

algo

Эта книга представляет собой архив сайта e-maxx.ru/algo по состоянию на 8 Sep 2010 19:20.

Алгебра

элементарные алгоритмы

- Функция Эйлера и её вычисление
- Бинарное возведение в степень за $O(\log N)$
- Алгоритм Евклида нахождения НОД (наибольшего общего делителя)
- Решето Эратосфена
- Расширенный алгоритм Евклида
- Числа Фибоначчи и их быстрое вычисление
- Обратный элемент в кольце по модулю
- Код Грея
- Длинная арифметика
- Дискретное логарифмирование по модулю M алгоритмом baby-step-giant-step Шэнкса за $O(\sqrt{M} \log M)$
- Диофантовы уравнения с двумя неизвестными: $AX+BY=C$
- Линейное модулярное уравнение с одной неизвестной: $AX=B$
- Китайская теорема об остатках. Алгоритм Гарнера
- Нахождение степени делителя факториала
- Троичная сбалансированная система счисления
- Вычисление факториала $N!$ по модулю P за $O(P \log N)$
- Перебор всех подмасок данной маски. Оценка 3^N для суммарного количества подмасок всех масок
- Первообразный корень. Алгоритм нахождения
- Дискретное извлечение корня

сложные алгоритмы

- Тест BPSW на простоту чисел за $O(\log N)$
- Эффективные алгоритмы факторизации: Полларда $p-1$, Полларда p , Бента, Полларда Монте-Карло, Ферма
- Быстрое преобразование Фурье за $O(N \log N)$. Применение к умножению двух полиномов или длинных чисел

Графы

элементарные алгоритмы

- Поиск в ширину
- Поиск в глубину
- Топологическая сортировка за $O(N + M)$
- Поиск компонент связности за $O(N + M)$

компоненты сильной связности, мосты и т.д.

- Поиск компонент сильной связности, построение конденсации графа за $O(N + M)$
- Поиск мостов за $O(N + M)$
- Поиск точек сочленения за $O(N + M)$

кратчайшие пути из одной вершины

- Алгоритм Дейкстры нахождения кратчайших путей от заданной вершины до всех остальных вершин за $O(N^2 + M)$
- Алгоритм Дейкстры для разреженного графа нахождения кратчайших путей от заданной вершины до всех остальных вершин за $O(M \log N)$
- Алгоритм Форда-Беллмана нахождения кратчайших путей от заданной вершины до всех остальных вершин за $O(NM)$
- Алгоритм Левита нахождения кратчайших путей от заданной вершины до всех остальных вершин за $O(NM)$

кратчайшие пути между всеми парами вершин

- Нахождение кратчайших путей между всеми парами вершин графа методом Флойда-Уоршюла за $O(N^3)$

минимальный остов

- Минимальное остовное дерево. Алгоритм Прима за $O(NM)$ и за $O(M \log N + N^2)$
- Минимальное остовное дерево. Алгоритм Крускала за $O(M \log N + N^2)$
- Минимальное остовное дерево. Алгоритм Крускала со структурой данных 'система непересекающихся множеств' за $O(M \log N)$
- Матричная теорема Кирхгофа. Нахождение количества остовных деревьев за $O(N^3)$

ЦИКЛЫ

- Нахождение отрицательного цикла в графе за $O(NM)$
- Нахождение Эйлерова пути или Эйлерова цикла за $O(M)$
- Проверка графа на ацикличность и нахождение цикла за $O(M)$

наименьший общий предок (LCA)

- Наименьший общий предок. Нахождение за $O(\sqrt{N})$ и $O(\log N)$ с препроцессингом $O(N)$
- Наименьший общий предок. Нахождение за $O(\log N)$ с препроцессингом $O(N \log N)$ (метод двоичного подъёма)
- Наименьший общий предок. Нахождение за $O(1)$ с препроцессингом $O(N)$ (алгоритм Фарах-Колтона и Бендерса)
- Задача RMQ (Range Minimum Query - минимум на отрезке). Решение за $O(1)$ с препроцессингом $O(N)$
- Наименьший общий предок. Нахождение за $O(1)$ в режиме оффлайн (алгоритм Тарьяна)

потоки и связанные с ними задачи

- Метод Эдмондса-Карпа нахождения максимального потока за $O(NM^2)$
- Метод Проталкивания предпотока нахождения максимального потока за $O(N^4)$
- Модификация метода Проталкивания предпотока за $O(N^3)$
- Поток с ограничениями
- Поток минимальной стоимости (min-cost-flow). Алгоритм увеличивающих путей за $O(N^3M)$
- Задача о назначениях. Решение с помощью min-cost-flow за $O(N^5)$
- Задача о назначениях. Венгерский алгоритм (алгоритм Куна) за $O(N^4)$
- Нахождение минимального разреза алгоритмом Штор-Вагнера за $O(N^3)$
- Поток минимальной стоимости, циркуляция минимальной стоимости. Алгоритм удаления циклов отрицательного веса
- Алгоритм Диница нахождения максимального потока

паросочетания и связанные с ними задачи

- Алгоритм Куна нахождения наибольшего паросочетания за $O(NM)$
- Проверка графа на двудольность и разбиение на две доли за $O(M)$
- Нахождение наибольшего по весу вершинно-взвешенного паросочетания за $O(N^3)$
- Алгоритм Эдмондса нахождения наибольшего паросочетания в произвольных графах за $O(N^3)$
- Покрытие путями ориентированного ациклического графа

связность

- Рёберная связность. Свойства и нахождение
- Вершинная связность. Свойства и нахождение
- Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин

К-ые пути

- Нахождение K-го кратчайшего пути без циклов с помощью бинарного поиска за $O(N^2K \log W)$

обратные задачи

- Обратная задача SSSP (inverse-SSSP - обратная задача кратчайших путей из одной вершины) за $O(M)$
- Обратная задача MST (inverse-MST - обратная задача минимального остова) за $O(NM^2)$

разное

- Покраска рёбер дерева (структуры данных) - решение за $O(\log N)$
- Задача 2-SAT (2-CNF). Решение за $O(N + M)$

Геометрия

элементарные алгоритмы

- Длина объединения отрезков на прямой за $O(N \log N)$
- Ориентированная площадь треугольника и предикат 'По часовой стрелке'
- Проверка двух отрезков на пересечение
- Нахождение уравнения прямой для отрезка
- Нахождение точки пересечения двух прямых
- Нахождение точки пересечения двух отрезков
- Нахождение площади простого многоугольника за $O(N)$
- Теорема Пика. Нахождение площади решётчатого многоугольника за $O(1)$
- Задача о покрытии отрезков точками

более сложные алгоритмы

- Пересечение окружности и прямой
 - Пересечение двух окружностей
 - Построение выпуклой оболочки алгоритмом Грэхэма-Эндрю за $O(N \log N)$
 - Нахождение площади объединения треугольников. Метод вертикальной декомпозиции
 - Проверка точки на принадлежность выпуклому многоугольнику за $O(\log N)$
 - Нахождение вписанной окружности в выпуклом многоугольнике с помощью тернарного поиска за $O(N \log^2 C)$
 - Нахождение вписанной окружности в выпуклом многоугольнике методом сжатия сторон за $O(N \log N)$
 - Диаграмма Вороного в двумерном случае, её свойства, применение. Простейший алгоритм построения за $O(N^4)$
 - Нахождение всех граней, внешней грани планарного графа за $O(N \log N)$
 - Нахождение пары ближайших точек алгоритмом разделяй-и-властвуй за $O(N \log N)$
 - Нахождение треугольника наименьшей площади за $O(N^2 \log N)$
-

Строки

- Z-функция строки и её вычисление за $O(N)$
 - Префикс-функция, её вычисление и применения. Алгоритм Кнута-Морриса-Пратта
 - Алгоритмы хэширования в задачах на строки
 - Алгоритм Рабина-Карпа поиска подстроки в строке за $O(N)$
 - Определение количества различных подстрок за $O(N^2 \log N)$
 - Разбор выражений за $O(N)$. Обратная польская нотация
 - Сuffixный массив. Построение за $O(N \log N)$ и применения
 - Сuffixный автомат. Построение за $O(N)$ и применения
 - Нахождение всех подпалиндромов за $O(N)$
 - Декомпозиция Линдана. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига за $O(N)$ времени и $O(1)$ памяти
 - Алгоритм Ахо-Корасик
 - Поиск подстроки в строке с помощью Z- или Префикс-функции. Алгоритм Кнута-Морриса-Пратта
 - Решение задачи "сжатие строки"
-

Структуры данных

- Sqrt-декомпозиция
 - Дерево Фенвика
 - Система непересекающихся множеств
 - Дерево отрезков
 - Декартово дерево (treap, дерамида)
 - Модификация стека и очереди для извлечения минимума за $O(1)$
 - Рандомизированная куча
-

Алгоритмы на последовательностях

- Задача RMQ (Range Minimum Query - минимум на отрезке)
 - Нахождение наи длиннейшей возрастающей подпоследовательности за $O(N^2)$
 - Нахождение наи длиннейшей возрастающей подпоследовательности за $O(N \log N)$
 - K-ая порядковая статистика за $O(N)$
-

Динамика

- Динамика по профилю. Задача "паркет"
 - Нахождение наибольшей нулевой подматрицы за $O(N M)$
-

Линейная алгебра

- Вычисление определителя методом Краута за $O(N^3)$
 - Метод Гаусса решения системы линейных уравнений за $O(N^3)$
 - Нахождение ранга матрицы за $O(N^3)$
 - Вычисление определителя матрицы методом Гаусса за $O(N^3)$
-

Численные методы

- Интегрирование по формуле Симпсона
- Поиск корней методом Ньютона (касательных)
- Тернарный поиск

Комбинаторика

- Биномиальные коэффициенты
- Числа Каталана
- Ожерелья
- Расстановка слонов на шахматной доске
- Правильные скобочные последовательности. Нахождение лексикографически следующей, К-ой, определение номера
- Количество помеченных графов, связных помеченных графов, помеченных графов с К компонентами связности
- Генерация сочетаний из N элементов
- Лемма Бернсайда. Теорема Пойа

Теория игр

- Игры на произвольных графах. Метод ретроспективного анализа за $O(M)$
- Теория Шрага-Гранди. Ним

Расписания

- Задача Джонсона при $N = 1$
- Задача Джонсона при $N = 2$
- Оптимальный выбор заданий при известных временах завершения и длительностях выполнения

Разное

- Задача Иосифа
- Игра Пятнашки: существование решения
- Дерево Штерна-Броко. Ряд Фарея

Функция Эйлера

Определение

Функция Эйлера $\phi(n)$ или $phi(n)$ — это количество чисел от 1 до n , взаимно простых с n .

Например, $\phi(1) = 1$, $\phi(2) = 1$, $\phi(3) = 2$, $\phi(4) = 2$, $\phi(5) = 4$.

Свойства

Если p — простое, то $\phi(p) = p - 1$. Кроме того, $\phi(p^a) = p^a - p^{a-1}$.

Если a и b взаимно простые, то $\phi(ab) = \phi(a)\phi(b)$.

Отсюда можно получить функцию Эйлера для любого n через его факторизацию (разложение на простые множители):

$$\begin{aligned}\phi(n) &= \phi(p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}) = \phi(p_1^{a_1})\phi(p_2^{a_2}) \cdots \phi(p_k^{a_k}) = \\ &= (p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1}) \cdots (p_k^{a_k} - p_k^{a_k-1}) = p_1^{a_1} \left(1 - \frac{1}{p_1}\right) p_2^{a_2} \left(1 - \frac{1}{p_2}\right) \cdots p_k^{a_k} \left(1 - \frac{1}{p_k}\right) = \\ &= n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)\end{aligned}$$

Реализация

Простейший код, вычисляющий функцию Эйлера, факторизуя число простейшим методом (за $O(\sqrt{n})$):

```
int phi (int n) {
    int result = n;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    if (n > 1)
        result -= result / n;
    return result;
}
```

Ключевое место — это нахождение **факторизации** числа n , что можно осуществить за время, значительно меньшее $O(\sqrt{n})$: см. [Эффективные алгоритмы факторизации](#).

Приложения функции Эйлера

Самое известное и важное свойство функции Эйлера выражается в **теореме Эйлера**:

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

где a и m взаимно прости.

В частном случае, когда m простое, теорема Эйлера превращается в так называемую **малую теорему Ферма**:

$$a^{m-1} \equiv 1 \pmod{m}$$

Теорема Эйлера достаточно часто встречается в практических приложениях, например, см. [Обратный элемент в поле по модулю](#).

Бинарное возведение в степень

Бинарное возведение в степень — это приём, позволяющий возводить любое число в n -ую степень за $O(\log n)$ умножений (вместо n умножений при обычном подходе).

Более того, описываемый здесь приём применим к любой **ассоциативной** операции, а не только к умножению чисел. Напомним, операция называется ассоциативной, если для любых a, b, c выполняется:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Наиболее очевидное обобщение — на остатки по некоторому модулю (очевидно, ассоциативность сохраняется). Следующим по "популярности" является обобщение на произведение матриц (его ассоциативность общеизвестна).

Алгоритм

Заметим, что для любого числа a и чётного числа n выполнимо очевидное тождество (следующее из ассоциативности операции умножения):

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

Оно и является основным в методе бинарного возведения в степень. Действительно, для чётного n мы показали, как, потратив всего одну операцию умножения, можно свести задачу к вдвое меньшей степени.

Осталось понять, что делать, если степень n нечётна. Здесь мы поступаем очень просто: перейдём к степени $n - 1$, которая будет уже чётной:

$$a^n = a^{n-1} \cdot a$$

Итак, мы фактически нашли рекуррентную формулу: от степени n мы переходим, если она чётна, к $n/2$, а иначе — к $n - 1$. Понятно, что всего будет не более $2 \log n$ переходов, прежде чем мы придём к $n = 0$ (базе рекуррентной формулы). Таким образом, мы получили алгоритм, работающий за $O(\log n)$ умножений.

Реализация

Простейшая рекурсивная реализация:

```
int binpow (int a, int n) {
```

```

if (n == 0)
    return 1;
if (n % 2 == 1)
    return binpow (a, n-1) * a;
else {
    int b = binpow (a, n/2);
    return b * b;
}

```

Нерекурсивная реализация, оптимизированная (деления на 2 заменены битовыми операциями):

```

int binpow (int a, int n) {
    int res = 1;
    while (n)
        if (n & 1) {
            res *= a;
            --n;
        }
        else {
            a *= a;
            n >>= 1;
        }
    return res;
}

```

Алгоритм Евклида нахождения НОД (наибольшего общего делителя)

Даны два целых неотрицательных числа a и b . Требуется найти их наибольший общий делитель, т.е. наибольшее число, которое является делителем одновременно и a , и b . На английском языке "наибольший общий делитель" пишется "greatest common divisor", и распространённым его обозначением является \gcd :

$$\gcd(a, b) = \max_{k=1\dots\infty : k|a \& k|b} k$$

(здесь символом " $|$ " обозначена делимость, т.е. " $k|a$ " обозначает " k делит a ")

Когда оно из чисел равно нулю, а другое отлично от нуля, их наибольшим общим делителем, согласно определению, будет это второе число. Когда оба числа равны нулю, результат не определён (подойдёт любое бесконечно большое число), мы положим в этом случае наибольший общий делитель равным нулю. Поэтому можно говорить о таком правиле: если одно из чисел равно нулю, то их наибольший общий делитель равен второму числу.

Алгоритм Евклида, рассмотренный ниже, решает задачу нахождения наибольшего общего делителя двух чисел a и b за $O(\log \min(a, b))$.

Данный алгоритм был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.), хотя, вполне возможно, этот алгоритм имеет более раннее происхождение.

Алгоритм

Сам алгоритм чрезвычайно прост и описывается следующей формулой:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b=0 \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Реализация

```

int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}

```

Используя тернарный условный оператор C++, алгоритм можно записать ещё короче:

```

int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}

```

Наконец, приведём и нерекурсивную форму алгоритма:

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap (a, b);
    }
    return a;
}
```

Доказательство корректности

Сначала заметим, что при каждой итерации алгоритма Евклида его второй аргумент строго убывает, следовательно, поскольку он неотрицательный, то алгоритм Евклида **всегда завершается**.

Для **доказательства корректности** нам необходимо показать, что $\gcd(a, b) = \gcd(b, a \bmod b)$ для любых $a \geq 0, b > 0$.

Покажем, что величина, стоящая в левой части равенства, делится на настоящую в правой, а стоящая в правой — делится на настоящую в левой. Очевидно, это будет означать, что левая и правая части совпадают, что и докажет корректность алгоритма Евклида.

Обозначим $d = \gcd(a, b)$. Тогда, по определению, $d|a$ и $d|b$.

Далее, разложим остаток от деления a на b через их частное:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Но тогда отсюда следует:

$$d | (a \bmod b)$$

Итак, вспоминая утверждение $d|b$, получаем систему:

$$\begin{cases} d | b, \\ d | (a \bmod b) \end{cases}$$

Воспользуемся теперь следующим простым фактом: если для каких-то трёх чисел p, q, r выполнено: $p|q$ и $p|r$, то выполняется и: $p | \gcd(q, r)$. В нашей ситуации получаем:

$$d | \gcd(b, a \bmod b)$$

Или, подставляя вместо d его определение как $\gcd(a, b)$, получаем:

$$\gcd(a, b) | \gcd(b, a \bmod b)$$

Итак, мы провели половину доказательства: показали, что левая часть делит правую. Вторая половина доказательства производится аналогично.

Время работы

Время работы алгоритма оценивается **теоремой Ламе**, которая устанавливает удивительную связь алгоритма Евклида и последовательности Фибоначчи:

Если $a > b \geq 1$ и $b < F_n$ для некоторого n , то алгоритм Евклида выполнит не более $n - 2$ рекурсивных вызовов.

Более того, можно показать, что верхняя граница этой теоремы — оптимальная. При $a = F_n, b = F_{n-1}$ будет выполнено именно $n - 2$ рекурсивных вызова. Иными словами, **последовательные числа Фибоначчи — наихудшие входные данные** для алгоритма Евклида.

Учитывая, что числа Фибоначчи растут экспоненциально (как константа в степени n), получаем, что алгоритм Евклида выполняется за $O(\log \min(a, b))$ операций умножения.

НОК (наименьшее общее кратное)

Вычисление наименьшего общего кратного (least common multiplier, lcm) сводится к вычислению \gcd следующим простым утверждением:

$$\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Таким образом, вычисление НОК также можно сделать с помощью алгоритма Евклида, с той же асимптотикой:

```
int lcm (int a, int b) {
    return a / gcd (a, b) * b;
}
```

(здесь выгодно сначала поделить на \gcd , а только потом домножать на b , поскольку это поможет избежать переполнений в некоторых случаях)

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]

Решето Эратосфена

Решето Эратосфена - это алгоритм, позволяющий найти все простые числа в отрезке $[1; N]$ за $O(N \log \log N)$ операций.

Идея проста - запишем ряд чисел $1 \dots N$, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем делящиеся на 3, кроме самого числа 3, затем на 5, затем на 7, 11, и все остальные простые до N .

```
int n; // входные данные
vector<char> prime (n+1, true);
prime[0] = prime[1] = false; // если кто-то будет использовать эти значения
for (int i=2; i<=n; ++i)
    if (prime[i])
        for (int j=i+i; j<=n; j+=i)
            prime[j] = false;
```

В таком виде алгоритм потребляет $O(N)$ памяти и выполняет $O(N \log \log N)$ действий.

Асимптотика

Докажем, что асимптотика алгоритма равна $O(N \log \log N)$.

Итак, для каждого простого $P \leq N$ будет выполняться внутренний цикл, который совершил $\frac{N}{P}$ действий. Следовательно, нам нужно оценить следующую величину:

$$\sum_{\substack{P \leq N, \\ P \text{ prime}}} \frac{N}{P} = N \sum_{\substack{P \leq N, \\ P \text{ prime}}} \frac{1}{P}$$

Вспомним здесь два известных факта: что число простых, меньше либо равных N , приблизительно равно $\frac{N}{\ln N}$, и что K -ое простое число приблизительно равно $K \ln K$ (это следует из первого утверждения). Тогда сумму можно записать таким образом:

$$\sum_{\substack{P \leq N, \\ P \text{ prime}}} \frac{1}{P} \approx \frac{1}{2} + \sum_{K=2}^{\frac{N}{\ln N}} \frac{1}{K \ln K}$$

Здесь мы выделили первое простое из суммы, поскольку при $K = 1$ согласно приближению $K \ln K$ получится 0, что приведёт к делению на нуль.

Теперь оценим такую сумму с помощью интеграла от той же функции по K от 2 до $\frac{N}{\ln N}$ (мы можем производить такое приближение, поскольку, фактически, сумма относится к интегралу как его приближение по формуле прямоугольников):

$$\sum_{K=2}^{\frac{N}{\ln N}} \frac{1}{K \ln K} \approx \int_2^{\frac{N}{\ln N}} \frac{1}{K \ln K} dK$$

Первообразная для подынтегральной функции есть $\ln \ln K$. Выполняя подстановку и убирая члены меньшего порядка, получаем:

$$\int_2^{\frac{N}{\ln N}} \frac{1}{K \ln K} dK = \ln \ln \frac{N}{\ln N} - \ln \ln 2 = \ln(\ln N - \ln \ln N) - \ln \ln 2 \approx \ln \ln N$$

Теперь, возвращаясь к первоначальной сумме, получаем её приближённую оценку:

$$\sum_{\substack{P \leq N, \\ P \text{ prime}}} \frac{N}{P} \approx N \ln \ln N + o(N)$$

что и требовалось доказать.

Более строгое доказательство (и дающее более точную оценку) можно найти в книге Hardy и Wright "An Introduction to the Theory of Numbers" (стр. 349).

Различные оптимизации решета Эратосфена

Самый большой недостаток алгоритма - то, что он "гуляет" по памяти, постоянно выходя за пределы кэш-памяти, из-за чего константа, скрытая в $O(N \log \log N)$, сравнительно велика.

Кроме того, для достаточно больших N узким местом становится объём потребляемой памяти. Можно заменить `vector<char>` на `vector<bool>`, за счёт чего снизив в 8 раз потребление памяти, однако потеря скорости алгоритма будет весьма серьёзной.

Ниже рассмотрены методы, позволяющие как уменьшить число выполняемых операций, так и значительно сократить потребление памяти.

Просеивание простыми до корня

Самый очевидный момент - что для того, чтобы найти все простые до N , достаточно выполнить просеивание только простыми, не превосходящими корня из N .

Таким образом, изменится внешний цикл алгоритма:

```
for (int i=2; i*i<=n; ++i)
```

На асимптотику такая оптимизация не влияет (действительно, повторив приведённое выше доказательство, мы получим оценку $N \ln \ln \sqrt{N} + o(N)$, что, по свойствам логарифма, асимптотически есть то же самое), хотя число операций заметно уменьшится.

Блочное решето

Непосредственно из предыдущего пункта следует, что нет никакой необходимости хранить всё время весь массив $\text{prime}[1 \dots N]$. Для выполнения просеивания достаточно хранить только простые до корня из N , т.е. $\text{prime}[1 \dots \sqrt{N}]$, а остальную часть массива prime строить поблочно, храня в текущий момент времени только один блок.

Пусть S — константа, определяющая размер блока, тогда всего будет $\lceil \frac{N}{S} \rceil$ блоков, k -ый блок ($k = 0 \dots \lfloor \frac{N}{S} \rfloor$) содержит числа в отрезке $[kS; kS + S - 1]$. Будем обрабатывать блоки по очереди, т.е. для каждого k -го блока будем перебирать все простые (от 1 до \sqrt{N}) и выполнять ими просеивание только внутри текущего блока. Аккуратно стоит обрабатывать первый блок — во-первых, простые из $[1; \sqrt{N}]$ не должны удалить сами себя, а во-вторых, числа 0 и 1 должны особо помечаться как не простые. При обработке последнего блока также следует не забывать о том, что последнее нужное число N не обязательно находится в конце блока.

Приведём реализацию блочного решета. Программа считывает число N и находит количество простых от 1 до N :

```
const int SQRT_MAXN = 100000; // корень из максимального значения N
const int S = 10000;
bool nprime[SQRT_MAXN], bl[S];
int primes[SQRT_MAXN], cnt;

int main() {
    int n;
    cin >> n;
    int nsqrt = (int) sqrt (n + .0);
    for (int i=2; i<=nsqrt; ++i)
        if (!nprime[i]) {
            primes[cnt++] = i;
            for (int j=i+i; j<=nsqrt; j+=i)
                nprime[j] = true;
        }

    int result = 0;
    for (int k=0, maxk=n/S; k<=maxk; ++k) {
        memset (bl, 0, sizeof bl);
        int start = k * S;
        for (int i=0; i<cnt; ++i)
            for (int j=max((start+primes[i]-1)/primes[i], 2)*primes[i]-start; j<S; j+=primes[i])
                bl[j] = true;
        if (k == 0)
            bl[0] = bl[1] = true;
        for (int i=0; i<S && start+i<=n; ++i)
            if (!bl[i])
                ++result;
    }
    cout << result;
}
```

Асимптотика блочного решета такая же, как и обычного решета Эратосфена (если, конечно, размер S блоков не будет совсем маленьким), зато объём используемой памяти сократится до $O(\sqrt{N} + S)$ и уменьшится "блуждание" по памяти. Но, с другой стороны, для каждого блока для каждого простого из $[1; \sqrt{N}]$ будет выполняться деление, что будет сильно сказываться при меньших размерах блока. Следовательно, при выборе константы S необходимо соблюсти баланс. Тестирование приведённого выше кода ($N = 10^8$) дало следующие результаты:

S	time(seconds)
10^2	50.2
10^3	6.6
10^4	1.6
3×10^4	1.5
5×10^4	1.6
10^5	1.7
10^6	22.1

Для $N = 10^9$ результаты получились такими:

S	time(seconds)
10^4	27
3×10^4	19
5×10^4	18
10^5	18

Таким образом, оптимальным является выбор S около значения 3×10^4 .

Расширенный алгоритм Евклида

В то время как "обычный" алгоритм Евклида находит наибольший общий делитель двух чисел a и b , расширенный алгоритм Евклида находит помимо НОД также коэффициенты x и y такие, что:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Внести вычисление этих коэффициентов в алгоритм Евклида несложно, достаточно вывести формулы, по которым они меняются при переходе от пары (a, b) к паре (b, r) .

