}

int Get(vector<pair<int, int>>& Z, int t) {
    int l = -1;
    int r = Z.size();
    while (l + 1 != r) {
        int m = (l + r) / 2;
        if (in[Z[m].first] <= t) {
            l = m;
        }
        else {
            r = m;
        }
    }
    return l;
}
```

```
182 int get_mn_up(int u, int v, SparseTablemin& ST) {
183     int t = Get(tree[u], in[v]);
184 
185     if (t != -1) {
186         auto [k, _] = tree[u][t];
187         if (in[k] <= in[v] && out[k] >= out[v]) {
188             if (mns[u].size() <= 1) {
189                 return mn_up[u];
190             }
191             auto set_it = mns[u].begin();
192             if (get_mn_down(u, ST) - dist[u] == get_mn_down(k, ST) - dist[k] + W[{u,
                    ↪ k}]) {
193                 ++set_it;
194             }
195             return min(*set_it, mn_up[u]);
196         }
197     }
198 
199     return mn_up[u];
200 }
201 
202 int get_mx_up(int u, int v, SparseTablemax& ST) {
203     int t = Get(tree[u], in[v]);
204 
205     if (t != -1) {
206         auto [k, _] = tree[u][t];
207         if (in[k] <= in[v] && out[k] >= out[v]) {
208             if (mxs[u].size() <= 1) {
209                 return mx_up[u];
210             }
211             auto set_it = mxs[u].begin();
212             if (get_mx_down(u, ST) - dist[u] == get_mx_down(k, ST) - dist[k] + W[{u,
                    ↪ k}]) {
213                 ++set_it;
214             }
215             return max(*set_it, mx_up[u]);
216         }
217     }
218 
219     return mx_up[u];
220 }
221 
222 void Solve() {
223     int n; cin >> n;
224     tree.resize(n);
225     g.resize(n);
226     for (int i = 0; i < n - 1; i++) {
227         int u, v, w; cin >> u >> v >> w;
228         --u; --v;
229         g[u].push_back({ v, w });
230         g[v].push_back({ u, w });
231         W[{u, v}] = w;
232         W[{v, u}] = w;
233     }
234 
235     parent.resize(n, vector<int>(LOGN));
236     depth.resize(n); DFS(0, 0, 0);
237     dist.resize(n); dfs(0, -1);
238     in.resize(n); for (int i = 0; i < n; i++) in[i] = -1;
239     out.resize(n); for (int i = 0; i < n; i++) out[i] = -1;
240 
```

```
241     vector<int> time(2 * n);
242     for (int i = 0; i < 2 * n; i++) {
243         if (in[a[i]] == -1) {
244             in[a[i]] = i; time[i] = dist[a[i]];
245         }
246         else {
247             out[a[i]] = i; time[i] = dist[a[i]];
248         }
249     }
250 
251     SparseTablemin STmin(time);
252     SparseTablemax STmax(time);
253 
254     mns.resize(n); mxs.resize(n); mn_up.resize(n); mx_up.resize(n);
255     for (int i = 0; i < n; i++) mn_up[i] = 0;
256     for (int i = 0; i < n; i++) mx_up[i] = 0;
257 
258     function<void(int, int)> go = [&](int u, int p) {
259         if (p != -1) {
260             mn_up[u] = mn_up[p] + W[{u, p}];
261             if (mn_up[u] > 0) {
262                 mn_up[u] = 0;
263             }
264             mx_up[u] = mx_up[p] + W[{u, p}];
265             if (mx_up[u] < 0) {
266                 mx_up[u] = 0;
267             }
268             if (mns[p].size() > 1) {
269                 auto it = mns[p].begin();
270                 if (get_mn_down(p, STmin) - dist[p] == get_mn_down(u, STmin) - dist[u
                        ↪ ] + W[{u, p}]) {
271                     ++it;
272                 }
273                 mn_up[u] = min(mn_up[u], *it + W[{u, p}]);
274             }
275             if (mxs[p].size() > 1) {
276                 auto it = mxs[p].begin();
277                 if (get_mx_down(p, STmax) - dist[p] == get_mx_down(u, STmax) - dist[u
                        ↪ ] + W[{u, p}]) {
278                     ++it;
279                 }
280                 mx_up[u] = max(mx_up[u], *it + W[{u, p}]);
281             }
282             if (mn_up[u] > 0) {
283                 mn_up[u] = 0;
284             }
285             if (mx_up[u] < 0) {
286                 mx_up[u] = 0;
287             }
288         }
289         for (auto [v, w] : g[u]) {
290             if (v != p) {
291                 mns[u].insert(get_mn_down(v, STmin) - dist[v] + w);
292                 mxs[u].insert(get_mx_down(v, STmax) - dist[v] + w);
293             }
294         }
295         for (auto [v, w] : g[u]) {
296             if (v != p) go(v, u);
297         }
298     }; go(0, -1);
299 
```