```
int gcd (int a, int b, int & x, int & y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd (b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

Расширенный алгоритм Евклида работает корректно даже для отрицательных чисел.

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]

Определение

Последовательность Фибоначчи определяется следующим образом:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}. \end{aligned}$$

Несколько первых её членов:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, , 55, 89, \dots$$

История

Эти числа ввёл в 1202 г. Леонардо Фибоначчи (Leonardo Fibonacci) (также известный как Леонардо Пизанский (Leonardo Pisano)). Однако именно благодаря математику 19 века Люка (Lucas) название "числа Фибоначчи" стало общепотребительным.

Впрочем, индийские математики упоминали числа этой последовательности ещё раньше: Гопала (Gopala) до 1135 г., Хемачандра (Hemachandra) — в 1150 г.

Числа Фибоначчи в природе

Сам Фибоначчи упоминал эти числа в связи с такой задачей: "Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов за год может произвести на свет эта пара, если известно, что каждый месяц, начиная со второго, каждая пара кроликов производит на свет одну пару?". Решением этой задачи и будут числа последовательности, называемой теперь в его честь. Впрочем, описанная Фибоначчи ситуация — больше игра разума, чем реальная природа.

Индийские математики Гопала и Хемачандра упоминали числа этой последовательности в связи с количеством ритмических рисунков, образующихся в результате чередования долгих и кратких слогов в стихах или сильных и слабых долей в музыке. Число таких рисунков, имеющих в целом n долей, равно F_n .

Числа Фибоначчи появляются и в работе Кеплера 1611 года, который размышлял о числах, встречающихся в природе (работа "О шестиугольных снежинках").

Интересен пример растения — тысячелистника, у которого число стеблей (а значит и цветков) всегда есть число Фибоначчи. Причина этого проста: будучи изначально с единственным стеблем, этот стебель затем делится на два, затем от главного стебля ответвляется ещё один, затем первые два стебля снова разветвляются, затем все стебли, кроме двух последних, разветвляются, и так далее. Таким образом, каждый стебель после своего появления "пропускает" одно разветвление, а затем начинает делиться на каждом уровне разветвлений, что и даёт в результате числа Фибоначчи.

Вообще говоря, у многих цветов (например, лилий) число лепестков является тем или иным числом Фибоначчи.

Также в ботанике известно явление "филлотаксиса". В качестве примера можно привести расположение семечек подсолнуха: если посмотреть сверху на их расположение, то можно увидеть одновременно две серии спиралей (как бы наложенных друг на друга): одни закручены по часовой стрелке, другие — против. Оказывается, что число этих спиралей примерно совпадает с двумя последовательными числами Фибоначчи: 34 и 55 или 89 и 144. Аналогичные факты верны и для некоторых других цветов, а также для сосновых шишек, брокколи, ананасов, и т.д.

Для многих растений (по некоторым данным, для 90% из них) верен и такой интересный факт. Рассмотрим какой-нибудь лист, и будем спускаться от него вниз до тех пор, пока не достигнем листа, расположенного на стебле точно так же (т.е. направленного точно в ту же сторону). Попутно будем считать все листья, попадавшиеся нам (т.е. расположенные по высоте между стартовым листом и конечным), но расположенными по-другому. Нумеруя их, мы будем постепенно совершать витки вокруг стебля (поскольку листья расположены на стебле по спирали). В зависимости от того, совершать витки по часовой стрелке или против, будет получаться разное число витков. Но оказывается, что число витков, совершенных нами по часовой стрелке, число витков, совершенных против часовой стрелки, и число встреченных листьев образуют 3 последовательных числа Фибоначчи.

Впрочем, следует отметить, что есть и растения, для которых приведённые выше подсчёты дадут числа из совсем других последовательностей, поэтому нельзя сказать, что явление филлотаксиса является законом, — это скорее занимательная тенденция.

Свойства

Числа Фибоначчи обладают множеством интересных математических свойств.

Вот лишь некоторые из них:

- Соотношение Кассини:

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$$

- Правило "сложения":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- Из предыдущего равенства при $k = n$ вытекает:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

- Из предыдущего равенства по индукции можно получить, что

$$F_{nk}$$

всегда кратно F_n .

- Верно и обратное к предыдущему утверждение:

если F_m кратно F_n , то m кратно n .

- НОД-равенство:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

- По отношению к алгоритму Евклида числа Фибоначчи обладают тем замечательным свойством, что они являются наихудшими входными данными для этого алгоритма (см. "Теорема Ламе" в [Алгоритме Евклида](#)).

Фибоначчиева система счисления

Теорема Цекендорфа утверждает, что любое натуральное число n можно представить единственным образом в виде суммы чисел Фибоначчи:

$$N = F_{k_1} + F_{k_2} + \dots + F_{k_r}$$

где $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, \dots, k_r \geq 2$ (т.е. в записи нельзя использовать два соседних числа Фибоначчи).

Отсюда следует, что любое число можно однозначно записать в **фибоначчиевой системе счисления**, например:

$$\begin{aligned} 9 &= 8 + 1 = F_6 + F_1 = (10001)_F, \\ 6 &= 5 + 1 = F_5 + F_1 = (1001)_F, \\ 19 &= 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F, \end{aligned}$$

причём ни в каком числе не могут идти две единицы подряд.

Нетрудно получить и правило прибавления единицы к числу в фибоначчиевой системе счисления: если младшая цифра равна 0, то её заменяем на 1, а если равна 1 (т.е. в конце стоит 01), то 01 заменяем на 10. Затем "исправляем" запись, последовательно исправляя везде

011 на 100. В результате за линейное время будет получена запись нового числа.

Перевод числа в фибоначиеву систему счисления осуществляется простым "жадным" алгоритмом: просто перебираем числа Фибоначчи от больших к меньшим и, если некоторое $F_k \leq n$, то F_k входит в запись числа n , и мы отнимаем F_k от n и продолжаем поиск.

Формула для n -го числа Фибоначчи

Формула через радикалы

Существует замечательная формула, называемая по имени французского математика Бине (Binet), хотя она была известна до него Муавру (Moivre):

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Эту формулу легко доказать по индукции, однако вывести её можно с помощью понятия образующих функций или с помощью решения функционального уравнения.

Сразу можно заметить, что второе слагаемое всегда по модулю меньше 1, и более того, очень быстро убывает (экспоненциально). Отсюда следует, что значение первого слагаемого даёт "почти" значение F_n . Это можно записать в строгом виде:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} \right],$$

где квадратные скобки обозначают округление до ближайшего целого.

Впрочем, для практического применения в вычислениях эти формулы мало подходят, потому что требуют очень высокой точности работы с дробными числами.

Матричная формула для чисел Фибоначчи

Нетрудно доказать матричное следующее равенство:

$$(F_{n-2} \ F_{n-1}) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n-1} \ F_n).$$

Но тогда, обозначая

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

получаем:

$$(F_0 \ F_1) \cdot P^n = (F_n \ F_{n+1}).$$

Таким образом, для нахождения n -го числа Фибоначчи надо возвести матрицу P в степень n .

Вспоминая, что возведение матрицы в n -ую степень можно осуществить за $O(\log n)$ (см. [[Бинарное возведение в степень]]), получается, что n -ое число Фибоначчи можно легко вычислить за $O(\log n)$ с использованием только целочисленной арифметики.

Периодичность последовательности Фибоначчи по модулю

Рассмотрим последовательность Фибоначчи F_i по некоторому модулю p . Докажем, что она является периодичной, и причём период начинается с $F_1 = 1$ (т.е. предпериод содержит только F_0).

Докажем это от противного. Рассмотрим $p^2 + 1$ пар чисел Фибоначчи, взятых по модулю p :

$$(F_1, F_2), (F_2, F_3), \dots, (F_{p^2+1}, F_{p^2+2}).$$

Поскольку по модулю p может быть только p^2 различных пар, то среди этой последовательности найдётся как минимум две одинаковые пары. Это уже означает, что последовательность периодична.

Выберем теперь среди всех таких одинаковых пар две одинаковые пары с наименьшими номерами. Пусть это пары с некоторыми номерами (F_a, F_{a+1}) и (F_b, F_{b+1}) . Докажем, что $a = b$. Действительно, в противном случае для них найдутся предыдущие пары (F_{a-1}, F_a) и (F_{b-1}, F_b) , которые, по свойству чисел Фибоначчи, также будут равны друг другу. Однако это противоречит тому, что мы выбрали совпадающие пары с наименьшими номерами, что и требовалось доказать.

Литература

- Роналд Грэхэм, Дональд Кнут, Орен Паташник. **Конкретная математика** [1998]

Обратный элемент в кольце по модулю

Определение

Пусть задан некоторый натуральный модуль m , и рассмотрим кольцо, образуемое этим модулем (т.е. состоящее из чисел от 0 до $m - 1$). Тогда для некоторых элементов этого кольца можно найти **обратный элемент**.

Обратным к числу a по модулю m называется такое число b , что:

$$a \cdot b \equiv 1 \pmod{m},$$

и его нередко обозначают через a^{-1} .

Понятно, что для нуля обратного элемента не существует никогда; для остальных же элементов обратный может как существовать, так и нет. Утверждается, что обратный существует только для тех элементов a , которые **взаимно прости** с модулем m .

Рассмотрим ниже два способа нахождения обратного элемента, работающих при условии, что он существует.

Нахождение с помощью Расширенного алгоритма Евклида

Рассмотрим вспомогательное уравнение (относительно неизвестных x и y):

$$a \cdot x + m \cdot y = 1$$

Это **линейное диофантово уравнение второго порядка**. Как показано в соответствующей статье, из условия $\gcd(a, m) = 1$ следует, что это уравнение имеет решение, которое можно найти с помощью **Расширенного алгоритма Евклида** (отсюда же, кстати говоря, следует, что когда $\gcd(a, m) \neq 1$, решения, а потому и обратного элемента, не существует).

С другой стороны, если мы возьмём от обеих частей уравнения остаток по модулю m , то получим:

$$a \cdot x = 1 \pmod{m}$$

Таким образом, найденное x и будет являться обратным к a .

Таким образом, код будет примерно таким:

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
    cout << "no solution";
else
    cout << x;
```

Асимптотика этого решения получается $O(\log m)$.

Нахождение с помощью Бинарного возведения в степень

Воспользуемся теоремой Эйлера:

$$a^{\phi(m)} \equiv 1 \pmod{m}$$

которая верна как раз для случая взаимно простых a и m .

Кстати говоря, в случае простого модуля m мы получаем ещё более простое утверждение — малую теорему Ферма:

$$a^{m-1} \equiv 1 \pmod{m}.$$

Умножим обе части каждого из уравнений на a^{-1} , получим:

для любого модуля m :

$$a^{\phi(m)-1} \equiv a^{-1} \pmod{m},$$

для простого модуля m :

$$a^{m-2} \equiv a^{-1} \pmod{m}.$$

Таким образом, мы получили формулы для непосредственного вычисления обратного. Для практического применения обычно используют эффективный **алгоритм бинарного возведения в степень**, который в нашем случае позволит произвести возведение в степень за $O(\log m)$.

Этот метод представляется несколько проще описанного в предыдущем пункте, однако он требует знания значения функции Эйлера, что фактически требует факторизации модуля m , что иногда может оказаться весьма сложной задачей.

Если же факторизация числа известна, то тогда и этот метод также работает за асимптотику $O(\log m)$.

Код Грея

Определение

Кодом Грея называется такая система нумерования неотрицательных чисел, когда коды двух соседних чисел отличаются ровно в одном бите.

Например, для чисел длины 3 бита имеем такую последовательность кодов Грея: 000, 001, 011, 010, 110, 111, 101, 100. Например, $G(4) = 6$.

Этот код был изобретен Фрэнком Грэем (Frank Gray) в 1953 году.

Нахождение кода Грея

Рассмотрим биты числа n и биты числа $G(n)$. Заметим, что i -ый бит $G(n)$ равен единице только в том случае, когда i -ый бит n равен единице, а $i + 1$ -ый бит равен нулю, или наоборот (i -ый бит равен нулю, а $i + 1$ -ый равен единице). Таким образом, имеем: $G(n) = n \oplus (n >> 1)$:

```
int g (int n) {
    return n ^ (n >> 1);
}
```

Нахождение обратного кода Грея

Требуется по коду Грея g восстановить исходное число n .

Будем идти от старших битов к младшим (пусть самый младший бит имеет номер 1, а самый старший — k). Получаем такие соотношения между битами n_i числа n и битами g_i числа g :

$$\begin{aligned} n_k &= g_k, \\ n_{k-1} &= g_{k-1} \oplus n_k = g_k \oplus g_{k-1}, \\ n_{k-2} &= g_{k-2} \oplus n_{k-1} = g_k \oplus g_{k-1} \oplus g_{k-2}, \\ n_{k-3} &= g_{k-3} \oplus n_{k-2} = g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3}, \\ \dots \end{aligned}$$

В виде программного кода это проще всего записать так:

```
int rev_g (int g) {
    int n = 0;
    for (; g; g>>=1)
        n ^= g;
    return n;
}
```

Применения

Коды Грея имеют несколько применений в различных областях, иногда достаточно неожиданных:

- n -битный код Грея соответствует гамильтонову циклу по n -мерному кубу.
- В технике, коды Грея используются для **минимизации ошибок** при преобразовании аналоговых сигналов в цифровые (например, в датчиках). В частности, коды Грея и были открыты в связи с этим применением.
- Коды Грея применяются в решении задачи о **Ханойских башнях**.

Пусть n — количество дисков. Начнём с кода Грея длины n , состоящего из одних нулей (т.е. $G(0)$), и будем двигаться по кодам Грея (от $G(i)$ переходить к $G(i+1)$). Поставим в соответствие каждому i -ому биту текущего кода Грея i -ый диск (причём самому младшему биту соответствует наименьший по размеру диск, а самому старшему биту — наибольший). Поскольку на каждом шаге изменяется ровно один бит, то мы можем понимать изменение бита i как перемещение i -го диска. Заметим, что для всех дисков, кроме наименьшего, на каждом шаге имеется ровно один вариант хода (за исключением стартовой и финальной позиций). Для наименьшего диска всегда имеется два варианта хода, однако имеется стратегия выбора хода, всегда приводящая к ответу: если n нечётно, то последовательность перемещений наименьшего диска имеет вид $f \rightarrow t \rightarrow r \rightarrow f \rightarrow t \rightarrow r \rightarrow \dots$ (где f — стартовый стержень, t — финальный стержень, r — оставшийся стержень), а если n чётно, то $f \rightarrow r \rightarrow t \rightarrow f \rightarrow r \rightarrow t \rightarrow \dots$

- Коды Грея также находят применение в теории **генетических алгоритмов**.

Длинная арифметика

Длинная арифметика — это набор программных средств (структуры данных и алгоритмы), которые позволяют работать с числами гораздо больших величин, чем это позволяют стандартные типы данных.

Виды целочисленной длинной арифметики

Вообще говоря, даже только в олимпиадных задачах набор средств достаточно велик, поэтому произведём классификацию различных видов длинной арифметики.

Классическая длинная арифметика

Основная идея заключается в том, что число хранится в виде массива его цифр.

Цифры могут использоваться из той или иной системы счисления, обычно применяются десятичная система счисления и её степени (десять тысяч, миллиард), либо двоичная система счисления.

Операции над числами в этом виде длинной арифметики производятся с помощью "школьных" алгоритмов сложения, вычитания, умножения, деления столбиком. Впрочем, к ним также применимы алгоритмы быстрого умножения: [Быстрое преобразование Фурье](#) и Алгоритм Карацубы.

Здесь описана работа только с неотрицательными длинными числами. Для поддержки отрицательных чисел необходимо ввести и поддерживать дополнительный флаг "отрицательности" числа, либо же работать в дополняющих кодах.

Структура данных

Хранить длинные числа будем в виде вектора чисел `int`, где каждый элемент — это одна цифра числа.

```
typedef vector<int> lnum;
```

Для повышения эффективности будем работать в системе по основанию миллиард, т.е. каждый элемент вектора `lnum` содержит не одну, а сразу 9 цифр:

```
const int base = 1000*1000*1000;
```

Цифры будут храниться в векторе в таком порядке, что сначала идут наименее значимые цифры (т.е. единицы, десятки, сотни, и т.д.).

Кроме того, все операции будут реализованы таким образом, что после выполнения любой из них лидирующие нули (т.е. лишние нули в начале числа) отсутствуют (разумеется, в предположении, что перед каждой операцией лидирующие нули также отсутствуют). Следует отметить, что в представленной реализации для числа ноль корректно поддерживаются сразу два представления: пустой вектор цифр, и вектор цифр, содержащий единственный элемент — ноль.

Вывод

Самое простое — это вывод длинного числа.

Сначала мы просто выводим самый последний элемент вектора (или 0, если вектор пустой), а затем выводим все оставшиеся элементы вектора, дополняя их нулями до 9 символов:

```
printf ("%d", a.empty() ? 0 : a.back());
for (int i=(int)a.size()-2; i>=0; --i)
    printf ("%09d", a[i]);
```

(здесь небольшой тонкий момент: нужно не забыть записать `(int)`, поскольку в противном случае число `a.size()` будет беззнаковым, и если `a.size()<=1`, то при вычитании произойдёт переполнение)

Чтение

Считываем строку в `string`, и затем преобразовываем её в вектор:

```
for (int i=(int)s.length(); i>0; i-=9)
    if (i < 9)
        a.push_back (atoi (s.substr (0, i).c_str ()));
    else
        a.push_back (atoi (s.substr (i-9, 9).c_str ()));
```

Если использовать вместо `string` массив `char`'ов, то код получится ещё компактнее:

```
for (int i=(int)strlen(s); i>0; i-=9) {
    s[i] = 0;
    a.push_back (atoi (i>=9 ? s+i-9 : s));
}
```

Если во входном числе уже могут быть лидирующие нули, то их после чтения можно удалить таким образом:

```
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Сложение

Прибавляет к числу `a` число `b` и сохраняет результат в `a`:

```
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
    if (i == a.size())
        a.push_back (0);
    a[i] += carry + (i < b.size() ? b[i] : 0);
    carry = a[i] >= base;
    if (carry) a[i] -= base;
}
```

Вычитание

Отнимает от числа a число b ($a \geq b$) и сохраняет результат в a:

```
int carry = 0;
for (size_t i=0; i<b.size() || carry; ++i) {
    a[i] -= carry + (i < b.size() ? b[i] : 0);
    carry = a[i] < 0;
    if (carry) a[i] += base;
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Здесь мы после выполнения вычитания удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

Умножение длинного на короткое

Умножает длинное a на короткое b ($b < \text{base}$) и сохраняет результат в a:

```
int carry = 0;
for (size_t i=0; i<a.size() || carry; ++i) {
    if (i == a.size())
        a.push_back(0);
    long long cur = carry + a[i] * 111 * b;
    a[i] = int (cur % base);
    carry = int (cur / base);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Здесь мы после выполнения деления удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

Умножение двух длинных чисел

Умножает a на b и результат сохраняет в c:

```
lnum c (a.size() + b.size());
for (size_t i=0; i<a.size(); ++i)
    for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {
        long long cur = c[i+j] + a[i] * 111 * (j < (int)b.size() ? b[j] : 0) + carry;
        c[i+j] = int (cur % base);
        carry = int (cur / base);
    }
while (c.size() > 1 && c.back() == 0)
    c.pop_back();
```

Деление длинного на короткое

Делит длинное a на короткое b ($b < \text{base}$), частное сохраняет в a, остаток в carry:

```
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
    long long cur = a[i] + carry * 111 * base;
    a[i] = int (cur / b);
    carry = int (cur % b);
}
while (a.size() > 1 && a.back() == 0)
    a.pop_back();
```

Длинная арифметика в факторизованном виде

Здесь идея заключается в том, чтобы хранить не само число, а его факторизацию, т.е. степени каждого входящего в него простого.

Этот метод также весьма прост, и в нём очень легко производить операции умножения и деления, однако невозможно произвести сложение или вычитание. С другой стороны, этот метод значительно экономит память в сравнении с "классическим" подходом, и позволяет производить умножение и деление значительно (асимптотически) быстрее.

Этот метод часто применяется, когда необходимо производить деление по непростому модулю: тогда достаточно хранить число в виде степеней по простым делителям этого модуля, и ещё одного числа — остатка по этому же модулю.

Длинная арифметика по системе простых модулей (Китайская теорема или схема Гарнера)

Суть в том, что выбирается некоторая система модулей (обычно небольших, помещающихся в стандартные типы данных), и число хранится в виде вектора из остатков от его деления на каждый из этих модулей.

Как утверждает Китайская теорема об остатках, этого достаточно, чтобы однозначно хранить любое число в диапазоне от 0 до произведения этих модулей минус один. При этом имеется [Алгоритм Гарнера](#), который позволяет произвести это восстановление из модульного вида в обычную, "классическую", форму числа.

Таким образом, этот метод позволяет экономить память по сравнению с "классической" длинной арифметикой (хотя в некоторых случаях не столь радикально, как метод факторизации). Кроме того, в модульном виде можно очень быстро производить сложения, вычитания, умножения и деления, — все за асимптотически одинаковое время, пропорциональное количеству модулей системы.

Однако всё это даётся ценой весьма трудоёмкого перевода числа из этого модульного вида в обычный вид, для чего, помимо немалых временных затрат, потребуется также реализация "классической" длинной арифметики с умножением.

Виды дробной длинной арифметики

Операции над дробными числами встречаются гораздо реже, и работать с огромными дробными числами значительно сложнее, поэтому в олимпиадах встречается только специфическое подмножество дробной длинной арифметики.

Длинная арифметика в несократимых дробях

Число представляется в виде несократимой дроби $\frac{a}{b}$, где a и b — целые числа. Тогда все операции над дробными числами нетрудно свести к операциям над числителями и знаменателями этих дробей.

Обычно при этом для хранения числителя и знаменателя приходится также использовать длинную арифметику, но, впрочем, самый простой её вид — Классическая длинная арифметика, хотя иногда достаточно применения для них 64-битного встроенного типа.

Выделение позиции плавающей точки в отдельный тип

Иногда в задаче требуется производить расчёты с очень большими либо очень маленькими числами, но при этом не допускать их переполнения. Встроенный 8-байтовый тип `double`, как известно, допускает значения экспоненты в диапазоне $[-308; 308]$, чего иногда может оказаться недостаточно.

Приём, собственно, очень простой — вводится ещё одна целочисленная переменная, отвечающая за экспоненту, а после выполнения каждой операции дробное число "нормализуется", т.е. возвращается в отрезок $[0.1; 1]$, путём увеличения или уменьшения экспоненты.

При перемножении или делении двух таких чисел надо соответственно сложить либо вычесть их экспоненты. При сложении или вычитании перед выполнением этой операции числа следует привести к одной экспоненте, для чего одно из них домножается на 10 в степени разности экспонент.

Наконец, понятно, что не обязательно выбирать 10 в качестве основания экспоненты. Исходя из устройства встроенных типов с плавающей точкой, самым выгодным представляется взять основание равным 2.

Дискретное логарифмирование

Задача дискретного логарифмирования заключается в том, чтобы по данным целым a , b , m решить уравнение:

$$a^x = b \pmod{m}$$

где a и m — взаимно просты (примечание: если они не взаимно просты, это конечно же означает, что решений нет или находить их легко; просто в этом случае описанный ниже алгоритм является некорректным).

Здесь описан алгоритм, известный как "**baby-step-giant-step algorithm**", предложенный **Шэнксом (Shanks)** в 1971 г., работающий за время за $O(\sqrt{m} \log m)$.

Алгоритм

Итак, имеем уравнение:

$$a^x = b \pmod{m}$$

Воспользуемся для его решения методом Meet-in-the-Middle. Для этого преобразуем уравнение. Положим

$$x = np - q$$

где n — это заранее выбранная константа, зависящая от m . Иногда p называют "giant step" (поскольку увеличение его на единицу увеличивает x сразу на n), а в противоположность ему q — "baby step".

Очевидно, что любое x (а мы рассматриваем только $x \in [0; m)$) можно представить в такой форме, причём для этого будет достаточно значений:

$$p \in [1; \left\lceil \frac{m}{n} \right\rceil], \quad q \in [0; n]$$

Тогда уравнение принимает вид:

$$a^{np-q} = b \pmod{m}$$

откуда, пользуясь тем, что a и m взаимно просты, получаем:

$$a^{np} = ba^q \pmod{m}$$

Чтобы решить исходное уравнение, нужно найти соответствующие значения p и q , чтобы значения левой и правой частей совпадли. Иначе говоря, надо решить уравнение:

$$f_1(p) = f_2(q),$$

Эта задача решается с помощью метода Meet-in-the-Middle следующим образом. Посчитаем значения функции f_1 для всех значений аргумента p , и отсортируем эти значения. Затем будем перебирать значение второй переменной q , вычислять вторую функцию f_2 , и искать это значение среди предвычисленных значений первой функции с помощью бинарного поиска.

Асимптотика

Сначала оценим время вычисления каждой из функций $f_1(p)$ и $f_2(q)$. И та, и другая содержит возведение в степень, которое можно выполнять с помощью алгоритма бинарного возведения в степень. Тогда функцию $f_1(p)$ мы можем вычислить за время $O(\log m)$, а $f_2(q)$ — за время $O(\log n)$.