```
300     int q; cin >> q;
301     for (int i = 0; i < q; i++) {
302         int u, v; cin >> u >> v;
303         --u; --v;
304         int mn1, mn2, mx1, mx2;
305         if (u == v) {
306             vector<int> MN;
307             auto it1 = mns[u].begin();
308             if (mns[u].size() > 0) MN.push_back(*it1);
309             ++it1;
310             if (mns[u].size() > 1) MN.push_back(*it1);
311             MN.push_back(mn_up[u]);
312 
313             vector<int> MX;
314             auto it2 = mxs[u].begin();
315             if (mxs[u].size() > 0) MX.push_back(*it2);
316             ++it2;
317             if (mxs[u].size() > 1) MX.push_back(*it2);
318             MX.push_back(mx_up[u]);
319 
320             sort(MN.begin(), MN.end());
321             mn1 = 0;
322             if (MN.size() > 0) mn1 = MN[0];
323             mn2 = 0;
324             if (MN.size() > 1) mn2 = MN[1];
325             sort(MX.begin(), MX.end());
326             mx1 = 0;
327             if (MX.size() > 0) mx1 = MX[MX.size() - 1];
328             mx2 = 0;
329             if (MX.size() > 1) mx2 = MX[MX.size() - 2];
330         }
331         else if (lca(u, v) == u) {
332             mn1 = get_mn_down(v, STmin) - dist[v];
333             mn2 = get_mn_up(u, v, STmin);
334             mx1 = get_mx_down(v, STmax) - dist[v];
335             mx2 = get_mx_up(u, v, STmax);
336         }
337         else if (lca(u, v) == v) {
338             swap(u, v);
339             mn1 = get_mn_down(v, STmin) - dist[v];
340             mn2 = get_mn_up(u, v, STmin);
341             mx1 = get_mx_down(v, STmax) - dist[v];
342             mx2 = get_mx_up(u, v, STmax);
343         }
344         else {
345             mn1 = get_mn_down(v, STmin) - dist[v];
346             mn2 = get_mn_down(u, STmin) - dist[u];
347             mx1 = get_mx_down(v, STmax) - dist[v];
348             mx2 = get_mx_down(u, STmax) - dist[u];
349         }
350         if (mn1 > 0) mn1 = 0;
351         if (mn2 > 0) mn2 = 0;
352         if (mx1 < 0) mx1 = 0;
353         if (mx2 < 0) mx2 = 0;
354         cout << (mx1 + mx2) - (mn1 + mn2) + 1 << '\n';
355     }
356 }
```

Задача «Счастливые билеты!!!»

Постановка задачи

Билет длины n (чётное число) считается счастливым, если сумма первых $n/2$ цифр равна сумме последних $n/2$ цифр. Цифры могут быть от 0 до 9, билет может начинаться с нулей. Для каждого запроса с чётным числом n требуется найти количество счастливых билетов длины n по модулю 998244353.

Производящая функция

Положим $k = n/2$. Тогда билет длины $2k$ состоит из двух половин по k цифр. Пусть $a_k(S)$ — число способов выбрать k цифр (от 0 до 9) с суммой S ($0 \le S \le 9k$). Тогда количество счастливых билетов длины $2k$ равно $\text{ans}(k) = \sum_{S=0}^{9k} a_k(S)^2 \pmod{998244353}$. Для каждой суммы S первая и вторая половины независимо дают $a_k(S)$ вариантов, что даёт $a_k(S)^2$ билетов.

Введём производящий многочлен для одной цифры $P(x) = 1 + x + x^2 + \cdots + x^9 = \frac{1-x^{10}}{1-x}$.

Тогда $a_k(S)$ — коэффициент при x^S в $P(x)^k$. Сумма квадратов коэффициентов $a_k(S)^2$ равна свободному члену (коэффициенту при x^0) в произведении $P(x)^k \cdot P(x^{-1})^k$: $\text{ans}(k) = [x^0]\big(P(x)P(x^{-1})\big)^k$. Преобразуем $C(x) = P(x)P(x^{-1})$:

$$C(x) = \frac{1-x^{10}}{1-x} \cdot \frac{1-x^{-10}}{1-x^{-1}} = \frac{(x^{10}-1)(x^{-10}-1)}{(x-1)(x^{-1}-1)} = \frac{-(x^{10}-1)^2/x^{10}}{-(x-1)^2/x} = \frac{(x^{10}-1)^2}{x^9(x-1)^2}.$$

Таким образом, $C(x)^k = \frac{(x^{10}-1)^{2k}}{x^{9k}(x-1)^{2k}}$. Искомый коэффициент при x^0 в $C(x)^k$ равен коэффициенту при x^{9k} в $\frac{(x^{10}-1)^{2k}}{(x-1)^{2k}} = (1-x^{10})^{2k} \cdot (1-x)^{-2k}$. Разложим оба множителя:

$$(1-x^{10})^{2k} = \sum_{j=0}^{2k} (-1)^j \binom{2k}{j} x^{10j},$$

$$(1-x)^{-2k} = \sum_{i=0}^{\infty} \binom{2k+i-1}{i} x^i.$$

Коэффициент при x^{9k} в произведении даётся суммой по всем j, для которых $10j \le 9k$:

$$\text{ans}(k) = \sum_{j=0}^{\lfloor 9k/10 \rfloor} (-1)^j \binom{2k}{j} \binom{2k + (9k-10j) - 1}{9k-10j} = \sum_{j=0}^{\lfloor 9k/10 \rfloor} (-1)^j \binom{2k}{j} \binom{11k-10j-1}{2k-1}$$

Асимптотика и предподсчёт

Для каждого запроса с параметром k суммируется не более $0.9k$ слагаемых. По условию сумма всех n_i (а значит, и сумма $k_i = n_i/2$) не превышает 10^7. Следовательно, суммарное количество итераций по всем запросам не превзойдёт 10^7.

Остаётся научиться быстро вычислять биномиальные коэффициенты $\binom{N}{M}$ по простому модулю 998244353. Для этого достаточно предподсчитать факториалы и обратные к ним факториалы до максимального необходимого N, равного $\max(11k) \le 55 \cdot 10^6 \le 10^8$.

Реализация