Сам алгоритм в первой части содержит вычисление функции $f_1(p)$ для каждого возможного значения p и дальнейшую сортировку значений, что даёт нам асимптотику:

$$O\left(\left\lceil \frac{m}{n} \right\rceil \left(\log m + \log \left\lceil \frac{m}{n} \right\rceil \right)\right) = O\left(\left\lceil \frac{m}{n} \right\rceil \log \left\lceil \frac{m}{n} \right\rceil\right)$$

Во второй части алгоритма происходит вычисление функции $f_2(q)$ для каждого возможного значения q и бинарный поиск по массиву значений f_1 , что даёт нам асимптотику:

$$O\left(n \left(\log n + \log \left\lceil \frac{m}{n} \right\rceil \right)\right)$$

Теперь, когда мы сложим эти две асимптотики, практически очевидно, что минимум достигается, когда $n \approx m/n$, т.е. для оптимальной работы алгоритма константу n следует выбирать равной:

$$n \approx \sqrt{m}$$

При таком выборе асимптотика алгоритма принимает вид:

$$O\left(\sqrt{m} \log m\right)$$

Примечание. Мы могли бы обменять ролями f_1 и f_2 (какую сортировать и затем выполнять по ней бинарный поиск), однако результат от этого не изменится.

Реализация

Функция `powmod` выполняет бинарное возведение числа a в степень b по модулю m , см. Бинарное возведение в степень.

Функция `solve` производит собственно решение задачи.

```
int powmod (int a, int b, int m) {
    int res = 1;
    while (b > 0)
        if (b & 1)
            res = (res * a) % m, --b;
        else
            a = (a * a) % m, b >>= 1;
    return res;
}

int solve (int a, int b, int m) {
    int msq = (int) sqrt (m + .0) + 1;
    int msq2 = m / msq + (m % msq ? 1 : 0);
    vector < pair<int,int> > vals (msq2);
    for (int i=1; i<=msq2; ++i)
        vals[i-1] = make_pair (powmod (a, i * msq, m), i);
    sort (vals.begin(), vals.end());
    for (int i=0; i<=msq; ++i) {
        int cur = powmod (a, i, m);
        cur = (cur * b) % m;
        vector < pair<int,int> > ::iterator it =
            lower_bound (vals.begin(), vals.end(), make_pair (cur, 0));
        if (it != vals.end() && it->first == cur)
            return it->second * msq - i;
    }
    return -1;
}
```

Диофантовы уравнения с двумя неизвестными: AX+BY=C

Диофантово уравнение с двумя неизвестными имеет вид:

$$A \cdot X + B \cdot Y = C$$

где A, B, C - заданные целые числа, X и Y - неизвестные целые числа.

Ниже рассматриваются несколько классических задач на эти уравнения: нахождение любого решения, получение всех решений, нахождение количества решений и сами решения в определённом отрезке, нахождение решения с наименьшей суммой неизвестных.

1. Нахождение одного решения

Найти одно из решений диофантова уравнения с двумя неизвестными можно с помощью [Расширенного алгоритма Евклида](#).

Расширенный алгоритм Евклида по заданным A и B находит их наибольший общий делитель G, а также такие коэффициенты X_g, Y_g , что:

$$A \cdot X_g + B \cdot Y_g = G$$

Если C делится на G = $\text{gcd}(A,B)$, то диофантово уравнение $A \cdot X + B \cdot Y = C$ **имеет решение**; в противном случае диофантово уравнение **решений не имеет**.

Предположим, что C делится на G, тогда, очевидно, выполняется:

$$A \cdot X_g \cdot (C/G) + B \cdot Y_g \cdot (C/G) = C$$

т.е. **одним из решений** диофантова уравнения являются числа:

$$\begin{aligned} X_0 &= X_g \cdot C / G \\ Y_0 &= Y_g \cdot C / G \end{aligned}$$

2. Получение всех решений

Покажем, как получить все остальные решения (а их бесконечное множество) диофантова уравнения, зная одно из решений (X_0, Y_0) .

Итак, пусть $G = \text{gcd}(A,B)$, а числа X_0, Y_0 удовлетворяют условию:

$$A \cdot X_0 + B \cdot Y_0 = C$$

Тогда заметим, что, прибавив к X_0 число B/G и одновременно отняв A/G от Y_0 , мы не нарушим равенства:

$$\begin{aligned} A \cdot (X_0 + B/G) + B \cdot (Y_0 - A/G) &= \\ = A \cdot X_0 + A \cdot B / G + B \cdot Y_0 - B \cdot A / G &= \\ = A \cdot X_0 + B \cdot Y_0 &= \\ = C \end{aligned}$$

Очевидно, что этот процесс можно повторять сколько угодно, т.е. **все числа** вида:

$$\begin{aligned} X &= X_0 + K \cdot B/G, \\ Y &= Y_0 - K \cdot A/G, \\ K &\in \mathbb{Z} \end{aligned}$$

являются **решениями** диофантова уравнения.

Более того, **только** числа такого вида и являются решениями, т.е. мы описали множество всех решений диофантова уравнения (оно получилось бесконечным, если не наложено дополнительных условий).

3. Нахождение количества решений и сами решения в заданном отрезке

Пусть даны два отрезка $[X_1; X_2]$ и $[Y_1; Y_2]$, и требуется найти количество решений (X, Y) диофантова уравнения, лежащих в данных отрезках соответственно.

Сначала найдём **решение с наименьшим X = MinX** таким, что $\text{MinX} \in [X_1; X_2]$. Для этого сначала найдём любое решение диофантова уравнения (см. пункт 1). Затем получим из него решение с наименьшим $\text{MinX} \in [X_1; X_2]$ - воспользуемся процедурой, описанной в пункте 2, и будем уменьшать/увеличивать MinX , пока оно не попадёт в заданный отрезок, и при этом будет наименьшим. Это можно сделать за $O(1)$ таким образом:

```
int
a, b, c, g, // коэффициенты диофантова уравнения, и g=gcd(a,b)
x0, y0, // одно из решений диофантова уравнения
x1, x2, // заданный отрезок
mx, my; // искомое решение с наименьшим x >= x1
int cnt = (x1 - x0) / b;
if (x0 + cnt * b < x1)
    ++cnt;
mx = x0 + cnt * b;
my = y0 - cnt * b;
```

Здесь предполагается, что $B > 0$ (если $B < 0$, то нужно предварительно изменить знаки A, B и C).

Аналогичным образом найдём решение с наибольшим $X = \text{Max}X \in [X_1; X_2]$.

Теперь, если $\text{Min}X > \text{Max}X$, то количество решений равно нулю.

Если же $\text{Min}X \leq \text{Max}X$, то нам осталось наложить условие на $Y: Y \in [Y_1; Y_2]$.

Пусть $\text{Min}Y$ и $\text{Max}Y$ - это соответствующие пары для $\text{Min}X$ и $\text{Max}X$. Если $\text{Min}Y > \text{Max}Y$, то обменяем их значения. Нам нужно найти пересечение отрезка $[\text{Min}Y; \text{Max}Y]$ и $[Y_1; Y_2]$. Будем увеличивать $\text{Min}Y$ (на $|A/G|$), пока оно не станет больше либо равно Y_1 , и будем уменьшать $\text{Max}Y$ (опять же, на $|A/G|$), пока оно не станет меньше либо равно Y_2 (выполняя это, мы не нарушим ограничения на X , поскольку уменьшая $\text{Min}Y$, мы только увеличиваем соответствующий ему $\text{Min}X$, аналогично и для $\text{Max}Y$).

В конце концов ответом на задачу будет являться величина $(\text{Max}Y - \text{Min}Y) / |A/G|$.

Таким образом, мы можем найти количество решений в заданном отрезке за $O(\log \max(A, B))$.

Аналогичным образом можно получить и сами решения, лежащие в заданном отрезке.

4. Нахождение решения в заданном отрезке с наименьшей суммой $X+Y$

Здесь на X и на Y также должны быть наложены какие-либо ограничения, иначе ответом практически всегда будет минус бесконечность.

Но идея решения такая же, как в предыдущем пункте: сначала находим любое решение диофантового уравнения, затем, применяя описанную в пункте 2 процедуру, придём к наилучшему решению.

Действительно, мы имеем право выполнить следующее преобразование (см. пункт 2):

$$\begin{aligned} X &+= K \frac{B}{G}, \\ Y &-= K \frac{A}{G}, \\ K &\in \mathbb{Z} \end{aligned}$$

Заметим, что при этом сумма $X+Y$ меняется следующим образом:

$$\begin{aligned} (X + K \frac{B}{G}) + (Y - K \frac{A}{G}) &= \\ &= X + Y + K \left(\frac{B}{G} - \frac{A}{G} \right) \end{aligned}$$

Т.е. если $A < B$, то нужно выбрать как можно меньшее значение K , если $A > B$, то нужно выбрать как можно большее значение K .

Если $A = B$, то мы никак не сможем улучшить решение, все решения будут обладать одной и той же суммой.

Таким образом, мы решили эту задачу за асимптотику $O(\log \max(A, B))$.

Линейное модулярное уравнение с одной неизвестной

Линейное модулярное уравнение с одной неизвестной имеет вид:

$$A X = B \pmod{N}$$

где A, B, N - заданные целые числа, X - неизвестное целое число.

Требуется найти искомое значение X , лежащее в отрезке $[0; N-1]$, поскольку на всей числовой прямой, ясно, может существовать бесконечно много решений (которые будут отличаться друг друга на NK , где K - любое целое число).

Решение с помощью Обратного элемента

Рассмотрим сначала более простой случай - когда A и N взаимно просты. Тогда можно найти Обратный элемент к числу A , и, умножив на него обе части уравнения, получить решение:

$$X = B A^{-1} \pmod{N}$$

Теперь рассмотрим случай, когда A и N не взаимно просты. Тогда, очевидно, решение будет существовать не всегда (например, $2X = 1 \pmod{4}$). Пусть $G = \gcd(A, N)$, т.е. их наибольший общий делитель (который в данном случае больше единицы). Тогда, если B делится на G , то, разделив обе части уравнения на это G (т.е. разделив A , B и N на G), мы придём к эквивалентному уравнению, в котором уже A и N будут взаимно просты, а такое уравнение мы уже научились решать.

Если же B не делится на $G = \gcd(A, N)$, то решения не существует, поскольку мы не сможем так разделить обе части уравнения, чтобы A и N были бы не взаимно просты.

Решение с помощью Расширенного алгоритма Евклида

Преобразуем модулярное уравнение к диофантову:

$$AX + NK = B$$

Решать это уравнение можно с помощью [Расширенного алгоритма Евклида](#).

Это уравнение имеет решение тогда и только тогда, когда B делится на $\gcd(A, N)$, т.е. мы пришли к тому же утверждению, что и при решении с помощью Обратного элемента.

Китайская теорема об остатках

Формулировка

В своей современной формулировке теорема звучит так:

Пусть $N = N_1 N_2 \dots N_k$, где N_i - попарно взаимно простые числа.

Поставим в соответствие произвольному числу A ($0 \leq A < N$) кортеж (A_1, \dots, A_k) , где $A_i = A \pmod{N_i}$.

Тогда это соответствие (между числами и кортежами) будет являться взаимно однозначным. И, более того, операции, выполняемые над числом A , можно эквивалентно выполнять над соответствующими элементами кортежами - путём независимого выполнения операций над каждым компонентом.

Т.е., если

$$\begin{aligned} A &\Leftrightarrow (A_1, \dots, A_k) \\ B &\Leftrightarrow (B_1, \dots, B_k) \end{aligned}$$

то справедливо:

$$\begin{aligned} (A+B) \pmod{N} &\Leftrightarrow ((A_1+B_1) \pmod{N_1}, \dots, (A_k+B_k) \pmod{N_k}) \\ (A-B) \pmod{N} &\Leftrightarrow ((A_1-B_1) \pmod{N_1}, \dots, (A_k-B_k) \pmod{N_k}) \\ (A \cdot B) \pmod{N} &\Leftrightarrow ((A_1 \cdot B_1) \pmod{N_1}, \dots, (A_k \cdot B_k) \pmod{N_k}) \end{aligned}$$

В своей первоначальной формулировке эта теорема была доказана китайским математиком Сунь-Цзы приблизительно в 100 г. н.э. А именно, он показал в частном случае эквивалентность решения системы модулярных уравнений и решения одного модулярного уравнения (см. следствие 2 ниже).

Следствие 1

Система модулярных уравнений:

$$\begin{aligned} X &= A_1 \pmod{N_1} \\ &\dots \\ X &= A_k \pmod{N_k} \end{aligned}$$

имеет единственное решение по модулю N .

(как и выше, $N = N_1 \dots N_k$, числа N_i попарно взаимно просты, а набор A_1, \dots, A_k - произвольный набор целых чисел)

Следствие 2

Следствием является связь между системой модулярных уравнений и одним соответствующим модулярным уравнением:

Уравнение:

$$X = A \pmod{N}$$

эквивалентно системе уравнений:

$$\begin{aligned} X &= A \pmod{N_1} \\ &\dots \\ X &= A \pmod{N_k} \end{aligned}$$

(как и выше, предполагается, что $N = N_1 N_2 \dots N_k$, числа N_i попарно взаимно просты, а A - произвольное целое число)

Алгоритм Гарнера

Из китайской теоремы об остатках следует, что можно заменять операции над числами операциями над кортежами. Напомним, каждому числу A ставится в соответствие кортеж (A_1, \dots, A_k) , где:

$$A_i = A \pmod{N_i}$$

Это может найти широкое применение на практике, поскольку мы таким образом можем заменять операции в длинной арифметике операциями с массивом "коротких" чисел. Скажем, массива из 1000 элементов "хватит" на числа примерно с 3000 знаками (если выбрать в качестве N_i первые 1000 простых). Но, разумеется, тогда нужно научиться **восстанавливать** число A по этому кортежу. Из следствия 1 видно, что такое восстановление возможно, и притом единственno (при условии $0 \leq A < (N_1 N_2 \dots N_k)$). **Алгоритм Гарнера** и является алгоритмом, позволяющим выполнить это восстановление, причём достаточно эффективно.

Обозначим через R_{ij} ($i=1..K-1, j=i+1..K$) число, являющееся обратным для N_i по модулю N_j (нахождение обратных элементов в поле описано [здесь](#)). Эти числа нам понадобятся ниже.

Будем искать решение в виде:

$$A = X_1 + X_2 N_1 + X_3 N_1 N_2 + \dots + X_k N_1 N_2 \dots N_{k-1}$$

т.е. в смешанной системе счисления.

Подставим это выражение для в первое уравнение системы, получим:

$$A_1 = X_1$$

Подставим теперь выражение во второе уравнение:

$$\begin{aligned} A_2 &= X_1 + X_2 N_1 \pmod{N_2}, \\ A_2 - X_1 &= X_2 N_1 \pmod{N_2} \end{aligned}$$

откуда, умножая обе части на обратное к N_1 по модулю N_2 , т.е. R_{12} , получим:

$$X_2 = R_{12} (A_2 - X_1) \pmod{N_2}$$

Подставляя в третье уравнение, аналогичным образом получаем:

$$\begin{aligned} A_3 &= X_1 + X_2 N_1 + X_3 N_1 N_2 \pmod{N_3}, \\ A_3 - X_1 &= X_2 N_1 + X_3 N_1 N_2 \pmod{N_3}, \\ R_{13} (A_3 - X_1) &= X_2 + X_3 N_2 \pmod{N_3}, \\ R_{13} (A_3 - X_1) - X_2 &= X_3 N_2 \pmod{N_3}, \\ X_3 &= R_{23} (R_{13} (A_3 - X_1) - X_2) \pmod{N_3} \end{aligned}$$

Уже достаточно ясно видна закономерность, которую можно выразить кодом следующим образом:

```
vector<int> x (k);
for (int i=0; i<k; ++i) {
    x[i] = a[i];
    for (int j=0; j<i; ++j) {
        x[i] = r[j][i] * (x[i] - x[j]) % n[i];
        if (x[i] < 0) x[i] += n[i];
    }
}
```

Итак, мы научились вычислять коэффициенты X_i за время $O(K^2)$, сам же ответ - число A - можно восстановить по формуле:

$$A = X_1 + X_2 N_1 + X_3 N_1 N_2 + \dots + X_k N_1 N_2 \dots N_{k-1}$$

Стоит заметить, что на практике почти всегда вычислять ответ нужно с помощью [Длинной арифметики](#), но при этом сами коэффициенты X_i по-прежнему вычисляются на встроенных типах, а потому весь алгоритм Гарнера является весьма эффективным.

Реализация алгоритма Гарнера

Приведём полную реализацию этого алгоритма (включая длинную арифметику).

Эта реализация поддерживает сложение, вычитание и умножение, причём поддерживает работу с отрицательными числами. Реализовано чтение длинного числа, перевод числа из модулярной системы в длинное число и вывод его.

О поддержке отрицательных чисел следует сказать особо. При чтении длинных чисел знак заносится в глобальную переменную `last_neg` (поскольку в векторе `lnum` нет никакой возможности сохранить его). Далее, сама модулярная схема не предполагает различий между положительными и отрицательными числами. Однако можно заметить, что, если результат по модулю не превосходит половины от произведения всех простых, то при вычислении результата положительные числа получаются меньше этой середины, а отрицательные - больше. Поэтому мы после классического алгоритма Гарнера сравниваем результат с серединой, и если он больше, то выводим минус, и инвертируем результат (т.е. отнимаем его от произведения всех простых, и выводим уже его).

```
typedef vector<int> lnum;
bool last_neg;

const int sz = 351;
int pr[sz];
int rev[sz][sz];

struct number {
    short a[sz];

    number() {
        memset (a, 0, sizeof a);
    }
}
```

```

number (const lnum & num) {
    for (int i=0; i<sz; ++i)
        a[i] = modulus (num, pr[i]);
}

void operator+= (const number & rt) {
    for (int i=0; i<sz; ++i) {
        a[i] += rt.a[i];
        if (a[i] >= pr[i]) a[i] -= pr[i];
    }
}

void operator-= (const number & rt) {
    for (int i=0; i<sz; ++i) {
        a[i] -= rt.a[i];
        if (a[i] < 0) a[i] += pr[i];
    }
}

void operator*= (const number & rt) {
    for (int i=0; i<sz; ++i)
        a[i] = ((int)a[i] * rt.a[i]) % pr[i];
}

void negate() {
    for (int i=0; i<sz; ++i)
        a[i] = a[i] ? pr[i] - a[i] : 0;
}

operator lnum() const {
    lnum res, cur (1,1);
    vector<int> x (sz);
    for (int i=0; i<sz; ++i) {
        x[i] = a[i];
        for (int j=0; j<i; ++j) {
            x[i] = rev[j][i] * (x[i] - x[j]) % pr[i];
            if (x[i] < 0) x[i] += pr[i];
        }
        lnum curadd = cur;
        multiply (curadd, x[i]);
        add (res, curadd);
        multiply (cur, pr[i]);
    }
    return res;
}
};

ostream & operator<< (ostream & stream, const number & a) {
    lnum b = a;

    lnum t (1, 1);
    for (int i=0; i<sz; ++i)
        multiply (t, pr[i]);
    subtract (t, b);
    if (compare (t, b) < 0) {
        printf ("-");
        swap (t, b);
    }

    return stream << b;
}

```

Реализация необходимых процедур длинной арифметики:

```

const int base = 1000*1000*1000;

void add (lnum & a, const lnum & b) {
    for (int i=0, carry=0; i<(int)max(a.size(),b.size()) || carry; ++i) {
        if (i == (int)a.size()) a.push_back (0);
        a[i] += carry + (i < (int)b.size() ? b[i] : 0);
        if (a[i] < base)
            carry = 0;
        else
            carry = 1, a[i] -= base;
    }
}

void subtract (lnum & a, const lnum & b) {
    for (int i=0, carry=0; i<(int)a.size(); ++i) {
        a[i] -= carry + (i < (int)b.size() ? b[i] : 0);
        if (a[i] >= 0)
            carry = 0;
        else
            carry = 1, a[i] += base;
    }
}

```

```

    }
    while (a.size() && !a.back()) a.pop_back();
}

void multiply (lnum & a, int b) {
    for (int i=0, carry=0; i<(int)a.size() || carry; ++i) {
        if (i == (int)a.size()) a.push_back (0);
        long long cur = a[i] * 111 * b + carry;
        carry = int (cur / base);
        a[i] = int (cur - carry * 111 * base);
    }
    while (a.size() && !a.back()) a.pop_back();
}

int modulus (const lnum & a, int b) {
    int carry = 0;
    for (int i=(int)a.size()-1; i>=0; --i)
        carry = int ((a[i] + carry * 111 * base) % b);
    return carry;
}

int compare (const lnum & a, const lnum & b) {
    if (a.size() != b.size())
        return a.size() < b.size() ? -1 : 1;
    for (int i=(int)a.size()-1; i>=0; --i)
        if (a[i] != b[i])
            return a[i] < b[i] ? -1 : 1;
    return 0;
}

istream & operator>> (istream & stream, lnum & a) {
    static char s[100*1000];
    scanf ("%s", s);
    last_neg = false;
    while (s[0] == '-' || s[0] == '+') {
        if (s[0] == '-')
            last_neg = !last_neg;
        memmove (s, s+1, strlen(s));
        if (s[0] == 0)
            scanf ("%s", s);
    }
    size_t len = strlen(s);
    if (len % 9) {
        size_t add = 9 - len%9;
        memmove (s+add, s, len);
        memset (s, '0', add);
        len += add;
    }
    for (size_t i=len; i>0; ) {
        s[i] = 0;
        if (i < 9)
            i = 0;
        else
            i -= 9;
        a.push_back (atoi (s+i));
    }
    while (a.size() && !a.back()) a.pop_back();
    return stream;
}

ostream & operator<< (ostream & stream, const lnum & a) {
    printf ("%d", a.empty() ? 0 : a.back());
    for (int i=(int)a.size()-2; i>=0; --i)
        printf ("%09d", a[i]);
    return stream;
}

```

Нахождение степени делителя факториала

Даны два числа: n и k . Требуется посчитать, с какой степенью делитель k входит в число $n!$, т.е. найти наибольшее x такое, что $n!$ делится на k^x .

Решение для случая простого k

Рассмотрим сначала случай, когда k простое.

Выпишем выражение для факториала в явном виде:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Заметим, что каждый k -ый член этого произведения делится на k , т.е. даёт $+1$ к ответу; количество таких членов равно $\lfloor n/k \rfloor$.

Далее, заметим, что каждый k^2 -ый член этого ряда делится на k^2 , т.е. даёт ещё $+1$ к ответу (учитывая, что k в первой степени уже было учтено до этого); количество таких членов равно $\lfloor n/k^2 \rfloor$.

И так далее, каждый k^i -ый член ряда даёт $+1$ к ответу, а количество таких членов равно $\lfloor n/k^i \rfloor$.

Таким образом, ответ равен величине:

$$\frac{n}{k} + \frac{n}{k^2} + \cdots + \frac{n}{k^i} + \cdots$$

Эта сумма, разумеется, не бесконечная, т.к. только первые примерно $\log_k n$ членов отличны от нуля. Следовательно, асимптотика такого алгоритма равна $O(\log_k n)$.

Реализация:

```
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += n;
    }
    return res;
}
```

Решение для случая составного k

Ту же идею применить здесь непосредственно уже нельзя.

Но мы можем факторизовать k , решить задачу для каждого его простого делителя, а потом выбрать минимум из ответов.

Более формально, пусть k_i — это i -ый делитель числа k , входящий в него в степени p_i . Решим задачу для k_i с помощью вышеописанной формулы за $O(\log n)$; пусть мы получили ответ Ans_i . Тогда ответом для составного k будет минимум из величин Ans_i/p_i .

Учитывая, что факторизация простейшим образом выполняется за $O(\sqrt{n})$, получаем итоговую асимптотику $O(\sqrt{n})$.

Троичная сбалансированная система счисления

Троичная сбалансированная система счисления — это нестандартная позиционная система счисления. Основание системы равно 3, однако она отличается от обычной троичной системы тем, что цифрами являются $-1, 0, 1$. Поскольку использовать -1 для одной цифры очень неудобно, то обычно принимают какое-то специальное обозначение. Условимся здесь обозначать минус единицу буквой z .

Например, число 5 в троичной сбалансированной системе записывается как $1zz$, а число -5 — как $z11$. Троичная сбалансированная система счисления позволяет записывать отрицательные числа без записи отдельного знака "минус". Троичная сбалансированная система позволяет дробные числа (например, $1/3$ записывается как 0.1).

Алгоритм перевода

Научимся переводить числа в троичную сбалансированную систему.

Для этого надо сначала перевести число в троичную систему.

Ясно, что теперь нам надо избавиться от цифр 2, для чего заметим, что $2 = 3 - 1$, т.е. мы можем заменить двойку в текущем разряде на -1 , при этом увеличив следующий (т.е. слева от него в естественной записи) разряд на 1. Если мы будем двигаться по записи справа налево и выполнять вышеописанную операцию (при этом в каких-то разрядах может происходить переполнение больше 3, в таком случае, естественно, "сбрасываем" лишние тройки в старший разряд), то придём к троичной сбалансированной записи. Как нетрудно убедиться, то же самое правило верно и для дробных чисел.

Более изящно вышеописанную процедуру можно описать так. Мы берём число в троичной системе счисления, прибавляем к нему бесконечное число $\dots 11111.11111 \dots$, а затем от каждого разряда результата отнимаем единицу (уже безо всяких переносов).

Зная теперь алгоритм перевода из обычной троичной системы в сбалансированную, легко можно реализовать операции сложения, вычитания и деления — просто сводя их к соответствующим операциям над троичными несбалансированными числами.

Вычисление факториала по модулю

В некоторых случаях необходимо считать по некоторому простому модулю p сложные формулы, которые в том числе могут содержать факториалы. Здесь мы рассмотрим случай, когда модуль p сравнительно мал. Понятно, что эта задача имеет смысл только в том случае, когда факториалы входят и в числитель, и в знаменатель дробей. Действительно, факториал $p!$ и все последующие обращаются в ноль по модулю p , однако в дробях все множители, содержащие p , могут сократиться, и полученное выражение уже будет отлично от нуля по модулю p .

Таким образом, формально задача такая. Требуется вычислить $n!$ по простому модулю p , при этом не учитывая все кратные p множители, входящие в факториал. Научившись эффективно вычислять такой факториал, мы сможем быстро вычислять значение различных комбинаторных формул (например, [Биномиальные коэффициенты](#)).

Алгоритм

Выпишем этот "модифицированный" факториал в явном виде:

$$\begin{aligned} n!_{\%p} &= 1 \cdot 2 \cdot 3 \cdots (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdots (2p-1) \cdot \underbrace{2}_{2p} \cdot (2p+1) \cdots (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdots n = \\ &= 1 \cdot 2 \cdot 3 \cdots (p-2) \cdot (p-1) \cdots \underbrace{1}_{p} \cdot 1 \cdot 2 \cdots (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \cdots (p-1) \cdot \underbrace{1}_{p^2} \cdot 1 \cdot 2 \cdots (n \% p) \pmod{p} \end{aligned}$$

При такой записи видно, что "модифицированный" факториал распадается на несколько блоков длины p (последний блок, возможно, короче), которые все одинаковы, за исключением последнего элемента:

$$n!_{\%p} = \underbrace{1 \cdot 2 \cdots (p-2) \cdot (p-1) \cdot 1}_{1st} \underbrace{1 \cdot 2 \cdots (p-1) \cdot 2}_{2nd} \cdots \underbrace{1 \cdot 2 \cdots (p-1) \cdot 1}_{p-th} \cdots \underbrace{1 \cdot 2 \cdots (n \% p)}_{(n/p)th} \pmod{p}$$

Общую часть блоков посчитать легко — это просто $(p-1)! \bmod p$, которую можно посчитать программно или по теореме Вильсона (Wilson) сразу найти $(p-1)! \bmod p = p-1$. Чтобы перемножить эти общие части всех блоков, надо найденную величину возвести в степень по модулю p , что можно сделать за $O(\log n)$ операций (см. [Бинарное возведение в степень](#); впрочем, можно заметить, что мы фактически возводим минус единицу в какую-то степень, а потому результатом всегда будет либо 1, либо $p-1$, в зависимости от чётности показателя). Значение в последнем, неполном блоке тоже можно посчитать отдельно за $O(p)$. Остались только последние элементы блоков, рассмотрим их внимательнее:

$$n!_{\%p} = \underbrace{\cdots 1 \cdots 2 \cdots 3 \cdots \cdots (p-1) \cdots 1 \cdots 1 \cdots 2 \cdots}_{\text{...}}$$

И мы снова пришли к "модифицированному" факториалу, но уже меньшей размерности (столько, сколько было полных блоков, а их было $\lfloor n/p \rfloor$). Таким образом, вычисление "модифицированного" факториала $n!_{\%p}$ мы свели за $O(p)$ операций к вычислению уже $(n/p)!_{\%p}$.

Раскрывая эту рекуррентную зависимость, мы получаем, что глубина рекурсии будет $O(\log_p n)$, итого **асимптотика алгоритма получается $O(p \log n)$** .

Реализация

Понятно, что при реализации не обязательно использовать рекурсию в явном виде: поскольку рекурсия хвостовая, её легко развернуть в цикл.

```
int factmod (int n, int p) {
    int res = 1;
    while (n > 1) {
        res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
        for (int i=2; i<=n%p; ++i)
            res = (res * i) % p;
        n /= p;
    }
    return res % p;
}
```

Эта реализация работает за $O(p \log n)$.

Перебор всех подмасок данной маски

Дана битовая маска M . Требуется эффективно перебрать все её подмаски, т.е. такие маски S , в которых могут быть включены только те биты, которые были включены в маске M .

Сразу рассмотрим реализацию этого алгоритма, основанную на трюках с битовыми операциями:

```
int s = m;
while (s > 0) {
    ... можно использовать s ...
    s = (s-1) & m;
}
```

или, используя более компактный оператор for:

```
for (int s=m; s; s=(s-1) & m)
    ... можно использовать s ...
```

Единственное исключение для обоих вариантов кода — подмаска, равная нулю, обработана не будет. Её обработку придётся выносить из цикла, или использовать менее изящную конструкцию, например:

```
for (int s=m; ; s=(s-1) & m) {
    ... можно использовать s ...
    if (s==0) break;
}
```

Разберём, почему приведённый выше код действительно находит все подмаски данной маски, причём без повторений, за O (их количества), и в порядке убывания.

Пусть у нас есть текущая подмаска S , и мы хотим перейти к следующей подмаске. Отнимем от маски S единицу, тем самым мы снимем самый правый единичный бит, а все биты правее него поставятся в 1. Затем удалим все "лишние" единичные биты, которые не входят в маску M и потому не могут входить в подмаску. Удаление осуществляется битовой операцией $\& M$. В результате мы "обрежем" маску $S - 1$ до того наибольшего значения, которое она может принять, т.е. до следующей подмаски после S в порядке убывания.

Таким образом, этот алгоритм генерирует все подмаски данной маски в порядке строгого убывания, затрачивая на каждый переход по две элементарные операции.

Особо рассмотрим момент, когда $S = 0$. После выполнения $S - 1$ мы получим маску, в которой все биты включены (битовое представление числа — 1), и после удаления лишних битов операцией $(S - 1) \& M$ получится не что иное, как маска M . Поэтому с маской $S = 0$ следует быть осторожным — если вовремя не остановиться на нулевой маске, то алгоритм может войти в бесконечный цикл.

Перебор всех масок с их подмасками. Оценка 3^N

Во многих задачах, особенно на динамическое программирование по маскам, требуется перебирать все маски, и для каждой маски — все подмаски:

```
for (int m=0; m<(1<<n); ++m)
    for (int s=m; s; s=(s-1) & m)
        ... использование s и m ...
```

Докажем, что внутренний цикл суммарно выполнит $O(3^N)$ итераций.

Доказательство: 1 способ. Рассмотрим i -ый бит. Для него, вообще говоря, есть ровно три варианта: он не входит в маску M (и потому в подмаску S); он входит в M , но не входит в S ; он входит в M и в S . Всего битов N , поэтому всего различных комбинаций будет 3^N , что и требовалось доказать.

Доказательство: 2 способ. Заметим, что если маска M имеет K включённых битов, то она будет иметь 2^K подмасок. Поскольку масок длины N с K включёнными битами есть C_N^K (см. "биномиальные коэффициенты"), то всего комбинаций будет:

$$\sum_{k=0}^N C_N^K 2^K$$

Посчитаем эту сумму. Для этого заметим, что она есть не что иное, как разложение в бином Ньютона выражения $(1 + 2)^N$, т.е. 3^N , что и требовалось доказать.

Первообразные корни

Определение

Первообразным корнем по модулю n (primitive root modulo n) называется такое число g , что все его степени по модулю n пробегают по

всем числам, взаимно простым с n . Математически это формулируется таким образом: если g является первообразным корнем по модулю n , то для любого целого a такого, что $\gcd(a, n) = 1$, найдётся такое целое k , что $g^k \equiv a \pmod{n}$.

В частности, для случая простого n степени первообразного корня пробегают по всем числам от 1 до $n - 1$.

Существование

Первообразный корень по модулю n существует тогда и только тогда, когда n является либо степенью нечётного простого, либо удвоенной степенью простого, а также в случаях $n = 1, n = 2, n = 4$.

Эта теорема (которая была полностью доказана Гауссом в 1801 г.) приводится здесь без доказательства.

Связь с функцией Эйлера

Пусть g — первообразный корень по модулю n . Тогда можно показать, что наименьшее число k , для которого $g^k \equiv 1 \pmod{n}$ (т.е. k — показатель g (multiplicative order)), равно $\phi(n)$. Более того, верно и обратное, и этот факт будет использован нами ниже в алгоритме нахождения первообразного корня.

Кроме того, если по модулю n есть хотя бы один первообразный корень, то всего их $\phi(\phi(n))$ (т.к. циклическая группа с k элементами имеет $\phi(k)$ генераторов).

Алгоритм нахождения первообразного корня

Наивный алгоритм потребует для каждого тестируемого значения g $O(n)$ времени, чтобы вычислить все его степени и проверить, что они все различны. Это слишком медленный алгоритм, ниже мы с помощью нескольких известных теорем из теории чисел получим более быстрый алгоритм.

Выше была приведена теорема о том, что если наименьшее число k , для которого $g^k \equiv 1 \pmod{n}$ (т.е. k — показатель g), равно $\phi(n)$, то g — первообразный корень. Так как для любого числа a выполняется теорема Эйлера ($a^{\phi(n)} \equiv 1 \pmod{n}$), то чтобы проверить, что g первообразный корень, достаточно проверить, что для всех чисел d , меньших $\phi(n)$, выполнялось $g^d \not\equiv 1 \pmod{n}$. Однако пока это слишком медленный алгоритм.

Из теоремы Лагранжа следует, что показатель любого числа по модулю n является делителем $\phi(n)$. Таким образом, достаточно проверить, что для всех собственных делителей $d \mid \phi(n)$ выполняется $g^d \not\equiv 1 \pmod{n}$. Это уже значительно более быстрый алгоритм, однако можно пойти ещё дальше.

Факторизуем число $\phi(n) = p_1^{a_1} \dots p_s^{a_s}$. Докажем, что в предыдущем алгоритме достаточно рассматривать в качестве d лишь числа вида $\frac{\phi(n)}{p_i}$. Действительно, пусть d — произвольный собственный делитель $\phi(n)$. Тогда, очевидно, найдётся такое j , что $d \mid \frac{\phi(n)}{p_j}$, т.е. $d \cdot k = \frac{\phi(n)}{p_j}$. Однако, если бы $g^d \equiv 1 \pmod{n}$, то мы получили бы:

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv (g^d)^k \equiv 1^k \equiv 1 \pmod{n},$$

т.е. всё равно среди чисел вида $\frac{\phi(n)}{p_i}$ нашлось бы то, для которого условие не выполнилось, что требовалось доказать.

Таким образом, алгоритм нахождения первообразного корня такой. Находим $\phi(n)$, факторизуем его. Теперь перебираем все числа $g = 1 \dots n$, и для каждого считаем все величины $g^{\frac{\phi(n)}{p_i}} \pmod{n}$. Если для текущего g все эти числа оказались отличными от 1, то это g и является искомым первообразным корнем.

Время работы алгоритма (считая, что у числа $\phi(n)$ имеется $O(\log \phi(n))$ делителей, а возведение в степень выполняется алгоритмом **Бинарного возведения в степень**, т.е. за $O(\log n)$ равно $O(\text{Ans} \cdot \log \phi(n) \cdot \log n)$ плюс время факторизации числа $\phi(n)$, где Ans — результат, т.е. значение искомого первообразного корня.

Про скорость роста первообразных корней с ростом n известны лишь приблизительные оценки. Известно, что первообразные корни — сравнительно небольшие величины. Одна из известных оценок — оценка Шупа (Shoup), что, в предположении истинности гипотезы Римана, первообразный корень есть $O(\log^6 n)$.

Реализация

Функция `powmod()` выполняет бинарное возведение в степень по модулю, а функция `generator(int p)` — находит первообразный корень по простому модулю p (факторизация числа $\phi(n)$ здесь осуществлена простейшим алгоритмом за $O(\sqrt{\phi(n)})$). Чтобы адаптировать эту функцию для произвольных p , достаточно добавить вычисление [функции Эйлера](#) в переменной `phi`.

```
int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 1ll * a % p), --b;
        else
            a = int (a * 1ll * a % p), b >>= 1;
    return res;
}

int generator (int p) {
    vector<int> fact;
    int phi = p-1, n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
```

```

fact.push_back (i);
while (n % i == 0)
    n /= i;
}
if (n > 1)
    fact.push_back (n);

for (int res=2; res<=p; ++res) {
    bool ok = true;
    for (size_t i=0; i<fact.size() && ok; ++i)
        ok &= powmod (res, phi / fact[i], p) != 1;
    if (ok) return res;
}
return -1;
}

```

Дискретное извлечение корня

Задача дискретного извлечения корня (по аналогии с [задачей дискретного логарифма](#)) звучит следующим образом. По данным n (n — простое), a , k требуется найти все x , удовлетворяющие условию:

$$x^k \equiv a \pmod{n}$$

Алгоритм решения

Решать задачу будем сведением её к задаче дискретного логарифма.

Для этого применим понятие [Первообразного корня по модулю \$n\$](#) . Пусть g — первообразный корень по модулю n (т.к. n — простое, то он существует). Найти его мы можем, как описано в соответствующей статье, за $O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$ плюс время факторизации числа $\phi(n)$.

Отбросим сразу случай, когда $a = 0$ — в этом случае сразу находим ответ $x = 0$.

Поскольку в данном случае (n — простое) любое число от 1 до $n - 1$ представимо в виде степени первообразного корня, то задачу дискретного корня мы можем представить в виде:

$$(g^y)^k \equiv a \pmod{n}$$

где

$$x \equiv g^y \pmod{n}$$

Тривиальным преобразованием получаем:

$$(g^k)^y \equiv a \pmod{n}$$

Здесь искомой величиной является y , таким образом, мы пришли к задаче дискретного логарифмирования в чистом виде. Эту задачу можно решить [алгоритмом baby-step-giant-step Шэнкса](#) за $O(\sqrt{n} \log n)$, т.е. найти одно из решений y_0 этого уравнения (или обнаружить, что это уравнение решений не имеет).

Пусть мы нашли некоторое решение y_0 этого уравнения, тогда одним из решений задачи дискретного корня будет $x_0 = g^{y_0} \pmod{n}$.

Нахождение всех решений, зная одно из них

Чтобы полностью решить поставленную задачу, надо научиться по одному найденному $x_0 = g^{y_0} \pmod{n}$ находить все остальные решения.

Для этого вспомним такой факт, что первообразный корень всегда имеет порядок $\phi(n)$ (см. [статью о первообразном корне](#)), т.е. наименьшей степенью g , дающей единицу, является $\phi(n)$. Поэтому добавление в показатель степени слагаемого с $\phi(n)$ ничего не меняет:

$$x^k \equiv g^{y_0 \cdot k + l \cdot \phi(n)} \equiv a \pmod{n} \quad \forall l \in \mathbb{Z}$$

Отсюда все решения имеют вид:

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod{n} \quad \forall l \in \mathbb{Z}$$

где l выбирается таким образом, чтобы дробь $\frac{l \cdot \phi(n)}{k}$ была целой. Чтобы эта дробь была целой, числитель должен быть кратен

наименьшему общему кратному $\phi(n)$ и k , откуда (вспоминая, что наименьшее общее кратное двух чисел $\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$), получаем:

$$x = g^{y_0 + i \frac{\phi(n)}{\gcd(k, \phi(n))}} \pmod{n} \quad \forall i \in \mathcal{Z}$$

Это окончательная удобная формула, которая даёт общий вид всех решений задачи дискретного корня.

Реализация

Приведём полную реализацию, включающую нахождение первообразного корня, дискретное логарифмирование и нахождение и вывод всех решений.

```

int gcd (int a, int b) {
    return a ? gcd (b%a, a) : b;
}

int powmod (int a, int b, int p) {
    int res = 1;
    while (b)
        if (b & 1)
            res = int (res * 1ll * a % p), --b;
        else
            a = int (a * 1ll * a % p), b >>= 1;
    return res;
}

int generator (int p) {
    vector<int> fact;
    int phi = p-1, n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != 1;
        if (ok) return res;
    }
    return -1;
}

int main() {

    int n, k, a;
    cin >> n >> k >> a;
    if (a == 0) {
        puts ("1\n0");
        return 0;
    }

    int g = generator (n);

    int sq = (int) sqrt (n + .0) + 1;
    vector < pair<int,int> > dec (sq);
    for (int i=1; i<=sq; ++i)
        dec[i-1] = make_pair (powmod (g, int (i * sq * 1ll * k % (n - 1)), n), i);
    sort (dec.begin(), dec.end());
    int any_ans = -1;
    for (int i=0; i<sq; ++i) {
        int my = int (powmod (g, int (i * 1ll * k % (n - 1)), n) * 1ll * a % n);
        vector < pair<int,int> ::iterator it =
            lower_bound (dec.begin(), dec.end(), make_pair (my, 0));
        if (it != dec.end() && it->first == my) {
            any_ans = it->second * sq - i;
            break;
        }
    }
    if (any_ans == -1) {
        puts ("0");
        return 0;
    }

    int delta = (n-1) / gcd (k, n-1);
    vector<int> ans;
    for (int cur=any_ans%delta; cur<n-1; cur+=delta)
        ans.push_back (powmod (g, cur, n));
    sort (ans.begin(), ans.end());
    printf ("%d\n", ans.size());
}

```

```
for (size_t i=0; i<ans.size(); ++i)
    printf ("%d ", ans[i]);
}
```

тест BPSW на простоту чисел

Введение

Алгоритм BPSW - это тест числа на простоту. Этот алгоритм назван по фамилиям его изобретателей: Роберт Бэйли (Ballie), Карл Померанс (Pomerance), Джон Селфридж (Selfridge), Сэмюэль Вагстафф (Wagstaff). Алгоритм был предложен в 1980 году. На сегодняшний день к алгоритму не было найдено ни одного контрпримера, равно как и не было найдено доказательство.

Алгоритм BPSW был проверен на всех числах до 10^{15} . Кроме того, контрпример пытались найти с помощью программы PRIMO (см. [6]), основанной на teste на простоту с помощью эллиптических кривых. Программа, проработав три года, не нашла ни одного контрпримера, на основании чего Мартин предположил, что не существует ни одного BPSW-псевдопростого, меньшего 10^{10000} (псевдопростое число - составное число, на котором алгоритм даёт результат "простое"). В то же время, Карл Померанс в 1984 году представил эвристическое доказательство того, что существует бесконечное множество BPSW-псевдопростых чисел.

Сложность алгоритма BPSW есть $O(\log^3(N))$ битовых операций. Если же сравнивать алгоритм BPSW с другими тестами, например, тестом Миллера-Рабина, то алгоритм BPSW обычно оказывается в 3-7 раз медленнее.

Алгоритм нередко применяется на практике. По-видимому, многие коммерческие математические пакеты, полностью или частично, полагаются на алгоритм BPSW для проверки чисел на простоту.

Краткое описание

Алгоритм имеет несколько различных реализаций, отличающихся друг от друга только деталями. В нашем случае алгоритм имеет вид:

1. Выполнить тест Миллера-Рабина по основанию 2.
 2. Выполнить сильный тест Лукаса-Селфриджа, используя последовательности Лукаса с параметрами Селфриджа.
 3. Вернуть "простое" только в том случае, когда оба теста вернули "простое".
- +0. Кроме того, в начало алгоритма можно добавить проверку на тривиальные делители, скажем, до 1000. Это позволит увеличить скорость работы на составных числах, правда, несколько замедлив алгоритм на простых.

Итак, алгоритм BPSW основывается на следующем:

1. (факт) тест Миллера-Рабина и тест Лукаса-Селфриджа если и ошибаются, то только в одну сторону: некоторые составные числа этими алгоритмами опознаются как простые. В обратную сторону эти алгоритмы не ошибаются никогда.
2. (предположение) тест Миллера-Рабина и тест Лукаса-Селфриджа если и ошибаются, то никогда не ошибаются на одном числе одновременно.

На самом деле, второе предположение вроде бы как и неверно - эвристическое доказательство-опровержение Померанса приведено ниже. Тем не менее, на практике ни одного псевдопростого до сих пор не нашли, поэтому условно можно считать второе предположение верным.

Реализация алгоритмов в данной статье

Все алгоритмы в данной статье будут реализованы на C++. Все программы тестировались только на компиляторе Microsoft C++ 8.0 SP1 (2005), также должны компилироваться на g++.

Алгоритмы реализованы с использованием шаблонов (templates), что позволяет применять их как к встроенным числовым типам, так и собственным классам, реализующим длинную арифметику. [пока длинная арифметика в статью не входит - TODO]

В самой статье будут приведены только самые существенные функции, тексты же вспомогательных функций можно скачать в приложении к статье. Здесь будут приведены только заголовки этих функций вместе с комментариями:

```
/// Модуль 64-битного числа
long long abs (long long n);
unsigned long long abs (unsigned long long n);

/// Возвращает true, если n четное
template <class T>
bool even (const T & n);

/// Делит число на 2
```

```

template <class T>
void bisect (T & n);

 $\text{/// Умножает число на 2}$ 
template <class T>
void redouble (T & n);

 $\text{/// Возвращает true, если n - точный квадрат простого числа}$ 
template <class T>
bool perfect_square (const T & n);

 $\text{/// Вычисляет корень из числа, округляя его вниз}$ 
template <class T>
T sq_root (const T & n);

 $\text{/// Возвращает количество бит в числе}$ 
template <class T>
unsigned bits_in_number (T n);

 $\text{/// Возвращает значение k-го бита числа (биты нумеруются с нуля)}$ 
template <class T>
bool test_bit (const T & n, unsigned k);

 $\text{/// Умножает a *= b (mod n)}$ 
template <class T>
void mulmod (T & a, T b, const T & n);

 $\text{/// Вычисляет a^k (mod n)}$ 
template <class T, class T2>
T powmod (T a, T2 k, const T & n);

 $\text{/// Переводит число n в форму q*2^p}$ 
template <class T>
void transform_num (T n, T & p, T & q);

 $\text{/// Алгоритм Евклида}$ 
template <class T, class T2>
T gcd (const T & a, const T2 & b);

 $\text{/// Вычисляет jacobi(a,b) - символ Якоби}$ 
template <class T>
T jacobi (T a, T b);

 $\text{/// Вычисляет pi(b) первых простых чисел. Возвращает вектор с простыми и в pi - pi(b)}$ 
template <class T, class T2>
const std::vector & get_primes (const T & b, T2 & pi);

 $\text{/// Тривиальная проверка n на простоту, перебираются все делители до m.}$ 
 $\text{/// Результат: 1 - если n точно простое, p - его найденный делитель, 0 - если неизвестно}$ 
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m);

```

Тест Миллера-Рабина

Я не буду заострять внимание на тесте Миллера-Рабина, поскольку он описывается во многих источниках, в том числе и на русском языке (например см. [\[5\]](#)).

Замечу лишь, что скорость его работы есть $O(\log^3(N))$ битовых операций и приведу готовую реализацию этого алгоритма:

```

template <class T, class T2>
bool miller_rabin (T n, T2 b)
{
    // сначала проверяем тривиальные случаи
    if (n == 2)
        return true;
    if (n < 2 || even (n))
        return false;

    // проверяем, что n и b взаимно просты (иначе это приведет к ошибке)
    // если они не взаимно просты, то либо n не просто, либо нужно увеличить b
    if (b < 2)
        b = 2;
    for (T g; (g = gcd (n, b)) != 1; ++b)
        if (n > g)
            return false;

    // разлагаем n-1 = q*2^p
    T n_1 = n;
    --n_1;
    T p, q;
    transform_num (n_1, p, q);

```

```

// вычисляем b^q mod n, если оно равно 1 или n-1, то n простое (или псевдопростое)
T rem = powmod (T(b), q, n);
if (rem == 1 || rem == n_1)
    return true;

// теперь вычисляем b^2q, b^4q, ... , b^{(n-1)/2}
// если какое-либо из них равно n-1, то n простое (или псевдопростое)
for (T i=1; i<p; i++)
{
    mulmod (rem, rem, n);
    if (rem == n_1)
        return true;
}

return false;
}

```

Сильный тест Лукаса-Селфриджа

Сильный тест Лукаса-Селфриджа состоит из двух частей: алгоритма Селфриджа для вычисления некоторого параметра, и сильного алгоритма Лукаса, выполняемого с этим параметром.

Алгоритм Селфриджа

Среди последовательности 5, -7, 9, -11, 13, ... найти первое число D, для которого $J(D, N) = -1$ и $\gcd(D, N) = 1$, где $J(x, y)$ - символ Якоби.

Параметрами Селфриджа будут $P = 1$ и $Q = (1 - D) / 4$.

Следует заметить, что параметр Селфриджа не существует для чисел, которые являются точными квадратами. Действительно, если число является точным квадратом, то перебор D дойдёт до \sqrt{N} , на котором окажется, что $\gcd(D, N) > 1$, т.е. обнаружится, что число N составное.

Кроме того, параметры Селфриджа будут вычислены неправильно для чётных чисел и для единицы; впрочем, проверка этих случаев не составит труда.

Таким образом, **перед началом алгоритма** следует проверить, что число N является нечётным, большим 2, и не является точным квадратом, иначе (при невыполнении хотя бы одного условия) нужно сразу выйти из алгоритма с результатом "составное".

Наконец, заметим, что если D для некоторого числа N окажется слишком большим, то алгоритм с вычислительной точки зрения окажется неприменимым. Хотя на практике такого замечено не было (оказывалось вполне достаточно 4-байтного числа), тем не менее вероятность этого события не следует исключать. Впрочем, например, на отрезке $[1; 10^6]$ $\max(D) = 47$, а на отрезке $[10^{19}; 10^{19}+10^6]$ $\max(D) = 67$. Кроме того, Бэйли и Вагстаф в 1980 году аналитически доказали это наблюдение (см. Ribenboim, 1995/96, стр. 142).