```
void Solve() {
    int MOD = 998244353;
    auto POW = [&](int a, int b) {
        int ans = 1; a %= MOD; b %= MOD;
        while (b) {
            if (b & 1) {
                ans = (ans * a) % MOD;
            } a = (a * a) % MOD; b >>= 1;
        } return ans;
    };

    int t; cin >> t;
    vector<int> Q(t);
    int mx = 0;
    for (int i = 0; i < t; i++) {
        int n; cin >> n;
        Q[i] = n;
        mx = max(mx, 11 * n / 2);
    }

    vector<int> fact(mx + 1), invfact(mx + 1);
    fact[0] = 1;
    for (int i = 1; i <= mx; i++) {
        fact[i] = 1LL * fact[i - 1] * i % MOD;
    }
    invfact[mx] = POW(fact[mx], MOD - 2);
    for (int i = mx - 1; i >= 0; i--) {
        invfact[i] = 1LL * invfact[i + 1] * (i + 1) % MOD;
    }

    auto C = [&](int n, int k) {
        if (k < 0 || k > n) {
            return 0LL;
        }
        return 1LL * fact[n] * invfact[k] % MOD * invfact[n - k] % MOD;
    };

    for (int n : Q) {
        int k = n / 2;
        long long ans = 0;
        for (int j = 0; j <= 9 * k / 10; j++) {
            int K = C(2 * k, j) * C(11 * k - 10 * j - 1, 2 * k - 1) % MOD;
            if (j % 2 == 1) {
                ans = (ans - K + MOD) % MOD;
            }
            else {
                ans = (ans + K) % MOD;
            }
        }
        cout << ans << '\n';
    }
}
```