Сильный алгоритм Лукаса

Параметрами алгоритма Лукаса являются числа **D, P и Q** такие, что $D = P^2 - 4*Q \neq 0$, и $P > 0$.

(нетрудно заметить, что параметры, вычисленные по алгоритму Селфриджа, удовлетворяют этим условиям)

Последовательности Лукаса - это последовательности U_k и V_k , определяемые следующим образом:

```

U0 = 0
U1 = 1
Uk = P * Uk-1 - Q * Uk-2
V0 = 2
V1 = P
Vk = P * Vk-1 - Q * Vk-2

```

Далее, пусть $M = N - J(D, N)$.

Если N простое, и $\gcd(N, Q) = 1$, то имеем:

$U_M = 0 \pmod{N}$

В частности, когда параметры D, P, Q вычислены алгоритмом Селфриджа, имеем:

$U_{N+1} = 0 \pmod{N}$

Обратное, вообще говоря, неверно. Тем не менее, псевдопростых чисел при данном алгоритме оказывается не очень много, на чём, собственно, и основывается алгоритм Лукаса.

Итак, алгоритм Лукаса заключается в вычислении U_M и сравнении его с нулём.

Далее, необходимо найти какой-то способ ускорения вычисления U_k , иначе, понятно, никакого практического смысла в этом алгоритма не было бы.

Имеем:

```

Uk = (ak - bk) / (a - b),
Vk = ak + bk,

```

где a и b - различные корни квадратного уравнения $x^2 - P x + Q = 0$.

Теперь следующие равенства можно доказать элементарно:

$$\begin{aligned} U_{2k} &= U_k V_k \pmod{N} \\ V_{2k} &= V_k^2 - 2 Q^k \pmod{N} \end{aligned}$$

Теперь, если представить $M = E 2^T$, где E - нечётное число, то легко получить:

$$U_M = U_E V_E V_{2E} V_{4E} \dots V_{2^{T-2}E} V_{2^{T-1}E} = 0 \pmod{N},$$

и хотя бы один из множителей равен нулю по модулю N .

Понятно, что **достаточно вычислить U_E и V_E** , а все последующие множители $V_{2E} V_{4E} \dots V_{2^{T-2}E} V_{2^{T-1}E}$ можно **получить уже из них**.

Таким образом, осталось научиться быстро вычислять U_E и V_E для нечётного E .

Сначала рассмотрим следующие формулы для сложения членов последовательностей Лукаса:

$$\begin{aligned} U_{i+j} &= (U_i V_j + U_j V_i) / 2 \pmod{N} \\ V_{i+j} &= (V_i V_j + D U_i U_j) / 2 \pmod{N} \end{aligned}$$

Следует обратить внимание, что деление выполняется в поле (\pmod{N}).

Формулы эти доказываются очень просто, и здесь их доказательство опущено.

Теперь, обладая формулами для сложения и для удвоения членов последовательностей Лукаса, понятен и способ ускорения вычисления U_E и V_E .

Действительно, рассмотрим двоичную запись числа E . Положим сначала результат - U_E и V_E - равными, соответственно, U_1 и V_1 . Пройдёмся по всем битам числа E от более младших к более старшим, пропустив только самый первый бит (начальный член последовательности). Для каждого i -го бита будем вычислять U_{2^i} и V_{2^i} из предыдущих членов с помощью формул удвоения. Кроме того, если текущий i -ый бит равен единице, то к ответу будем прибавлять текущие U_{2^i} и V_{2^i} с помощью формул сложения. По окончании алгоритма, выполняющегося за $O(\log(E))$, мы **получим искомые U_E и V_E** .

Если U_E или V_E оказались равными нулю (\pmod{N}), то число N простое (или псевдопростое). Если они оба отличны от нуля, то вычисляем V_{2E} , V_{4E} , ..., $V_{2^{T-2}E}$, $V_{2^{T-1}E}$. Если хотя бы один из них сравним с нулём по модулю N , то число N простое (или псевдопростое). Иначе число N составное.

Обсуждение алгоритма Селфриджа

Теперь, когда мы рассмотрели алгоритм Лукаса, можно более подробно остановиться на его параметрах D, P, Q , одним из способов получения которых является алгоритм Селфриджа.

Напомним базовые требования к параметрам:

$$\begin{aligned} P &> 0, \\ D &= P^2 - 4*Q ? 0. \end{aligned}$$

Теперь продолжим изучение этих параметров.

D не должно быть точным квадратом (\pmod{N}).

Действительно, иначе получим:

$D = b^2$, отсюда $J(D, N) = 1$, $P = b + 2$, $Q = b + 1$, отсюда $U_{n-1} = (Q^{n-1} - 1) / (Q - 1)$.

Т.е. если D - точный квадрат, то алгоритм Лукаса становится практически обычным вероятностным тестом.

Один из лучших способов избежать подобного - **потребовать, чтобы $J(D, N) = -1$** .

Например, можно выбрать первое число D из последовательности 5, -7, 9, -11, 13, ..., для которого $J(D, N) = -1$. Также пусть $P = 1$. Тогда $Q = (1 - D) / 4$. Этот способ был предложен Селфриджем.

Впрочем, имеются и другие способы выбора D . Можно выбирать его из последовательности 5, 9, 13, 17, 21, ... Так же пусть P - наименьшее нечётное, приводящее \sqrt{D} . Тогда $Q = (P^2 - D) / 4$.

Понятно, что от выбора конкретного способа вычисления параметров Лукаса зависит и его результат - псевдопростые могут отличаться при различных способах выбора параметра. Как показала практика, алгоритм, предложенный Селфриджем, оказался очень удачным: все псевдопростые Лукаса-Селфриджа не являются псевдопростыми Миллера-Рабина, по крайней мере, ни одного контрпримера найдено не было.

Реализация сильного алгоритма Лукаса-Селфриджа

Теперь осталось только реализовать алгоритм:

```
template <class T, class T2>
bool lucas_selfridge (const T & n, T2 unused)
{
    // сначала проверяем тривиальные случаи
    if (n == 2)
        return true;
    if (n < 2 || even (n))
        return false;

    // проверяем, что n не является точным квадратом, иначе алгоритм даст ошибку
    if (perfect_square (n))
```

```

return false;

// алгоритм Селфриджа: находим первое число d такое, что:
// jacobi(d,n)=-1 и оно принадлежит ряду { 5,-7,9,-11,13,... }
T2 dd;
for (T2 d_abs = 5, d_sign = 1; ; d_sign = -d_sign, ++++d_abs)
{
    dd = d_abs * d_sign;
    T g = gcd (n, d_abs);
    if (1 < g && g < n)
        // нашли делитель - d_abs
        return false;
    if (jacobi (T(dd), n) == -1)
        break;
}

// параметры Селфриджа
T2
p = 1,
q = (p*p - dd) / 4;

// разлагаем n+1 = d*2^s
T n_1 = n;
++n_1;
T s, d;
transform_num (n_1, s, d);

// алгоритм Лукаса
T
u = 1,
v = p,
u2m = 1,
v2m = p,
qm = q,
qm2 = q*2,
qkd = q;
for (unsigned bit = 1, bits = bits_in_number(d); bit < bits; bit++)
{
    mulmod (u2m, v2m, n);
    mulmod (v2m, v2m, n);
    while (v2m < qm2)
        v2m += n;
    v2m -= qm2;
    mulmod (qm, qm, n);
    qm2 = qm;
    redouble (qm2);
    if (test_bit (d, bit))
    {
        T t1, t2;
        t1 = u2m;
        mulmod (t1, v, n);
        t2 = v2m;
        mulmod (t2, u, n);

        T t3, t4;
        t3 = v2m;
        mulmod (t3, v, n);
        t4 = u2m;
        mulmod (t4, u, n);
        mulmod (t4, (T)dd, n);

        u = t1 + t2;
        if (!even (u))
            u += n;
        bisect (u);
        u %= n;

        v = t3 + t4;
        if (!even (v))
            v += n;
        bisect (v);
        v %= n;
        mulmod (qkd, qm, n);
    }
}

// точно простое (или псевдо-простое)
if (u == 0 || v == 0)
    return true;

// довычисляем оставшиеся члены
T qkd2 = qkd;
redouble (qkd2);
for (T2 r = 1; r < s; ++r)
{
}

```

```

mulmod (v, v, n);
v -= qkd2;
if (v < 0) v += n;
if (v < 0) v += n;
if (v >= n) v -= n;
if (v >= n) v -= n;
if (v == 0)
    return true;
if (r < s-1)
{
    mulmod (qkd, qkd, n);
    qkd2 = qkd;
    redouble (qkd2);
}
}

return false;
}

```

Код BPSW

Теперь осталось просто скомбинировать результаты всех 3 тестов: проверка на небольшие тривиальные делители, тест Миллера-Рабина, сильный тест Лукаса-Селфриджа.

```

template <class T>
bool baillie_pomerance_selfridge_wagstaff (T n)
{

// сначала проверяем на тривиальные делители - например, до 29
int div = prime_div_trivial (n, 29);
if (div == 1)
    return true;
if (div > 1)
    return false;

// тест Миллера-Рабина по основанию 2
if (!miller_rabin (n, 2))
    return false;

// сильный тест Лукаса-Селфриджа
return lucas_selfridge (n, 0);
}

```

[Отсюда](#) можно скачать программу (исходник + exe), содержащую полную реализацию теста BPSW. [77 КБ]

Краткая реализация

Длину кода можно значительно уменьшить в ущерб универсальности, отказавшись от шаблонов и различных вспомогательных функций.

```

const int trivial_limit = 50;
int p[1000];

int gcd (int a, int b) {
    return a ? gcd (b%a, a) : b;
}

int powmod (int a, int b, int m) {
    int res = 1;
    while (b)
        if (b & 1)
            res = (res * 111 * a) % m, --b;
        else
            a = (a * 111 * a) % m, b >>= 1;
    return res;
}

bool miller_rabin (int n) {
    int b = 2;
    for (int g; (g = gcd (n, b)) != 1; ++b)
        if (n > g)
            return false;
    int p=0, q=n-1;
    while ((q & 1) == 0)
        ++p, q >>= 1;
    int rem = powmod (b, q, n);
    if (rem == 1 || rem == n-1)
        return true;
    for (int i=1; i<p; ++i) {

```

```

rem = (rem * 111 * rem) % n;
if (rem == n-1) return true;
}
return false;
}

int jacobi (int a, int b)
{
if (a == 0) return 0;
if (a == 1) return 1;
if (a < 0)
if ((b & 2) == 0)
return jacobi (-a, b);
else
return - jacobi (-a, b);
int al=a, e=0;
while ((al & 1) == 0)
al >>= 1, ++e;
int s;
if ((e & 1) == 0 || (b & 7) == 1 || (b & 7) == 7)
s = 1;
else
s = -1;
if ((b & 3) == 3 && (al & 3) == 3)
s = -s;
if (al == 1)
return s;
return s * jacobi (b % al, al);
}

bool bpsw (int n) {
if ((int)sqrt(n+0.0) * (int)sqrt(n+0.0) == n) return false;
int dd=5;
for (;;) {
int g = gcd (n, abs(dd));
if (1<g && g<n) return false;
if (jacobi (dd, n) == -1) break;
dd = dd<0 ? -dd+2 : -dd-2;
}
int p=1, q=(p*p-dd)/4;
int d=n+1, s=0;
while ((d & 1) == 0)
++s, d>>=1;
long long u=1, v=p, u2m=1, v2m=p, qm=q, qm2=q*2, qkd=q;
for (int mask=2; mask<=d; mask<<=1) {
u2m = (u2m * v2m) % n;
v2m = (v2m * v2m) % n;
while (v2m < qm2) v2m += n;
v2m -= qm2;
qm = (qm * qm) % n;
qm2 = qm * 2;
if (d & mask) {
long long t1 = (u2m * v) % n, t2 = (v2m * u) % n,
t3 = (v2m * v) % n, t4 = (((u2m * u) % n) * dd) % n;
u = t1 + t2;
if (u & 1) u += n;
u = (u >> 1) % n;
v = t3 + t4;
if (v & 1) v += n;
v = (v >> 1) % n;
qkd = (qkd * qm) % n;
}
}
if (u==0 || v==0) return true;
long long qkd2 = qkd*2;
for (int r=1; r<s; ++r) {
v = (v * v) % n - qkd2;
if (v < 0) v += n;
if (v < 0) v += n;
if (v >= n) v -= n;
if (v >= n) v -= n;
if (v == 0) return true;
if (r < s-1) {
qkd = (qkd * 111 * qkd) % n;
qkd2 = qkd * 2;
}
}
return false;
}

bool prime (int n) { // эту функцию нужно вызывать для проверки на простоту
for (int i=0; i<trivial_limit && p[i]<n; ++i)
if (n % p[i] == 0)
return false;
if (p[trivial_limit-1]*p[trivial_limit-1] >= n)

```

```

    return true;
    if (!miller_rabin (n))
        return false;
    return bpsw (n);
}

void prime_init() { // вызвать до первого вызова prime() !
    for (int i=2, j=0; j<trivial_limit; ++i) {
        bool pr = true;
        for (int k=2; k*k<=i; ++k)
            if (i % k == 0)
                pr = false;
        if (pr)
            p[j++] = i;
    }
}

```

Эвристическое доказательство-опровержение Померанса

Померанс в 1984 году предложил следующее эвристическое доказательство.

Утверждение: Количество BPSW-псевдопростых от 1 до X больше X^{1-a} для любого $a > 0$.

Доказательство.

Пусть $k > 4$ - произвольное, но фиксированное число. Пусть T - некоторое большое число.

Пусть $P_k(T)$ - множество таких простых p в интервале $[T; T^k]$, для которых:

- (1) $p \equiv 3 \pmod{8}$, $J(5,p) = -1$
- (2) число $(p-1)/2$ не является точным квадратом
- (3) число $(p-1)/2$ составлено исключительно из простых $q < T$
- (4) число $(p-1)/2$ составлено исключительно из таких простых q , что $q \equiv 1 \pmod{4}$
- (5) число $(p+1)/4$ не является точным квадратом
- (6) число $(p+1)/4$ составлено исключительно из простых $d < T$
- (7) число $(p+1)/4$ составлено исключительно из таких простых d , что $d \equiv 3 \pmod{4}$

Понятно, что приблизительно $1/8$ всех простых в отрезке $[T; T^k]$ удовлетворяет условию (1). Также можно показать, что условия (2) и (5) сохраняют некоторую часть чисел. Эвристически, условия (3) и (6) также позволяют нам оставить некоторую часть чисел из отрезка $(T; T^k)$. Наконец, событие (4) обладает вероятностью $(c(\log T)^{-1/2})$, так же как и событие (7). Таким образом, мощность множества $P_k(T)$ приблизительно равна при $T \rightarrow \infty$

□
где c - некоторая положительная константа, зависящая от выбора k .

Теперь мы **можем построить число n** , не являющееся точным квадратом, составленное из l простых из $P_k(T)$, где l нечетно и меньше $T^2 / \log(T^k)$. Количество способов выбрать такое число n есть примерно

□
для большого T и фиксированного k . Кроме того, каждое такое число n меньше e^{T^2} .

Обозначим через Q_1 произведение простых $q < T$, для которых $q \equiv 1 \pmod{4}$, а через Q_3 - произведение простых $q < T$, для которых $q \equiv 3 \pmod{4}$. Тогда $\gcd(Q_1, Q_3) = 1$ и $Q_1 Q_3 \leq e^T$. Таким образом, количество способов выбрать n с **дополнительными условиями**

```
n = 1 (mod Q1), n = -1 (mod Q3)
```

должно быть, эвристически, как минимум

$$e^{T^2} (1 - 3/k) / e^{2T} > e^{T^2} (1 - 4/k)$$

для большого T .

Но **каждое такое n - это контрпример к тесту BPSW**. Действительно, n будет числом Кармайкла (т.е. числом, на котором тест Миллера-Рабина будет ошибаться при любом основании), поэтому оно автоматически будет псевдопростым по основанию 2. Поскольку $n \equiv 3 \pmod{8}$ и каждое $p | n$ равно $3 \pmod{8}$, очевидно, что n также будет сильным псевдопростым по основанию 2. Поскольку $J(5,n) = -1$, то каждое простое $p | n$ удовлетворяет $J(5,p) = -1$, и так как $p+1 | n+1$ для любого простого $p | n$, отсюда следует, что n - псевдопростое Лукаса для любого теста Лукаса с дискриминантой 5.

Таким образом, мы показали, что для любого фиксированного k и всех больших T , будет как минимум $e^{T^2} (1 - 4/k)$ контрпримеров к тесту BPSW среди чисел, меньших e^{T^2} . Теперь, если мы положим $x = e^{T^2}$, будет как минимум $x^{1 - 4/k}$ контрпримеров, меньших x . Поскольку k - случайное число, то наше доказательство означает, что **количество контрпримеров, меньших x , есть число, большее x^{1-a} для любого $a > 0$** .

Практические испытания теста BPSW

В этом разделе будут рассмотрены результаты, полученные мной в результате тестирования моей реализации теста BPSW. Все испытания проводились на встроенном типе - 64-битном числе long long. Длинная арифметика не тестировалась.

Тестирования проводились на компьютере с процессором Celeron 1.3 GHz.

Все времена даны в **микросекундах** (10^{-6} сек).

Среднее время работы на отрезке чисел в зависимости от предела тривиального перебора

Имеется в виду параметр, передаваемый функции prime_div_trivial(), который в коде выше равен 29.

[Скачать](#) тестовую программу (исходник и exe-файл). [83 KB]

Если запускать тест **на всех нечетных числах** из отрезка, то результаты получаются такими:

начало отрезка	конец отрезка	предел > перебора >	0	10^2	10^3	10^4	10^5
1	10^5		8.1	4.5	0.7	0.7	0.9
10^6	10^6+10^5		12.8	6.8	7.0	1.6	1.6
10^9	10^9+10^5		28.4	12.6	12.1	17.0	17.1
10^{12}	$10^{12}+10^5$		41.5	16.5	15.3	19.4	54.4
10^{15}	$10^{15}+10^5$		66.7	24.4	21.1	24.8	58.9

Если запускать тест **только на простых числах** из отрезка, то скорость работы такова:

начало отрезка	конец отрезка	предел > перебора >	0	10^2	10^3	10^4	10^5
1	10^5		42.9	40.8	3.1	4.2	4.2
10^6	10^6+10^5		75.0	76.4	88.8	13.9	15.2
10^9	10^9+10^5		186.5	188.5	201.0	294.3	283.9
10^{12}	$10^{12}+10^5$		288.3	288.3	302.2	387.9	1069.5
10^{15}	$10^{15}+10^5$		485.6	489.1	496.3	585.4	1267.4

Таким образом, оптимально выбирать **предел тривиального перебора равным 100 или 1000**.

Для всех следующих тестов я выбрал предел 1000.

Среднее время работы на отрезке чисел

Теперь, когда мы выбрали предел тривиального перебора, можно более точно протестировать скорость работы на различных отрезках.

[Скачать](#) тестовую программу (исходник и exe-файл). [83 KB]

начало отрезка	конец отрезка	время работы на нечетных числах	время работы на простых числах
1	10^5	1.2	4.2
10^6	10^6+10^5	13.8	88.8
10^7	10^7+10^5	16.8	115.5
10^8	10^8+10^5	21.2	164.8
10^9	10^9+10^5	24.0	201.0
10^{10}	$10^{10}+10^5$	25.2	225.5
10^{11}	$10^{11}+10^5$	28.4	266.5
10^{12}	$10^{12}+10^5$	30.4	302.2
10^{13}	$10^{13}+10^5$	33.0	352.2
10^{14}	$10^{14}+10^5$	37.5	424.3
10^{15}	$10^{15}+10^5$	42.3	499.8
10^{16}	$10^{16}+10^5$	46.5	553.6
10^{17}	$10^{17}+10^5$	48.9	621.1

Или, в виде графика, приблизительное время работы теста BPSW на одном числе:

□

То есть мы получили, что на практике, на небольших числах (до 10^{17}), **алгоритм работает за $O(\log N)$** . Это объясняется тем, что для встроенного типа int64 операция деления выполняется за $O(1)$, т.е. сложность деления не зависит от количества битов в числе.

Если же применить тест BPSW к длинной арифметике, то ожидается, что он будет работать как раз за $O(\log^3(N))$. [TODO]

Приложение. Все программы

Литература

Использованная мной литература, полностью доступная в Интернете:

1. Robert Baillie; Samuel S. Wagstaff
Lucas pseudoprimes
Math. Comp. 35 (1980) 1391-1417
mpqs.free.fr/LucasPseudoprimes.pdf
2. Daniel J. Bernstein
Distinguishing prime numbers from composite numbers: the state of the art in 2004
Math. Comp. (2004)
cr.yp.to/primetests/prime2004-20041223.pdf
3. Richard P. Brent
Primality Testing and Integer Factorisation
The Role of Mathematics in Science (1990)
www.maths.anu.edu.au/~brent/pd/rpb120.pdf
4. H. Cohen; H. W. Lenstra
Primality Testing and Jacobi Sums
Amsterdam (1984)
www.openaccess.leidenuniv.nl/bitstream/1887/2136/1/346_065.pdf
5. Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest
Introduction to Algorithms
[без ссылки]
The MIT Press (2001)
6. M. Martin
PRIMO - Primality Proving
www.ellipsa.net
7. F. Morain
Elliptic curves and primality proving
Math. Comp. 61(203) (1993)
citeseer.ist.psu.edu/rd/43190198%2C72628%2C1%2C0.25%2CDownload/ftp%3AqSqqSqftp.inria.frqSqINRIaqSqupublicationqSqupubli-ps-gzqSqRqSqRR-1256.ps.gz
8. Carl Pomerance
Are there counter-examples to the Baillie-PSW primality test?
Math. Comp. (1984)
www.pseudoprime.com/dopo.pdf
9. Eric W. Weisstein
Baillie-PSW primality test
MathWorld (2005)
mathworld.wolfram.com/Baillie-PSWPrimalityTest.html
10. Eric W. Weisstein
Strong Lucas pseudoprime
MathWorld (2005)
mathworld.wolfram.com/StrongLucasPseudoprime.html
11. Paulo Ribenboim
The Book of Prime Number Records
Springer-Verlag (1989)
[без ссылки]

Список других рекомендуемых книг, которых мне не удалось найти в Интернете:

12. Zhiyu Mo; James P. Jones
A new primality test using Lucas sequences
Preprint (1997)
13. Hans Riesel
Prime numbers and computer methods for factorization
Boston: Birkhauser (1994)

Эффективные алгоритмы факторизации

Здесь приведены реализации нескольких алгоритмов факторизации, каждый из которых по отдельности может работать как быстро, так и очень медленно, но в сумме они дают весьма быстрый метод.

Описания этих методов не приводятся, тем более что они достаточно хорошо описаны в Интернете.

Метод Полларда p-1

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_p_1 (T n)
{
    // параметры алгоритма, существенно влияют на производительность и качество поиска
    const T b = 13;
    const T q[] = { 2, 3, 5, 7, 11, 13 };

    // несколько попыток алгоритма
    T a = 5 % n;
    for (int j=0; j<10; j++)
    {

        // ищем такое a, которое взаимно просто с n
        while (gcd (a, n) != 1)
        {
            mulmod (a, a, n);
            a += 3;
            a %= n;
        }

        // вычисляем a^M
        for (size_t i = 0; i < sizeof q / sizeof q[0]; i++)
        {
            T qq = q[i];
            T e = (T) floor (log ((double)b) / log ((double)qq));
            T aa = powmod (a, powmod (qq, e, n), n);
            if (aa == 0)
                continue;

            // проверяем, не найден ли ответ
            T g = gcd (aa-1, n);
            if (1 < g && g < n)
                return g;
        }
    }

    // если ничего не нашли
    return 1;
}
```

Метод Полларда "Ро"

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```
template <class T>
T pollard_rho (T n, unsigned iterations_count = 100000)
{
    T
    b0 = rand() % n,
    b1 = b0,
    g;
    mulmod (b1, b1, n);
    if (++b1 == n)
        b1 = 0;
    g = gcd (abs (b1 - b0), n);
    for (unsigned count=0; count<iterations_count && (g == 1 || g == n); count++)
    {
        mulmod (b0, b0, n);
        if (++b0 == n)
            b0 = 0;
        mulmod (b1, b1, n);
        ++b1;
        mulmod (b1, b1, n);
```

```

    if (++b1 == n)
        b1 = 0;
    g = gcd (abs (b1 - b0), n);
}
return g;
}

```

Метод Бента (модификация метода Полларда "Ро")

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```

template <class T>
T pollard_bent (T n, unsigned iterations_count = 19)
{
    T
    b0 = rand() % n,
    b1 = (b0*b0 + 2) % n,
    a = b1;
    for (unsigned iteration=0, series_len=1; iteration<iterations_count; iteration++, series_len*=2)
    {
        T g = gcd (b1-b0, n);
        for (unsigned len=0; len<series_len && (g==1 && g==n); len++)
        {
            b1 = (b1*b1 + 2) % n;
            g = gcd (abs(b1-b0), n);
        }
        b0 = a;
        a = b1;
        if (g != 1 && g != n)
            return g;
    }
    return 1;
}

```

Метод Полларда Монте-Карло

Вероятностный тест, быстро даёт ответ далеко не для всех чисел.

Возвращает либо найденный делитель, либо 1, если делитель не был найден.

```

template <class T>
T pollard_monte_carlo (T n, unsigned m = 100)
{
    T b = rand() % (m-2) + 2;

    static std::vector<T> primes;
    static T m_max;
    if (primes.empty())
        primes.push_back (3);
    if (m_max < m)
    {
        m_max = m;
        for (T prime=5; prime<=m; +++prime)
        {
            bool is_prime = true;
            for (std::vector<T>::const_iterator iter=primes.begin(), end=primes.end();
                 iter!=end; ++iter)
            {
                T div = *iter;
                if (div*div > prime)
                    break;
                if (prime % div == 0)
                {
                    is_prime = false;
                    break;
                }
            }
            if (is_prime)
                primes.push_back (prime);
        }
    }

    T g = 1;
    for (size_t i=0; i<primes.size() && g==1; i++)
    {
        T cur = primes[i];
        while (cur <= n)
            cur *= primes[i];
        cur /= primes[i];
        b = powmod (b, cur, n);
        g = gcd (abs(b-1), n);
        if (g == n)

```

```
    g = 1;
}

return g;
}
```

Метод Ферма

Это стопроцентный метод, но он может работать очень медленно, если у числа есть маленькие делители.

Поэтому запускать его стоит только после всех остальных методов.

```
template <class T, class T2>
T ferma (const T & n, T2 unused)
{
    T2
    x = sq_root (n),
    y = 0,
    r = x*x - y*y - n;
    for (;;)
        if (r == 0)
            return x!=y ? x-y : x+y;
        else
            if (r > 0)
            {
                r -= y+y+1;
                ++y;
            }
            else
            {
                r += x+x+1;
                ++x;
            }
}
```

Тривиальное деление

Этот элементарный метод пригодится, чтобы сразу обрабатывать числа с очень маленькими делителями.

```
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m)
{
    // сначала проверяем тривиальные случаи
    if (n == 2 || n == 3)
        return 1;
    if (n < 2)
        return 0;
    if (even (n))
        return 2;

    // генерируем простые от 3 до m
    T2 pi;
    const vector<T2> & primes = get_primes (m, pi);

    // делим на все простые
    for (std::vector<T2>::const_iterator iter=primes.begin(), end=primes.end();
         iter!=end; ++iter)
    {
        const T2 & div = *iter;
        if (div * div > n)
            break;
        else
            if (n % div == 0)
                return div;
    }

    if (n < m*m)
        return 1;
    return 0;
}
```

Собираем всё вместе

Объединяем все методы в одной функции.

Также функция использует тест на простоту, иначе алгоритмы факторизации могут работать очень долго. Например, можно выбрать тест BPSW ([читать статью по BPSW](#)).

```
template <class T, class T2>
void factorize (const T & n, std::map<T,unsigned> & result, T2 unused)
{
```

```

if (n == 1)
;
else
// проверяем, не простое ли число
if (isprime (n))
++result[n];
else
// если число достаточно маленькое, то его разлагаем простым перебором
if (n < 1000*1000)
{
T div = prime_div_trivial (n, 1000);
++result[div];
factorize (n / div, result, unused);
}
else
{
// число большое, запускаем на нем алгоритмы факторизации
T div;
// сначала идут быстрые алгоритмы Полларда
div = pollard_monte_carlo (n);
if (div == 1)
div = pollard_rho (n);
if (div == 1)
div = pollard_p_1 (n);
if (div == 1)
div = pollard_bent (n);
// придется запускать 100%-ый алгоритм Ферма
if (div == 1)
div = ferma (n, unused);
// рекурсивно обрабатываем найденные множители
factorize (div, result, unused);
factorize (n / div, result, unused);
}
}
}

```

Приложение

[Скачать \[5 КБ\]](#) исходник программы, которая использует все указанные методы факторизации и тест BPSW на простоту.

Быстрое преобразование Фурье за $O(N \log N)$. Применение к умножению двух полиномов или длинных чисел

Здесь мы рассмотрим алгоритм, который позволяет перемножить два полинома длиной n за время $O(n \log n)$, что значительно лучше времени $O(n^2)$, достигаемого тривиальным алгоритмом умножения. Очевидно, что умножение двух длинных чисел можно свести к умножению полиномов, поэтому два длинных числа также можно перемножить за время $O(n \log n)$.

Изобретение Быстрого преобразования Фурье приписывается Кули (Cooley) и Таки (Tukey) — 1965 г. На самом деле БПФ неоднократно изобреталось до этого, но важность его в полной мере не осознавалась до появления современных компьютеров. Некоторые исследователи приписывают открытие БПФ Рунге (Runge) и Кёнигу (Konig) в 1924 г. Наконец, открытие этого метода приписывается ещё Гауссу (Gauss) в 1805 г.

Дискретное преобразование Фурье (ДПФ)

Пусть имеется многочлен n -ой степени:

$$A(x) = a_0x^0 + a_1x^1 + \dots + a_{n-1}x^{n-1}$$

Не теряя общности, можно считать, что n является степенью 2. Если в действительности n не является степенью 2, то мы просто добавим недостающие коэффициенты, положив их равными нулю.

Из теории функций комплексного переменного известно, что комплексных корней n -ой степени из единицы существует ровно n . Обозначим эти корни через $w_{n,k}$, $k = 0 \dots n - 1$, тогда известно, что $w_{n,k} = e^{i\frac{2\pi k}{n}}$. Кроме того, один из этих корней $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (называемый главным значением корня n -ой степени из единицы) таков, что все остальные корни являются его степенями: $w_{n,k} = (w_n)^k$.

Тогда **дискретным преобразованием Фурье (ДПФ)** (discrete Fourier transform, DFT) многочлена $A(x)$ (или, что то же самое, ДПФ вектора его коэффициентов $(a_0, a_1, \dots, a_{n-1})$) называются значения этого многочлена в точках $x = w_{n,k}$, т.е. это вектор:

$$\text{DFT}(a_0, a_1, \dots, a_{n-1}) = (y_0, y_1, \dots, y_{n-1}) = (A(w_{n,0}), A(w_{n,1}), \dots, A(w_{n,n-1})) = (A(w_n^0), A(w_n^1), \dots, A(w_n^{n-1}))$$

Аналогично определяется и **обратное дискретное преобразование Фурье** (InverseDFT). Обратное ДПФ для вектора значений многочлена $(y_0, y_1, \dots, y_{n-1})$ — это вектор коэффициентов многочлена $(a_0, a_1, \dots, a_{n-1})$:

$$\text{InverseDFT}(y_0, y_1, \dots, y_{n-1}) = (a_0, a_1, \dots, a_{n-1})$$

Таким образом, если прямое ДПФ переходит от коэффициентов многочлена к его значениям в комплексных корнях n -ой степени из единицы, то обратное ДПФ — наоборот, по значениям многочлена восстанавливает коэффициенты многочлена.

Применение ДПФ для быстрого умножения полиномов

Пусть даны два многочлена A и B . Посчитаем ДПФ для каждого из них: $\text{DFT}(A)$ и $\text{DFT}(B)$ — это два вектора-значения многочленов.

Теперь, что происходит при умножении многочленов? Очевидно, в каждой точке их значения просто перемножаются, т.е.

$$(A \times B)(x) = A(x) \times B(x)$$

Но это означает, что если мы перемножим вектора $\text{DFT}(A)$ и $\text{DFT}(B)$, просто умножив каждый элемент одного вектора на соответствующий ему элемент другого вектора, то мы получим не что иное, как ДПФ от многочлена $A \times B$:

$$\text{DFT}(A \times B) = \text{DFT}(A) \times \text{DFT}(B)$$

Наконец, применяя обратное ДПФ, получаем:

$$A \times B = \text{InverseDFT}(\text{DFT}(A) \times \text{DFT}(B))$$

Где, повторимся, справа под произведением двух ДПФ понимается попарные произведения элементов векторов. Такое произведение, очевидно, требует для вычисления только $O(n)$ операций. Таким образом, если мы научимся вычислять ДПФ и обратное ДПФ за время $O(n \log n)$, то и произведение двух полиномов (а, следовательно, и двух длинных чисел) мы сможем найти за ту же асимптотику.

Следует заметить, что, во-первых, два многочлена следует привести к одной степени (просто дополнив коэффициенты одного из них нулями). Во-вторых, в результате произведения двух многочленов степени n получается многочлен степени $2n - 1$, поэтому, чтобы результат получился корректным, предварительно нужно удвоить степени каждого многочлена (опять же, дополнив их нулевыми коэффициентами).

Быстрое преобразование Фурье

Быстрое преобразование Фурье (fast Fourier transform) — это метод, позволяющий вычислять ДПФ за время $O(n \log n)$. Этот метод основывается на свойствах комплексных корней из единицы (а именно, на том, что степени одних корней дают другие корни).

Основная идея БПФ заключается в разделении вектора коэффициентов на два вектора, рекурсивном вычислении ДПФ для них, и объединении результатов в одно БПФ.

Итак, пусть имеется многочлен $A(x)$ степени n , где n — степень двойки, и $n > 1$:

$$A(x) = a_0x^0 + a_1x^1 + \dots + a_{n-1}x^{n-1}$$

Разделим его на два многочлена, один — с чётными, а другой — с нечётными коэффициентами:

$$A_0(x) = a_0x^0 + a_2x^1 + \dots + a_{n-2}x^{n/2-1}$$

$$A_1(x) = a_1x^0 + a_3x^1 + \dots + a_{n-1}x^{n/2-1}$$

Нетрудно убедиться, что:

$$A(x) = A_0(x^2) + xA_1(x^2) \quad (1)$$

Многочлены A_0 и A_1 имеют вдвое меньшую степень, чем многочлен A . Если мы сможем за линейное время по вычисленным $\text{DFT}(A_0)$ и $\text{DFT}(A_1)$ вычислить $\text{DFT}(A)$, то мы и получим искомый алгоритм быстрого преобразования Фурье (т.к. это стандартная схема алгоритма "разделяй и властвуй", и для неё известна асимптотическая оценка $O(n \log n)$).

Итак, пусть мы имеем вычисленные вектора $\{y_k^0\}_{k=0}^{n/2-1} = \text{DFT}(A_0)$ и $\{y_k^1\}_{k=0}^{n/2-1} = \text{DFT}(A_1)$. Найдём выражения для $\{y_k\}_{k=0}^{n-1} = \text{DFT}(A)$.

Во-первых, вспоминая (1), мы сразу получаем значения для первой половины коэффициентов:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots n/2 - 1$$

Для второй половины коэффициентов после преобразований также получаем простую формулу:

$$\begin{aligned} y_{k+n/2} &= A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) = \\ &= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1 \end{aligned}$$

(Здесь мы воспользовались (1), а также тождествами $w_n^n = 1$, $w_n^{n/2} = -1$.)

Итак, в результате мы получили формулы для вычисления всего вектора $\{y_k\}$:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \dots n/2 - 1$$

$$y_{k+n/2} = y_k^0 - w_n^k y_k^1, \quad k = 0 \dots n/2 - 1$$

(эти формулы, т.е. две формулы вида $a + bc$ и $a - bc$, иногда называют "преобразование бабочки" ("butterfly operation"))

Тем самым, мы окончательно построили алгоритм БПФ.

Обратное БПФ

Итак, пусть дан вектор $(y_0, y_1, \dots, y_{n-1})$ — значения многочлена A степени n в точках $x = w_n^k$. Требуется восстановить коэффициенты

$(a_0, a_1, \dots, a_{n-1})$ многочлена. Эта известная задача называется **интерполяцией**, для этой задачи есть и общие алгоритмы решения, однако в данном случае будет получен очень простой алгоритм (простой тем, что он практически не отличается от прямого БПФ).

ДПФ мы можем записать, согласно его определению, в матричном виде:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Тогда вектор $(a_0, a_1, \dots, a_{n-1})$ можно найти, умножив вектор $(y_0, y_1, \dots, y_{n-1})$ на обратную матрицу к матрице, стоящей слева (которая, кстати, называется **матрицей Вандермонда**):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Непосредственной проверкой можно убедиться в том, что эта обратная матрица такова:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}$$

Таким образом, получаем формулу:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Сравнивая её с формулой для y_k :

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj}$$

мы замечаем, что эти две задачи почти ничем не отличаются, поэтому коэффициенты a_k можно находить таким же алгоритмом "разделяй и властвуй", как и прямое БПФ, только вместо w_n^k везде надо использовать w_n^{-k} , а каждый элемент результата надо разделить на n .

Таким образом, вычисление обратного ДПФ почти не отличается от вычисления прямого ДПФ, и его также можно выполнять за время $O(n \log n)$.

Реализация

Рассмотрим простую рекурсивную **реализацию БПФ** и обратного БПФ, реализуем их в виде одной функции, поскольку различия между прямым и обратным БПФ минимальны. Для хранения комплексных чисел воспользуемся стандартным в C++ STL типом `complex` (определенным в заголовочном файле `<complex>`).

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
    int n = (int) a.size();
    if (n == 1) return;

    vector<base> a0 (n/2), a1 (n/2);
    for (int i=0, j=0; i<n; i+=2, ++j) {
        a0[j] = a[i];
        a1[j] = a[i+1];
    }
    fft (a0, invert);
    fft (a1, invert);

    double ang = 2*PI/n * (invert ? -1 : 1);
    base w (1), wn (cos(ang), sin(ang));
    for (int i=0; i<n/2; ++i) {
        a[i] = a0[i] + w * a1[i];
        a[i+n/2] = a0[i] - w * a1[i];
        if (invert)
            a[i] /= 2, a[i+n/2] /= 2;
        w *= wn;
    }
}
```

В аргумент `a` функции передаётся входной вектор коэффициентов, в нём же и будет содержаться результат. Аргумент `invert` показывает,

прямое или обратное ДПФ следует вычислить. Внутри функции сначала проверяется, что если длина вектора a равна единице, то ничего делать не надо — он сам и является ответом. Иначе вектор a разделяется на два вектора a_0 и a_1 , для которых рекурсивно вычисляется ДПФ. Затем вычисляется величина w_n , и заводится переменная w , содержащая текущую степень w_n . Затем вычисляются элементы результирующего ДПФ по вышеописанным формулам.

Если указан флаг `invert = true`, то w_n заменяется на w_n^{-1} , а каждый элемент результата делится на 2 (учитывая, что эти деления на 2 произойдут в каждом уровне рекурсии, то в итоге как раз получится, что все элементы поделятся на n).

Тогда функция для **перемножения двух многочленов** будет выглядеть следующим образом:

```
void multiply (const vector<int> & a, const vector<int> & b, vector<int> & res) {
    vector<base> fa (a.begin(), a.end()), fb (b.begin(), b.end());
    size_t n = 1;
    while (n < max (a.size(), b.size())) n <= 1;
    n <= 1;
    fa.resize (n), fb.resize (n);

    fft (fa, false), fft (fb, false);
    for (size_t i=0; i<n; ++i)
        fa[i] *= fb[i];
    fft (fa, true);

    res.resize (n);
    for (size_t i=0; i<n; ++i)
        res[i] = int (fa[i].real() + 0.5);
}
```

Эта функция работает с многочленами с целочисленными коэффициентами (хотя, понятно, теоретически ничто не мешает ей работать и с дробными коэффициентами). Однако здесь проявляется проблема большой погрешности при вычислении ДПФ: погрешность может оказаться значительной, поэтому округлять числа лучше самым надёжным способом — прибавлением 0.5 и последующим округлением вниз.

Наконец, функция для **перемножения двух длинных чисел** практически ничем не отличается от функции для перемножения многочленов. Единственная особенность — что после выполнения умножения чисел как многочлены их следует нормализовать, т.е. выполнить все переносы разрядов:

```
int carry = 0;
for (size_t i=0; i<n; ++i) {
    res[i] += carry;
    carry = res[i] / 10;
    res[i] %= 10;
}
```

(Поскольку длина произведения двух чисел никогда не превзойдёт суммарной длины чисел, то размера вектора `res` хватит, чтобы выполнить все переносы.)

Улучшенная реализация

Для увеличения эффективности откажемся от рекурсии в явном виде. В приведённой выше рекурсивной реализации мы явно разделяли вектор a на два вектора — элементы на чётных позициях отнесли к одному временно созданному вектору, а на нечётных — к другому. Однако, если бы мы переупорядочили элементы определённым образом, то необходимость в создании временных векторов тогда бы отпала (т.е. все вычисления мы могли бы производить "на месте", прямо в самом векторе a).

Заметим, что на первом уровне рекурсии элементы, младшие (первые) биты позиций которых равны нулю, относятся к вектору a_0 , а младшие биты позиций которых равны единице — к вектору a_1 . На втором уровне рекурсии выполняется то же самое, но уже для вторых битов, и т.д. Поэтому если мы в позиции i каждого элемента $a[i]$ инвертируем порядок битов, и переупорядочим элементы массива a в соответствии с новыми индексами, то мы и получим искомый порядок (он называется **поразрядно обратной перестановкой** (bit-reversal permutation)).

Например, для $n = 8$ этот порядок имеет вид:

$$a = \left\{ [(a_0, a_4), (a_2, a_6)], [(a_1, a_5), (a_3, a_7)] \right\}$$

Действительно, на первом уровне рекурсии (окружено фигурными скобками) обычного рекурсивного алгоритма происходит разделение вектора на две части: $[a_0, a_2, a_4, a_6]$ и $[a_1, a_3, a_5, a_7]$. Как мы видим, в поразрядно обратной перестановке этому соответствует просто разделение вектора на две половинки: первые $n/2$ элементов, и последние $n/2$ элементов. Затем происходит рекурсивный вызов от каждой половинки; пусть результирующее ДПФ от каждой из них было возвращено на месте самих элементов (т.е. в первой и второй половинах вектора a соответственно):

$$a = \left\{ [y_0^0, y_1^0, y_2^0, y_3^0], [y_0^1, y_1^1, y_2^1, y_3^1] \right\}$$

Теперь нам надо выполнить объединение двух ДПФ в одно для всего вектора. Но элементы встали так удачно, что и объединение можно выполнить прямо в этом массиве. Действительно, возьмём элементы y_0^0 и y_0^1 , применим к ним преобразование бабочки, и результат поставим на их место — и это место и окажется тем самым, которое и должно было получиться:

$$a = \left\{ [y_0^0 + w_n^0 y_1^1, y_1^0, y_2^0, y_3^0], [y_0^0 - w_n^0 y_1^1, y_1^1, y_2^1, y_3^1] \right\}$$

Аналогично, применяем преобразование бабочки к y_1^0 и y_1^1 и результат ставим на их место, и т.д. В итоге получаем:

$$a = \left\{ [y_0^0 + w_n^0 y_1^1, y_1^0 + w_n^1 y_1^1, y_2^0 + w_n^2 y_1^1, y_3^0 + w_n^3 y_1^1], [y_0^0 - w_n^0 y_1^1, y_1^0 - w_n^1 y_1^1, y_2^0 - w_n^2 y_1^1, y_3^0 - w_n^3 y_1^1] \right\}$$

Т.е. мы получили именно искомое ДПФ от вектора a .

Мы описали процесс вычисления ДПФ на первом уровне рекурсии, но понятно, что те же самые рассуждения верны и для всех остальных уровней рекурсии. Таким образом, **после применения поразрядно обратной перестановки вычислять ДПФ можно на месте**, без привлечения дополнительных массивов.

Но теперь можно **избавиться и от рекурсии** в явном виде. Итак, мы применили поразрядно обратную перестановку элементов. Теперь выполним всю работу, выполняемую нижним уровнем рекурсии, т.е. вектор a разделим на пары элементов, для каждого применим преобразование бабочки, в результате в векторе a будут находиться результаты работы нижнего уровня рекурсии. На следующем шаге разделим вектор a на четвёрки элементов, к каждой применим преобразование бабочки, в результате получим ДПФ для каждой четвёрки. И так далее, наконец, на последнем шаге мы, получив результаты ДПФ для двух половинок вектора a , применим к ним преобразование бабочки и получим ДПФ для всего вектора a .

Итак, реализация:

```
typedef complex<double> base;

int rev (int num, int lg_n) {
    int res = 0;
    for (int i=0; i<lg_n; ++i)
        if (num & (1<<i))
            res |= 1<<(lg_n-1-i);
    return res;
}

void fft (vector<base> & a, bool invert) {
    int n = (int) a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n)   ++lg_n;

    for (int i=0; i<n; ++i)
        if (i < rev(i,lg_n))
            swap (a[i], a[rev(i,lg_n)]);

    for (int len=2; len<=n; len<=1) {
        double ang = 2*PI/len * (invert ? -1 : 1);
        base wlen (cos(ang), sin(ang));
        for (int i=0; i<n; i+=len) {
            base w (1);
            for (int j=0; j<len/2; ++j) {
                base u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
        if (invert)
            for (int i=0; i<n; ++i)
                a[i] /= n;
    }
}
```

Вначале к вектору a применяется поразрядно обратная перестановка, для чего вычисляется количество значащих бит ($\lg n$) в числе n , и для каждой позиции i находится соответствующая ей позиция, битовая запись которой есть битовая запись числа i , записанная в обратном порядке. Если получившаяся в результате позиция оказалась больше i , то элементы в этих двух позициях надо обменять (если не это условие, то каждая пара обменяется дважды, и в итоге ничего не произойдёт).

Затем выполняется $\lg n - 1$ стадий алгоритма, на k -ой из которых ($k = 2 \dots \lg n$) вычисляются ДПФ для блоков длины 2^k . Для всех этих блоков будет одно и то же значение первообразного корня w_{2^k} , которое и запоминается в переменной $wlen$. Цикл по i итерируется по блокам, а вложенный в него цикл по j применяет преобразование бабочки ко всем элементам блока.

Можно выполнить дальнейшую **оптимизацию реверса битов**. В предыдущей реализации мы явно проходили по всем битам числа, попутно строя поразрядно инвертированное число. Однако реверс битов можно выполнять и по-другому.

Например, пусть j — уже подсчитанное число, равное обратной перестановке битов числа i . Тогда, при переходе к следующему числу $i + 1$ мы должны к числу j прибавить единицу, но прибавить её в такой "инвертированной" системе счисления. В обычной двоичной системе счисления прибавить единицу — значит удалить все единицы, стоящие на конце числа (т.е. группу младших единиц), а перед ними поставить единицу. Соответственно, в "инвертированной" системе мы должны идти по битам числа, начиная со старших, и пока там стоят единицы, удалять их и переходить к следующему биту; когда же встретится первый нулевой бит, поставить в него единицу и остановиться.

Итак, получаем такую реализацию:

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
    int n = (int) a.size();

    for (int i=1, j=0; i<n; ++i) {
        int bit = n >> 1;
        for (; j>=bit; bit>>=1)
            j -= bit;
        j += bit;
        if (i < j)
            swap (a[i], a[j]);
    }

    for (int len=2; len<=n; len<=1) {
        double ang = 2*PI/len * (invert ? -1 : 1);
```

```

base wlen (cos(ang), sin(ang));
for (int i=0; i<n; i+=len) {
    base w (1);
    for (int j=0; j<len/2; ++j) {
        base u = a[i+j], v = a[i+j+len/2] * w;
        a[i+j] = u + v;
        a[i+j+len/2] = u - v;
        w *= wlen;
    }
}
if (invert)
    for (int i=0; i<n; ++i)
        a[i] /= n;
}

```

Впрочем, возможны и другие реализации реверса битов (например, частичный предпосчт в таблицах).

Другой полезной оптимизацией является **отсечение по длине**: когда длина массива становится маленькой (скажем, 4), вычислять ДПФ для него эти рекурсивным алгоритмом уже слишком затратно. Более целесообразно расписать эти случаи в виде явных формул (например, при $n = 4$ все синусы и косинусы будут принимать только значения $\{-1; 0; 1\}$), в результате можно получить прирост скорости ещё на несколько десятков процентов.

Дискретное преобразование Фурье в модульной арифметике

В основе дискретного преобразования Фурье лежат комплексные числа, корни n -ой степени из единицы. Для эффективного его вычисления использовались такие особенности корней, как существование n различных корней, образующих группу (т.е. степень одного корня — всегда другой корень; среди них есть один элемент — генератор группы, называемый примитивным корнем).

Но то же самое верно и в отношении корней n -ой степени из единицы в модульной арифметике. Точнее, не для любого модуля p найдётся n различных корней из единицы, однако такие модули всё же существуют. По-прежнему нам важно найти среди них примитивный корень, т.е.:

$$(w_n)^n = 1 \pmod{p}, \\ (w_n)^k \neq 1 \pmod{p}, \quad 1 \leq k < n$$

Все остальные $n - 1$ корней n -ой степени из единицы по модулю p можно получить как степени примитивного корня w_n (как и в комплексном случае).

Для применения в алгоритме Быстрого преобразования Фурье нам было нужно, чтобы примитивный корень существовал для некоторого n , являвшегося степенью двойки, а также всех меньших степеней. И если в комплексном случае примитивный корень существовал для любого n , то в случае модульной арифметики это, вообще говоря, не так. Однако, заметим, что если $n = 2^k$, т.е. k -ая степень двойки, то по модулю $m = 2^{k-1}$ имеем:

$$(w_n^2)^m = (w_n)^n = 1 \pmod{p} \\ (w_n^2)^k = w_n^{2k} \neq 1 \pmod{p}, \quad 1 \leq k < m$$

Таким образом, если w_n — примитивный корень $n = 2^k$ -ой степени из единицы, то w_n^2 — примитивный корень 2^{k-1} -ой степени из единицы. Следовательно, для всех степеней двойки, меньших n , примитивные корни нужной степени также существуют, и могут быть вычислены как соответствующие степени w_n .

Последний штрих — для обратного ДПФ мы использовали вместо w_n обратный ему элемент: w_n^{-1} . Но по простому модулю p обратный элемент также всегда найдётся.

Таким образом, все нужные нам свойства соблюдаются и в случае модульной арифметики, при условии, что мы выбрали некоторый достаточно большой модуль p и нашли в нём примитивный корень n -ой степени из единицы.

Например, можно взять такие значения: модуль $p = 7340033$, $w_{2^{20}} = 5$. Если этого модуля будет недостаточно, для нахождения другой пары можно воспользоваться фактом, что для модулей вида $c2^k + 1$ (но по-прежнему обязательно простых) всегда найдётся примитивный корень степени 2^k из единицы.

```

const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1<<20;

void fft (vector<int> & a, bool invert) {
    int n = (int) a.size();

    for (int i=1, j=0; i<n; ++i) {
        int bit = n >> 1;
        for (; j>=bit; bit>>=1)
            j -= bit;
        j += bit;
        if (i < j)
            swap (a[i], a[j]);
    }

    for (int len=2; len<=n; len<<=1) {
        int wlen = invert ? root_1 : root;
        for (int i=len; i<root_pw; i<<=1)
            wlen = int (wlen * 111 * wlen % mod);
        for (int i=0; i<n; i+=len) {
            int w = 1;
            for (int j=0; j<len/2; ++j) {
                int u = a[i+j], v = int (a[i+j+len/2] * 111 * w % mod);
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
}

```

```

    a[i+j] = u+v < mod ? u+v : u+v-mod;
    a[i+j+len/2] = u-v >= 0 ? u-v : u-v+mod;
    w = int (w * lll * wlen % mod);
}
}
if (invert) {
    int nrev = reverse (n, mod);
    for (int i=0; i<n; ++i)
        a[i] = int (a[i] * lll * nrev % mod);
}
}

```

Здесь функция `reverse` находит обратный к n элемент по модулю mod (см. [Обратный элемент в поле по модулю](#)). Константы mod , root , root_pw определяют модуль и примитивный корень, а root_l — обратный к root элемент по модулю mod .

Как показывает практика, реализация целочисленного ДПФ работает даже медленней реализации с комплексными числами (из-за огромного количества операций взятия по модулю), однако она имеет такие преимущества, как меньшее использование памяти и отсутствие погрешностей округления.

Некоторые применения

Помимо непосредственного применения для перемножения многочленов или длинных чисел, опишем здесь некоторые другие приложения дискретного преобразования Фурье.

Всевозможные суммы

Задача: даны два массива $a[]$ и $b[]$. Требуется найти всевозможные числа вида $a[i] + b[j]$, и для каждого такого числа вывести количество способов получить его.

Например, для $a = (1, 2, 3)$ и $b = (2, 4)$ получаем: число 3 можно получить 1 способом, 4 — также одним, 5 — 2, 6 — 1, 7 — 1.

Построим по массивам a и b два многочлена A и B . В качестве степеней в многочлене будут выступать сами числа, т.е. значения $a[i]$ ($b[i]$), а в качестве коэффициентов при них — сколько раз это число встречается в массиве a (b).

Тогда, перемножив эти два многочлена за $O(n \log n)$, мы получим многочлен C , где в качестве степеней будут всевозможные числа вида $a[i] + b[i]$, а коэффициенты при них будут как раз искомыми количествами

Всевозможные скалярные произведения

Даны два массива $a[]$ и $b[]$ одной длины n . Требуется вывести значения каждого скалярного произведения вектора a на очередной циклический сдвиг вектора b .

Инвертируем массив a и припишем к нему в конец n нулей, а к массиву b — просто припишем самого себя. Затем перемножим их как многочлены за $O(n \log n)$. Теперь рассмотрим коэффициенты произведения $c[n \dots 2n - 1]$ (как всегда, все индексы в 0-индексации). Имеем:

$$c[k] = \sum_{i+j=k} a[i]b[j]$$

Поскольку все элементы $a[i] = 0$, $i = n \dots 2n - 1$, то мы получаем:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i]$$

Нетрудно увидеть в этой сумме, что это именно скалярное произведение вектора a на $k - n$ -ый циклический сдвиг. Таким образом, эти коэффициенты — и есть ответ на задачу.

Две полоски

Даны две полоски, заданные как два булевыхских (т.е. числовых со значениями 0 или 1) массива $a[]$ и $b[]$. Требуется найти все такие позиции на первой полоске, что если приложить, начиная с этой позиции, вторую полоску, ни в каком месте не получится `true` сразу на обеих полосках. Эту задачу можно переформулировать таким образом: дана карта полоски, в виде 0/1 — можно вставлять в эту клетку или нет, и дана некоторая фигурка в виде шаблона (в виде массива, в котором 0 — нет клетки, 1 — есть), требуется найти все позиции в полоске, к которым можно приложить фигурку.

Эта задача фактически ничем не отличается от предыдущей задачи — задачи о скалярном произведении. Действительно, скалярное произведение двух 0/1 массивов — это количество элементов, в которых одновременно оказались единицы. Наша задача в том, чтобы найти все циклические сдвиги второй полоски так, чтобы не нашлось ни одного элемента, в котором бы в обеих полосках оказались единицы. Т.е. мы должны найти все циклические сдвиги второго массива, при которых скалярное произведение равно нулю.

Таким образом, и эту задачу мы решили за $O(n \log n)$.

Поиск в ширину

Это один из основных алгоритмов на графах.

В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер.

Алгоритм работает за $O(N+M)$.

Приложения алгоритма

- Поиск кратчайшего пути в невзвешенном графе.
- Нахождение решения какой-либо задачи с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое - рёбрами графа.
- Нахождение кратчайшего пути в 0-1-графе (взвешенном, но веса каждого ребра равны 0 или 1):
Здесь сгодится поиск в ширину, слегка модифицированный: если текущее ребро нулевого веса, то новую вершину добавляем не в конец, а в начало очереди. В остальном всё то же самое.
- Нахождение кратчайшего цикла в неориентированном невзвешенном графе:
Производим поиск в ширину из каждой вершины. Если на каком-либо шаге в поиске мы сможем пойти в used-вершину (отличную от предка), то это и будет кратчайший цикл из текущей вершины.
- Найти все рёбра, лежащие на кратчайшем пути между заданной парой вершин.
Просто запустить 2 поиска в ширину: из A в B, и из B в A. Теперь для каждого ребра элементарно узнать, лежит ли он на кратчайшем пути.
- Найти все рёбра, лежащие на всех кратчайших путях.
Ребро (A,B) принадлежит какому-либо кратчайшему пути из вершины S в вершину T, если $D[S][A] + 1 + D[B][T] = D[S][T]$, где $D[i][j]$ - расстояние между вершинами i и j.
- Найти кратчайший чётный путь в графе (т.е. путь чётной длины).
Сделать граф, вершины которого - состояния: текущая вершина, текущая чётность пути.
- Найти все вершины, лежащие на кратчайшем пути.
Нужно 2 поиска в ширину, и вершина i лежит на кратчайшем пути между A и B, если $D[A][i] + D[i][B] = D[A][B]$, где $D[i][j]$ - расстояние между вершинами i и j.

Реализация

```
vector<vector<int>> g; // граф
int n; // число вершин
int start; // стартовая вершина

// считываем граф
...

// собственно поиск
vector<int> q (n); // очередь посещения вершин
int h=0, t=0; // указатели на начало и конец очереди
vector<bool> used (n); // покрашена вершина или нет
vector<int> parent (n); // предки для восстановления пути
q[t++] = start; used[start] = true; parent[start] = -1; // идём в стартовую вершину
while (h < t)
{
    int cur = q[h++];
    for (unsigned i=0; i < g[cur].size(); ++i)
    {
        int to = g[cur][i];
        if (!used[to])
        {
            used[to] = true;
            q[t++] = to;
        }
    }
}

// вывод какого-то найденного пути
int to = ...;
if (!used[to])
    cout << "No path!";
else
{
    vector<int> path;
    for (int cur = to; cur != -1; cur = parent[cur])
        path.push_back (cur);
    cout << "Path: ";
    for (unsigned i = path.size(); i-- > 0; )
        cout << path[i] << ' ';
}
```

Поиск в глубину

Это один из основных алгоритмов на графах.

В результате поиска в глубину находится лексикографически первый путь в графе.

Алгоритм работает за $O(N+M)$.

Применения алгоритма

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе.
- Проверка, является ли одна вершина дерева предком другой:
В начале и конце итерации поиска в глубину будет запоминать "время" захода и выхода в каждой вершине. Теперь за $O(1)$ можно найти ответ: вершина i является предком вершины j тогда и только тогда, когда $\text{start}_i < \text{start}_j$ и $\text{end}_i > \text{end}_j$.
- Задача LCA (наименьший общий предок).
- Топологическая сортировка:
Запускаем серию поисков в глубину, чтобы обойти все вершины графа. Отсортируем вершины по времени выхода по убыванию - это и будет ответом.
- Проверка графа на ацикличность и нахождение цикла
- Поиск компонент сильной связности:
Сначала делаем топологическую сортировку, потом транспонируем граф и проводим снова серию поисков в глубину в порядке, определяемом топологической сортировкой. Каждое дерево поиска - сильносвязная компонента.
- Поиск мостов:
Сначала превращаем граф в ориентированный, делая серию поисков в глубину, и ориентируя каждое ребро так, как мы пытались по нему пройти. Затем находим сильносвязные компоненты. Мостами являются те рёбра, концы которых принадлежат разным сильносвязным компонентам.

Реализация

```
vector<vector<int>> g; // граф
int n; // число вершин

vector<int> color; // цвет вершины (0, 1, или 2)

vector<int> time_in, time_out; // "времена" захода и выхода из вершины
int dfs_timer = 0; // "таймер" для определения времён

void dfs (int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (vector<int>::iterator i=g[v].begin(); i!=g[v].end(); ++i)
        if (color[*i] == 0)
            dfs (*i);
    color[v] = 2;
    time_out[v] = dfs_timer++;
}
```

Это наиболее общий код. Во многих случаях времена захода и выхода из вершины не важны, так же как и не важны цвета вершин (но тогда надо будет ввести аналогичный по смыслу булевский массив used). Вот наиболее простая реализация:

```
vector<vector<int>> g; // граф
int n; // число вершин

vector<char> used;

void dfs (int v) {
    used[v] = true;
    for (vector<int>::iterator i=g[v].begin(); i!=g[v].end(); ++i)
        if (!used[*i])
            dfs (*i);
}
```

Топологическая сортировка

Быстрее всего эту задачу можно решить с помощью поиска в глубину - за $O(N+M)$.

Алгоритм

Произведём серию поисков в глубину, чтобы посетить весь граф. Отсортируем вершины по убыванию времени выхода - это и будет ответом.

Вместо сортировки можно просто сделать вектор `ans`, который будет изначально пустым, и добавлять в него текущую вершину `v` при возврате из текущей вершины. В таком случае вообще времена выхода в явном виде не потребуются.

Реализация

```
vector < vector<int> > g; // граф
int n; // число вершин

vector<bool> used;

vector<int> ans;

void dfs (int v)
{
    used[v] = true;
    for (vector<int>::iterator i=g[v].begin(); i!=g[v].end(); ++i)
        if (!used[*i])
            dfs (*i);
    ans.push_back (v);
}

void topological_sort (vector<int> & result)
{
    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs (i);
    result = ans;
}
```

Поиск компонент связности

Эта задача элементарно решается с помощью поиска в глубину или в ширину. Алгоритм работает за **O (N+M)**.

Алгоритм

Просто производим серию поисков в глубину (в ширину), чтобы посетить все вершины графа. Каждое дерево поиска и будет содержать отдельную компоненту связности.

Реализация

Код на основе поиска в ширину:

```
void find_connected_components (const vector < vector<int> > & g, int n)
{
    vector<bool> used (n);
    cout << "Components:\n";
    for (int v=0; v<n; ++v)
        if (!used[v])
        {
            cout << "[ ";
            vector<int> q (n);
            int h=0, t=0;
            q[t++] = v;
            used[v] = true;
            while (h < t)
            {
                int cur = q[h++];
                for (vector<int>::iterator i=g[cur].begin(); i!=g[cur].end(); ++i)
                    if (!used[*i])
                    {
                        used[*i] = true;
                        q[t++] = *i;
                        cout << ", " << *i;
                    }
            }
        }
}
```

```

    }
    cout << " ]\n";
}
}

```

Поиск компонент сильной связности, построение конденсации графа

Определения, постановка задачи

Дан ориентированный граф G , множество вершин которого V и множество рёбер — E . Петли и кратные рёбра допускаются. Обозначим через n количество вершин графа, через m — количество рёбер.

Компонентой сильной связности (strongly connected component) называется такое (максимальное по включению) подмножество вершин C , что любые две вершины этого подмножества достижимы друг из друга, т.е. для $\forall u, v \in C$:

$u \mapsto v, v \mapsto u$

где символом \mapsto здесь и далее мы будем обозначать достижимость, т.е. существование пути из первой вершины во вторую.

Понятно, что компоненты сильной связности для данного графа не пересекаются, т.е. фактически это разбиение всех вершин графа. Отсюда логично определение **конденсации** G^{SCC} как графа, получаемого из данного графа сжатием каждой компоненты сильной связности в одну вершину. Каждой вершине графа конденсации соответствует компонента сильной связности графа G , а ориентированное ребро между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин $u \in C_i, v \in C_j$, между которыми существовало ребро в исходном графе, т.е. $(u, v) \in E$.

Важнейшим свойством графа конденсации является то, что он **ацикличен**. Действительно, предположим, что $C \mapsto C'$, докажем, что $C' \not\mapsto C$. Из определения конденсации получаем, что найдутся две вершины $u \in C$ и $v \in C'$, что $u \mapsto v$. Доказывать будем от противного, т.е. предположим, что $C' \mapsto C$, тогда найдутся две вершины $u' \in C$ и $v' \in C'$, что $v' \mapsto u'$. Но т.к. u и u' находятся в одной компоненте сильной связности, то между ними есть путь; аналогично для v и v' . В итоге, объединяя пути, получаем, что $v \mapsto u$, и одновременно $u \mapsto v$. Следовательно, u и v должны принадлежать одной компоненте сильной связности, т.е. получили противоречие, что и требовалось доказать.

Описываемый ниже алгоритм выделяет в данном графе все компоненты сильной связности. Построить по нему граф конденсации не составит труда.

Алгоритм

Описываемый здесь алгоритм был предложен независимо Косараю (Kosaraju) и Шариром (Sharir) в 1979 г. Это очень простой в реализации алгоритм, основанный на двух сериях [поисков в глубину](#), и потому работающий за время $O(n + m)$.

На первом шаге алгоритма выполняется серия обходов в глубину, посещающая весь график. Для этого мы проходимся по всем вершинам графа и из каждой ещё не посещённой вершины вызываем обход в глубину. При этом для каждой вершины v запомним **время выхода** $tout[v]$. Эти времена выхода играют ключевую роль в алгоритме, и эта роль выражена в приведённой ниже теореме.

Сначала введём обозначение: время выхода $tout[C]$ из компоненты C сильной связности определим как максимум из значений $tout[v]$ для всех $v \in C$. Кроме того, в доказательстве теоремы будут упоминаться и времена входа в каждую вершину $tin[v]$, и аналогично определим времена входа $tin[C]$ для каждой компоненты сильной связности как минимум из величин $tin[v]$ для всех $v \in C$.

Теорема. Пусть C и C' — две различные компоненты сильной связности, и пусть в графике конденсации между ними есть ребро (C, C') . Тогда $tout[C] > tout[C']$.

При доказательстве возникает два принципиально различных случая в зависимости от того, в какую из компонент первой зайдёт обход в глубину, т.е. в зависимости от соотношения между $tin[C]$ и $tin[C']$:

- Первой была достигнута компонента C . Это означает, что в какой-то момент времени обход в глубину заходит в некоторую вершину v компоненты C , при этом все остальные вершины компонент C и C' ещё не посещены. Но, т.к. по условию в графике конденсаций есть ребро (C, C') , то из вершины v будет достижима не только вся компонента C , но и вся компонента C' . Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент C и C' , а, значит, они станут потомками по отношению к v в дереве обхода в глубину, т.е. для любой вершины $u \in C \cup C', u \neq v$ будет выполнено $tout[v] > tout[u]$, ч.т.д.
- Первой была достигнута компонента C' . Опять же, в какой-то момент времени обход в глубину заходит в некоторую вершину $v \in C'$, причём все остальные вершины компонент C и C' не посещены. Поскольку по условию в графике конденсаций существовало ребро (C, C') , то, вследствие ацикличности графа конденсаций, не существует обратного пути $C' \not\mapsto C$, т.е. обход в глубину из вершины v не достигнет вершин C . Это означает, что они будут посещены обходом в глубину позже, откуда и следует $tout[C] > tout[C']$, ч.т.д.

Доказанная теорема является **основой алгоритма** поиска компонент сильной связности. Из неё следует, что любое ребро (C, C') в графике конденсаций идёт из компоненты с большей величиной $tout$ в компоненту с меньшей величиной.

Если мы отсортируем все вершины $v \in V$ в порядке убывания времени выхода $tout[v]$, то первой окажется некоторая вершина u , принадлежащая "корневой" компоненте сильной связности, т.е. в которую не входит ни одно ребро в графике конденсаций. Теперь нам хотелось бы запустить такой обход из этой вершины u , который бы посетил только эту компоненту сильной связности и не зашёл ни в какую

другую; научившись это делать, мы сможем постепенно выделить все компоненты сильной связности: удалив из графа вершины первой выделенной компоненты, мы снова найдём среди оставшихся вершину с наибольшей величиной tout , снова запустим из неё этот обход, и т.д.

Чтобы научиться делать такой обход, рассмотрим **транспонированный граф** G^T , т.е. граф, полученный из G изменением направления каждого ребра на противоположное. Нетрудно понять, что в этом графе будут те же компоненты сильной связности, что и в исходном графе. Более того, граф конденсации $(G^T)^{\text{SCC}}$ для него будет равен транспонированному графу конденсации исходного графа G^{SCC} . Это означает, что теперь из рассматриваемой нами "корневой" компоненты уже не будут выходить рёбра в другие компоненты.

Таким образом, чтобы обойти всю "корневую" компоненту сильной связности, содержащую некоторую вершину v , достаточно запустить обход из вершины v в графе G^T . Этот обход посетит все вершины этой компоненты сильной связности и только их. Как уже говорилось, дальше мы можем мысленно удалить эти вершины из графа, находить очередную вершину с максимальным значением $\text{tout}[v]$ и запускать обход на транспонированном графе из неё, и т.д.

Итак, мы построили следующий **алгоритм** выделения компонент сильной связности:

1 шаг. Запустить серию обходов в глубину графа G , которая возвращает вершины в порядке увеличения времени выхода tout , т.е. некоторый список order .

2 шаг. Построить транспонированный граф G^T . Запустить серию обходов в глубину/ширину этого графа в порядке, определяемом списком order (а именно, в обратном порядке, т.е. в порядке уменьшения времени выхода). Каждое множество вершин, достигнутое в результате очередного запуска обхода, и будет очередной компонентой сильной связности.

Асимптотика алгоритма, очевидно, равна $O(n + m)$, поскольку он представляет собой всего лишь два обхода в глубину/ширину.

Наконец, уместно отметить связь с понятием **топологической сортировки**. Во-первых, шаг 1 алгоритма представляет собой не что иное, как топологическую сортировку графа G (фактически именно это и означает сортировка вершин по времени выхода). Во-вторых, сама схема алгоритма такова, что и компоненты сильной связности он генерирует в порядке уменьшения их времён выхода, таким образом, он генерирует компоненты – вершины графа конденсации в порядке топологической сортировки.

Реализация

```
vector<vector<int>> g, gr;
vector<char> used;
vector<int> order, component;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!used[g[v][i]])
            dfs1 (g[v][i]);
    order.push_back (v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back (v);
    for (size_t i=0; i<gr[v].size(); ++i)
        if (!used[gr[v][i]])
            dfs2 (gr[v][i]);
}

int main() {
    int n;
    ... чтение n ...
    for (;;) {
        int a, b;
        ... чтение очередного ребра (a,b) ...
        g[a].push_back (b);
        gr[b].push_back (a);
    }

    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs1 (i);
    used.assign (n, false);
    for (int i=0; i<n; ++i) {
        int v = order[n-1-i];
        if (!used[v]) {
            dfs2 (v);
            ... вывод очередной компонент ...
            component.clear ();
        }
    }
}
```

Здесь в g хранится сам граф, а gr — транспонированный граф. Функция dfs1 выполняет обход в глубину на графике G , функция dfs2 — на транспонированном G^T . Функция dfs1 заполняет список order вершинами в порядке увеличения времени выхода (фактически, делает топологическую сортировку). Функция dfs2 сохраняет все достигнутые вершины в списке component , который после каждого запуска будет содержать очередную компоненту сильной связности.

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]

- M. Sharir. A strong-connectivity algorithm and its applications in data-flow analysis [1979]

Поиск мостов

Пусть дан связный неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным.

Опишем алгоритм, основанный на поиске в глубину, работающий за $O(n + m)$, где n — количество вершин, m — рёбер.

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root . Заметим следующий факт (который несложно доказать).

Если текущее ребро таково, что ведёт в вершину, из которой и из любого её потомка нет обратного ребра в текущую вершину или её предка, то это ребро является мостом. В противном случае оно мостом не является.

Теперь осталось научиться для каждой вершины эффективно проверять, не найдётся ли из её потомка обратное ребро в текущую вершину или её предка. Для этого воспользуемся временами входа [поиска в глубину](#).

Итак, пусть $\text{tin}[v]$ — это время захода поиска в глубину в вершину v . Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $\text{tin}[v]$, времён захода в каждую вершину p , являющуюся концом некоторого обратного ребра (v, p) , а также из всех значений $fup[to]$ для каждой вершины to , являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} \text{tin}[v], \\ \text{tin}[p], & \text{for all } (v,p) \text{ — back edge} \\ fup[to], & \text{for all } (v,to) \text{ — tree edge} \end{cases}$$

(здесь "back edge" — обратное ребро, "tree edge" — ребро дерева)

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to , что $fup[to] < \text{tin}[v]$. Если $fup[to] = \text{tin}[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v .

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] > \text{tin}[v]$, то это ребро является мостом; в противном случае оно мостом не является.

Реализация

Здесь $\text{IS_BRIDGE}(a, b)$ — это некая функция, которая будет реагировать на то, что ребро (a, b) является мостом, например, выводить это ребро на экран.

Если говорить о самой реализации, то здесь нам нужно уметь различать три случая: когда мы идём по ребру дерева поиска в глубину, когда идём по обратному ребру, и когда пытаемся пойти по ребру дерева в обратную сторону. Это, соответственно, случаи $\text{used}[to] = \text{false}$, $\text{used}[to] = \text{true} \&\& to \neq \text{parent}$, и $to = \text{parent}$. Таким образом, нам надо передавать в функцию поиска в глубину вершину-предка текущей вершины.

Стоит заметить, что эта реализация некорректно работает при наличии в графе **кратных рёбер**: она фактически не обращает внимания, кратное ли ребро или оно единственное. Разумеется, кратные рёбра не должны входить в ответ, поэтому при вызове IS_BRIDGE можно проверять дополнительно, не кратное ли ребро мы хотим добавить в ответ. Другой способ — более аккуратная работа с предками, т.е. передавать в dfs не вершину-предка, а номер ребра, по которому мы вошли в вершину (для этого надо будет дополнительно хранить номера всех рёбер).

```
vector<vector<int>> g;
vector<char> used;
int timer;
vector<int> tin, fup;

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                IS_BRIDGE(v,to);
        }
    }
}
```

```

}

int main() {
    int n;
    ... чтение n и g ...

    timer = 0;
    used.assign (n, false);
    tin.resize (n);
    fup.resize (n);
    dfs (0);
}

```

Точки сочленения

Пусть дан связный неориентированный граф. **Точкой сочленения** называется такая вершина графа, удаление которой делает граф несвязным. Граф, не имеющий точек сочленения, называется двусвязным.

Опишем алгоритм, основанный на [поиске в глубину](#), позволяющий найти все точки сочленения в графе за время $O(N+M)$.

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root . Заметим два следующих факта (которые несложно доказать):

- Пусть v - вершина графа, $v \neq \text{root}$. Тогда, если найдётся такой потомок t вершины v в дереве поиска, что ни из него, ни из какого-либо его потомка нет ребра (обратного) в предка вершины v , то вершина v будет являться точкой сочленения. В противном случае, вершина v не является точкой сочленения.
- Рассмотрим теперь корень root дерева поиска. Тогда он является точкой сочленения тогда и только тогда, когда он имеет как минимум двух потомков в дереве поиска.

Теперь осталось научиться для каждой вершины эффективно проверять, не найдётся ли из её потомка обратное ребро в предка текущей вершины. Для этого воспользуемся временами входа поиска в глубину.

Итак, пусть $\text{tin}[v]$ - это время захода поиска в глубину в вершину v . Теперь введём массив $\text{fup}[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $\text{fup}[v]$ равно минимуму из времени захода в саму вершину $\text{tin}[v]$, времён захода в каждую из вершин p , являющихся концом некоторого обратного ребра (v,p) , а также из всех значений $\text{fup}[to]$ для каждой вершины to , являющейся непосредственным сыном v в дереве поиска:

```

fup[v] = min {
    tin[v],
    tin[p], где (v,p) - обратное ребро,
    fup[to], где (v,to) - ребро дерева
}

```

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to , что $\text{fup}[to] < \text{tin}[v]$.

Таким образом, если для текущей вершины $v \neq \text{root}$ найдётся такой непосредственный сын to , что $\text{fup}[to] \geq \text{tin}[v]$, то вершина v является точкой сочленения; в противном случае она точкой сочленения не является.

Реализация

На выходе в векторе `cutpoint` для каждой вершины будет содержаться булево значение - является она точкой сочленения или нет.

Если говорить о самой реализации, то здесь нужно уметь отличать случай, когда мы пытаемся идти в предка текущей вершины в дереве поиска в глубину. Такой проход не является проходом по обратному ребру, но тем не менее, в данном случае мы можем проигнорировать этот момент. Дело в том, что мы различаем два случая: когда в результате прохода по обратному ребру $\text{fup}[to] < \text{tin}[v]$, и, наоборот, когда $\text{fup}[to] \geq \text{tin}[v]$. Отсюда можно заметить, что проход по ребру в предка только сделает $\text{fup}[v]$ как максимум $\text{tin}[p]$, но не меньше, а потому этот проход нам ничего не ухудшает. Таким образом, строку, которая отмечена комментарием в нижеприведённом коде, можно безо всяких последствий удалить (как и передачу предка p в функцию).

```

vector < vector<int> > g;
vector<char> used;
vector<char> cutpoint;
int timer;
vector<int> tin, fup;

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    int children = 0;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];

```

```

if (to == p) continue; // эту строку можно удалить
if (used[to])
    fup[v] = min (fup[v], tin[to]);
else {
    ++children;
    dfs (to, v);
    fup[v] = min (fup[v], fup[to]);
    if (fup[to] >= tin[v])
        cutpoint[v] = true;
}
}
if (p == -1)
    cutpoint[v] = children > 1;
}

int main() {
int n;
... чтение n и g ...

timer = 0;
used.assign (n, false);
cutpoint.assign (n, false);
tin.resize (n);
fup.resize (n);
dfs (0);
}

```

Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры

Постановка задачи

Дан ориентированный или неориентированный взвешенный граф с n вершинами и m рёбрами. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершина s . Требуется найти длины кратчайших путей из вершины s во все остальные вершины, а также предоставить способ вывода самих кратчайших путей.

Эта задача называется "задачей о кратчайших путях с единственным источником" (single-source shortest paths problem).

Алгоритм

Здесь описывается алгоритм, который предложил датский исследователь **Дейкстра** (Dijkstra) в 1959 г.

Заведём массив $d[]$, в котором для каждой вершины v будем хранить текущую длину $d[v]$ кратчайшего пути из s в v . Изначально $d[s] = 0$, а для всех остальных вершин эта длина равна бесконечности (при реализации на компьютере обычно в качестве бесконечности выбирают просто достаточно большое число, заведомо большее возможной длины пути):

$$d[v] = \infty, v \neq s$$

Кроме того, для каждой вершины v будем хранить, помечена она ещё или нет, т.е. заведём булевский массив $u[]$. Изначально все вершины не помечены, т.е.

$$u[v] = \text{false}$$

Сам алгоритм Дейкстры состоит из n итераций. На очередной итерации выбирается вершина v с наименьшей величиной $d[v]$ среди ещё не помеченных, т.е.:

$$d[v] = \min_{p: u[p]=\text{false}} d[p]$$

(Понятно, что на первой итерации выбрана будет стартовая вершина s .)

Выбранная таким образом вершина v отмечается помеченной. Далее, на текущей итерации, из вершины v производятся **релаксации**: просматриваются все рёбра (v, to) , исходящие из вершины v , и для каждой такой вершины to алгоритм пытается улучшить значение $d[to]$. Пусть длина текущего ребра равна len , тогда в виде кода релаксация выглядит как:

$$d[to] = \min(d[to], d[v] + \text{len})$$

На этом текущая итерация заканчивается, алгоритм переходит к следующей итерации (снова выбирается вершина с наименьшей величиной d , из неё производятся релаксации, и т.д.). При этом в конце концов, после n итераций, все вершины графа станут помеченными, и алгоритм свою работу завершает. Утверждается, что найденные значения $d[v]$ есть искомые длины кратчайших путей из s в v .

Стоит заметить, что, если не все вершины графа достижимы из вершины s , то значения $d[v]$ для них так и останутся бесконечными. Понятно, что несколько последних итераций алгоритма будут как раз выбирать эти вершины, но никакой полезной работы производить эти итерации не будут (поскольку бесконечное расстояние не сможет прорелаксировать другие, даже тоже бесконечные расстояния). Поэтому алгоритм можно сразу останавливать, как только в качестве выбранной вершины берётся вершина с бесконечным расстоянием.

Восстановление путей. Разумеется, обычно нужно знать не только длины кратчайших путей, но и получить сами пути. Покажем, как сохранить информацию, достаточную для последующего восстановления кратчайшего пути из s до любой вершины. Для этого достаточно так называемого **массива предков**: массива $p[]$, в котором для каждой вершины $v \neq s$ хранится номер вершины $p[v]$, являющейся предпоследней в кратчайшем пути до вершины v . Здесь используется тот факт, что если мы возьмём кратчайший путь до какой-то вершины v , а затем удалим из этого пути последнюю вершину, то получится путь, оканчивающийся некоторой вершиной $p[v]$, и этот путь будет кратчайшим для вершины $p[v]$. Итак, если мы будем обладать этим массивом предков, то кратчайший путь можно будет восстановить по нему, просто каждый раз беря предка от текущей вершины, пока мы не придём в стартовую вершину s — так мы получим искомый кратчайший путь, но записанный в обратном порядке. Итак, кратчайший путь P до вершины v равен:

$$P = (s, \dots, p[p[p[v]]], p[p[v]], p[v], v)$$

Осталось понять, как строить этот массив предков. Однако это делается очень просто: при каждой успешной релаксации, т.е. когда из выбранной вершины v происходит улучшение расстояния до некоторой вершины to , мы записываем, что предком вершины to является вершина v :

$$p[to] = v$$

Доказательство

Основное утверждение, на котором основана корректность алгоритма Дейкстры, следующее. Утверждается, что после того как какая-либо вершина v становится помеченной, текущее расстояние до неё $d[v]$ уже является кратчайшим, и, соответственно, больше меняться не будет.

Доказательство будем производить по индукции. Для первой итерации справедливость очевидна — для вершины s имеем $d[s] = 0$, что и является длиной кратчайшего пути до неё. Пусть теперь это утверждение выполнено для всех предыдущих итераций, т.е. всех уже помеченных вершин; докажем, что оно не нарушается после выполнения текущей итерации. Пусть v — вершина, выбранная на текущей итерации, т.е. вершина, которую алгоритм собирается пометить. Докажем, что $d[v]$ действительно равно длине кратчайшего пути до неё (обозначим эту длину через $l[v]$).

Рассмотрим кратчайший путь P до вершины v . Понятно, этот путь можно разбить на два пути: P_1 , состоящий только из помеченных вершин (как минимум стартовая вершина s будет в этом пути), и остальная часть пути P_2 (она тоже может включать помеченные вершины, но начинается обязательно с непомеченной). Обозначим через p первую вершину пути P_2 , а через q — последнюю вершину пути P_1 .

Докажем сначала наше утверждение для вершины p , т.е. докажем равенство $d[p] = l[p]$. Однако это практически очевидно: ведь на одной из предыдущих итераций мы выбирали вершину q и выполняли релаксацию из неё. Поскольку (в силу самого выбора вершины p) кратчайший путь до p равен кратчайшему пути до q плюс ребро (p, q) , то при выполнении релаксации из q величина $d[p]$ действительно установится в требуемое значение.

Вследствие неотрицательности стоимостей рёбер длина кратчайшего пути $l[p]$ (а она по только что доказанному равна $d[p]$) не превосходит длины $l[v]$ кратчайшего пути до вершины v . Учитывая, что $l[v] \leq d[v]$ (ведь алгоритм Дейкстры не мог найти более короткого пути, чем это вообще возможно), в итоге получаем соотношения:

$$d[p] = l[p] \leq l[v] \leq d[v]$$

С другой стороны, поскольку p и v — вершины непомеченные, то так как на текущей итерации была выбрана именно вершина v , а не вершина p , то получаем другое неравенство:

$$d[p] \geq d[v]$$

Из этих двух неравенств заключаем равенство $d[p] = d[v]$, а тогда из найденных до этого соотношений получаем и:

$$d[v] = l[v]$$

что и требовалось доказать.

Реализация

Итак, алгоритм Дейкстры представляет собой n итераций, на каждой из которых выбирается непомеченная вершина с наименьшей величиной $d[v]$, эта вершина помечается, и затем просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра делается попытка улучшить значение $d[]$ на другом конце ребра.

Время работы алгоритма складывается из:

- n раз поиск вершины с наименьшей величиной $d[v]$ среди всех непомеченных вершин, т.е. среди $O(n)$ вершин
- m раз производится попытка релаксаций

При простейшей реализации этих операций на поиск вершины будет затрачиваться $O(n)$ операций, а на одну релаксацию — $O(1)$ операций, и итоговая **асимптотика** алгоритма составляет:

$$O(n^2 + m)$$

Реализация:

```
const int INF = 1000000000;

int main() {
    int n;
    ... чтение n ...
    vector < vector < pair<int, int> > > g (n);
```

```

... чтение графа ...
int s = ...; // стартовая вершина

vector<int> d (n, INF), p (n);
d[s] = 0;
vector<char> u (n);
for (int i=0; i<n; ++i) {
    int v = -1;
    for (int j=0; j<n; ++j)
        if (!u[j] && (v == -1 || d[j] < d[v]))
            v = j;
    if (d[v] == INF)
        break;
    u[v] = true;

    for (size_t j=0; j<g[v].size(); ++j) {
        int to = g[v][j].first,
            len = g[v][j].second;
        if (d[v] + len < d[to]) {
            d[to] = d[v] + len;
            p[to] = v;
        }
    }
}

```

Здесь граф g хранится в виде списков смежности: для каждой вершины v список $g[v]$ содержит список рёбер, исходящих из этой вершины, т.е. список пар $\langle \text{int}, \text{int} \rangle$, где первый элемент пары — вершина, в которую ведёт ребро, а второй элемент — вес ребра.

После чтения заводятся массивы расстояний $d[]$, меток $u[]$ и предков $p[]$. Затем выполняются n итераций. На каждой итерации сначала находится вершина v , имеющая наименьшее расстояние $d[]$ среди непомеченных вершин. Если расстояние до выбранной вершины v оказывается равным бесконечности, то алгоритм останавливается. Иначе вершина помечается как помеченная, и просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра выполняются релаксации. Если релаксация успешна (т.е. расстояние $d[to]$ меняется), то пересчитывается расстояние $d[to]$ и сохраняется предок $p[]$.

После выполнения всех итераций в массиве $d[]$ оказываются длины кратчайших путей до всех вершин, а в массиве $p[]$ — предки всех вершин (кроме стартовой s). Восстановить путь до любой вершины t можно следующим образом:

```

vector<int> path;
for (int v=t; v!=s; v=p[v])
    path.push_back (v);
path.push_back (s);
reverse (path.begin(), path.end());

```

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]
- Edsger Dijkstra. **A note on two problems in connexion with graphs** [1959]

Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры для разреженных графов

Постановку задачи, алгоритм и его доказательство см. в [статье об общем алгоритме Дейкстры](#).

Алгоритм

Напомним, что сложность алгоритма Дейкстры складывается из двух основных операций: время нахождения вершины с наименьшей величиной расстояния $d[v]$, и время совершения релаксации, т.е. время изменения величины $d[to]$.

При простейшей реализации эти операции потребуют соответственно $O(n)$ и $O(1)$ времени. Учитывая, что первая операция всего выполняется $O(n)$ раз, а вторая — $O(m)$, получаем асимптотику простейшей реализации алгоритма Дейкстры: $O(n^2 + m)$.

Понятно, что эта асимптотика является оптимальной для плотных графов, т.е. когда $m \approx n^2$. Чем более разрежен граф (т.е. чем меньше m по сравнению с максимальным количеством рёбер n^2), тем менее оптимальной становится эта оценка, и по вине первого слагаемого. Таким образом, надо улучшать время выполнения операций первого типа, не сильно ухудшая при этом время выполнения операций второго типа.

Для этого надо использовать различные вспомогательные структуры данных. Наиболее привлекательными являются **Фibonacciевы кучи**, которые позволяют производить операцию первого вида за $O(\log n)$, а второго — за $O(1)$. Поэтому при использовании Фibonacciевых

куч время работы алгоритма Дейкстры составит $O(n \log n + m)$, что является практически теоретическим минимумом для алгоритма поиска кратчайшего пути. Кстати говоря, эта оценка является оптимальной для алгоритмов, основанных на алгоритме Дейкстры, т.е. Фибоначчиевы кучи являются оптимальными с этой точки зрения (это утверждение об оптимальности на самом деле основано на невозможности существования такой "идеальной" структуры данных — если бы она существовала, то можно было бы выполнять сортировку за линейное время, что, как известно, в общем случае невозможно; впрочем, интересно, что существует алгоритм Торупа (Thorup), который ищет кратчайший путь с оптимальной, линейной, асимптотикой, но основан он на совсем другой идеи, чем алгоритм Дейкстры, поэтому никакого противоречия здесь нет). Однако, Фибоначчиевы кучи довольно сложны в реализации (и, надо отметить, имеют немалую константу, скрытую в асимптотике).

В качестве компромисса можно использовать структуры данных, позволяющие выполнять **оба типа операций** (фактически, это извлечение минимума и обновление элемента) за $O(\log n)$. Тогда время работы алгоритма Дейкстры составит:

$$O(n \log n + m \log n) = O(m \log n)$$

В качестве такой структуры данных программистам на C++ удобно взять стандартный контейнер `set` или `priority_queue`. Первый основан на красно-чёрном дереве, второй — на бинарной куче. Поэтому `priority_queue` имеет меньшую константу, скрытую в асимптотике, однако у него есть и недостаток: он не поддерживает операцию удаления элемента, из-за чего приходится делать "обходной манёвр", который фактически приводит к замене в асимптотике $\log n$ на $\log m$ (с точки зрения асимптотики это на самом деле ничего не меняет, но скрытую константу увеличивает).

Реализация

set

Начнём с контейнера `set`. Поскольку в контейнере нам надо хранить вершины, упорядоченные по их величинам $d[]$, то удобно в контейнер помещать пары: первый элемент пары — расстояние, а второй — номер вершины. В результате в `set` будут храниться пары, автоматически упорядоченные по расстояниям, что нам и нужно.

```
const int INF = 1000000000;

int main() {
    int n;
    ... чтение n ...
    vector<vector<pair<int,int>> g(n);
    ... чтение графа ...
    int s = ...; // стартовая вершина

    vector<int> d(n, INF), p(n);
    d[s] = 0;
    set<pair<int,int>> q;
    q.insert(make_pair(d[s], s));
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());

        for (size_t j=0; j<g[v].size(); ++j) {
            int to = g[v][j].first,
                len = g[v][j].second;
            if (d[v] + len < d[to]) {
                q.erase(make_pair(d[to], to));
                d[to] = d[v] + len;
                p[to] = v;
                q.insert(make_pair(d[to], to));
            }
        }
    }
}
```

В отличие от обычного алгоритма Дейкстры, становится ненужным массив $u[]$. Его роль, как и функцию нахождения вершины с наименьшим расстоянием, выполняет `set`. Изначально в него помещаем стартовую вершину s с её расстоянием. Основной цикл алгоритма выполняется, пока в очереди есть хоть одна вершина. Из очереди извлекается вершина с наименьшим расстоянием, и затем из неё выполняются релаксации. Перед выполнением каждой успешной релаксации мы сначала удаляем из `set` старую пару, а затем, после выполнения релаксации, добавляем обратно новую пару (с новым расстоянием $d[to]$).

priority_queue

Принципиально здесь отличий от `set` нет, за исключением того момента, что удалять из `priority_queue` произвольные элементы невозможно (хотя теоретически кучи поддерживают такую операцию, в стандартной библиотеке она не реализована). Поэтому приходится совершать "обходной манёвр": при релаксации просто не будем удалять старые пары из очереди. В результате в очереди могут находиться одновременно несколько пар для одной и той же вершины (но с разными расстояниями). Среди этих пар нас интересует только одна, для которой элемент `first` равен $d[v]$, а все остальные являются фиктивными. Поэтому надо сделать небольшую модификацию: в начале каждой итерации, когда мы извлекаем из очереди очередную пару, будем проверять, фиктивная она или нет (для этого достаточно сравнить `first` и $d[v]$). Следует отметить, что это важная модификация: если не сделать её, то это приведёт к значительному ухудшению асимптотики (до $O(nm)$).

Ещё нужно помнить о том, что `priority_queue` упорядочивает элементы по убыванию, а не по возрастанию, как обычно. Проще всего преодолеть эту особенность не указанием своего оператора сравнения, а просто помещая в качестве элементов `first` расстояния со знаком минус. В результате в корне кучи будут оказываться элементы с наименьшим расстоянием, что нам и нужно.

```
const int INF = 1000000000;

int main() {
```

```

int n;
... чтение n ...
vector < vector < pair<int,int> > > g (n);
... чтение графа ...
int s = ...; // стартовая вершина

vector<int> d (n, INF), p (n);
d[s] = 0;
priority_queue < pair<int,int> > q;
q.push (make_pair (0, s));
while (!q.empty ()) {
    int v = q.top ().second, cur_d = -q.top ().first;
    q.pop ();
    if (cur_d > d[v]) continue;

    for (size_t j=0; j<g[v].size (); ++j) {
        int to = g[v][j].first,
            len = g[v][j].second;
        if (d[v] + len < d[to]) {
            d[to] = d[v] + len;
            p[to] = v;
            q.push (make_pair (-d[to], to));
        }
    }
}
}

```

Как правило, на практике версия с `priority_queue` оказывается несколько быстрее версии с `set`.

Избавление от pair

Можно ещё немного улучшить производительность, если в контейнерах всё же хранить не пары, а только номера вершин. При этом, понятно, надо перегрузить оператор сравнения для вершин: сравнивать две вершины надо по расстояниям до них `d[]`.

Поскольку в результате релаксации величина расстояния до какой-то вершины меняется, то надо понимать, что "сама по себе" структура данных не перестроится. Поэтому, хотя может показаться, что удалять/добавлять элементы в контейнер в процессе релаксации не надо, это приведёт к разрушению структуры данных. По-прежнему перед релаксацией надо удалить из структуры данных вершину `to`, а после релаксации вставить её обратно — тогда никакие соотношения между элементами структуры данных не нарушаются.

А поскольку удалять элементы можно из `set`, но нельзя из `priority_queue`, то получается, что этот приём применим только к `set`. На практике он заметно увеличивает производительность, особенно когда для хранения расстояний используются большие типы данных (как `long long` или `double`).

Алгоритм Форда-Беллмана

С помощью алгоритма Форда-Беллмана можно найти кратчайшие пути между заданной вершиной и всеми остальными вершинами за $O(NM)$.

В отличие от алгоритма Дейкстры, этот алгоритм применим и к графикам, содержащим отрицательные рёбра. Впрочем, если график содержит отрицательный цикл, то, понятно, результат алгоритма будет неопределён (несмотря на это, алгоритм успешно используется при решении задачи "Отрицательный цикл").

Алгоритм

Пусть массив D — результирующий массив, который будет вычислен окончательно к концу алгоритма. Пусть также N — число вершин, S — номер стартовой вершины. Сначала заполним D значениями "бесконечность" (например, миллиард), за исключением стартовой вершины, в которой $D[S] = 0$. Выполним $N-1$ итерацию алгоритма, в каждой из которых будем пытаться улучшить ответ — уменьшить значения в массиве. Для этого мы просто перебираем все рёбра и каждым ребром пробуем улучшить длину пути. После выполнения всех $N-1$ итераций, массив D будет содержать искомые значения (если, конечно, график не содержит отрицательных циклов).

Реализация

Код, выполняющий алгоритм Форда-Беллмана из вершины 1:

```

typedef pair<int,int> rib;
typedef vector<rib> graph_line;
typedef graph_line::iterator graph_iter;
typedef vector<graph_line> graph;

const int inf = 1000*1000*1000;

int main()

```

```

{
int n;
graph g;
... чтение графа ...

vector<long long> d (n, inf);
d[0] = 0;
vector<int> from (n, -1);
from[0] = 0;
for (int count=1; count<n; count++)
{
    bool anychanged = false;
    for (int v=0; v<n; v++)
        if (d[v] != inf)
            for (graph_iter i=g[v].begin(); i!=g[v].end(); ++i)
            {
                int to = i->first, l = i->second;
                if (d[to] > d[v]+l)
                {
                    d[to] = d[v]+l;
                    from[to] = v;
                    anychanged = true;
                }
            }
    if (!anychanged) break;
}
... вывод массива d ...
}

```

Во-первых, этот код использует небольшую оптимизацию алгоритма: все $N-1$ итерации не обязательно выполняются, если обнаружено, что уже на каком-то более раннем шаге ответ уже получен. Действительно, если на некотором шаге мы не улучшили D ни в одной вершине, то, очевидно, и все последующие итерации ничего не смогут улучшить. Следовательно, можно остановить алгоритм прямо на этом шаге. На самом деле, эта оптимизация в некоторых случаях позволяет очень существенно снизить время работы алгоритма.

Во-вторых, представленный код строит не только массив расстояний D , но и массив предков $from$. По окончании работы алгоритма в каждом элементе $from[i]$ будет содержаться "предок" вершины i , т.е. $from[i]$ - это вершина, из которой мы пришли в вершину i , двигаясь по кратчайшему пути. Следовательно, имея массив $from$, мы можем теперь и **восстановить** собственно путь до любой вершины:

```

int v = n-1; // это вершина, путь до которой нам нужно вывести. например, n-1
vector<int> path;
while (true)
{
    path.push_back (v);
    if (v == from[v]) break;
    v = from[v];
}
reverse (path.begin(), path.end()); // путь получится сначала задом наперёд
for (size_t i=0; i<path.size(); ++i)
    cout << path[i] << ' ';

```

Алгоритм Левита нахождения кратчайших путей от заданной вершины до всех остальных вершин за $O(NM)$

Пусть дан граф с N вершинами и M ребрами, для каждого из которых указан его вес L_i . Также дана стартовая вершина V_0 . Требуется найти кратчайшие пути от вершины V_0 до всех остальных вершин.

Алгоритм Левита решает эту задачу весьма эффективно (по поводу асимптотики и скорости работы см. ниже).

Описание

Пусть массив $D[1..N]$ будет содержать текущие кратчайшие длины путей, т.е. D_i - это текущая длина кратчайшего пути от вершины V_0 до вершины i . Изначально массив D заполнен значениями "бесконечность", кроме $D_{V_0} = 0$. По окончании работы алгоритма этот массив будет содержать окончательные кратчайшие расстояния.

Пусть массив $P[1..N]$ содержит текущих предков, т.е. P_i - это вершина, предшествующая вершине i в кратчайшем пути от вершины V_0 до i . Так же как и массив D , массив P изменяется постепенно по ходу алгоритма и к концу его принимает окончательные значения.

Теперь собственно сам алгоритм Левита. На каждом шаге поддерживается три множества вершин:

- M_0 - вершины, расстояние до которых уже вычислено (но, возможно, не окончательно);
- M_1 - вершины, расстояние до которых вычисляется;
- M_2 - вершины, расстояние до которых ещё не вычислено.

Вершины в множестве M_1 хранятся в виде двунаправленной очереди (deque).

Изначально все вершины помещаются в множество M_2 , кроме вершины V_0 , которая помещается в множество M_1 .

На каждом шаге алгоритма мы берём вершину из множества M_1 (достаём верхний элемент из очереди). Пусть V - это выбранная вершина. Переводим эту вершину во множество M_0 . Затем просматриваем все рёбра, выходящие из этой вершины. Пусть T - это второй конец текущего ребра (т.е. не равный V), а L - это длина текущего ребра.

- Если T принадлежит M_2 , то T переносим во множество M_1 в конец очереди. D_T полагаем равным $D_V + L$.
- Если T принадлежит M_1 , то пытаемся улучшить значение D_T : $D_T = \min(D_T, D_V + L)$. Сама вершина T никак не передвигается в очереди.
- Если T принадлежит M_0 , и если D_T можно улучшить ($D_T > D_V + L$), то улучшаем D_T , а вершину T возвращаем в множество M_1 , помешав её в начало очереди.

Разумеется, при каждом обновлении массива D следует обновлять и значение в массиве P .

Подробности реализации

Создадим массив $ID[1..N]$, в котором для каждой вершины будем хранить, какому множеству она принадлежит: 0 - если M_2 (т.е. расстояние равно бесконечности), 1 - если M_1 (т.е. вершина находится в очереди), и 2 - если M_0 (некоторый путь уже был найден, расстояние меньше бесконечности).

Очередь обработки можно реализовать стандартной структурой данных deque. Однако есть более эффективный способ. Во-первых, очевидно, в очереди в любой момент времени будет храниться максимум N элементов. Но, во-вторых, мы можем добавлять элементы и в начало, и в конец очереди. Следовательно, мы можем организовать очередь на массиве размера N , однако нужно зациклить его. Т.е. делаем массив $Q[1..N]$, указатели (int) на первый элемент QH и на элемент после последнего QT . Очередь пуста, когда $QH == QT$. Добавление в конец - просто запись в $Q[QT]$ и увеличение QT на 1; если QT после этого вышел за пределы очереди ($QT == N$), то делаем $QT = 0$. Добавление в начало очереди - уменьшаем QH на 1, если она вышла за пределы очереди ($QH == -1$), то делаем $QH = N-1$.

Сам алгоритм реализуем в точности по описанию выше.

Асимптотика

Мне не известна более-менее хорошая асимптотическая оценка этого алгоритма. Я встречал только оценку $O(NM)$ у похожего алгоритма.

Однако на практике алгоритма зарекомендовал себя очень хорошо: время его работы я оцениваю как $O(M \log N)$, хотя, повторюсь, это исключительно экспериментальная оценка.

Реализация

```
typedef pair<int,int> rib;
typedef vector < vector<rib> > graph;

const int inf = 1000*1000*1000;

int main()
{
    int n, v1, v2;
    graph g (n);

    ... чтение графа ...

    vector<int> d (n, inf);
    d[v1] = 0;
    vector<int> id (n);
    deque<int> q;
    q.push_back (v1);
    vector<int> p (n, -1);

    while (!q.empty())
    {
        int v = q.front(), q.pop_front();
        id[v] = 1;
        for (size_t i=0; i<g[v].size(); ++i)
        {
            int to = g[v][i].first, len = g[v][i].second;
            if (d[to] > d[v] + len)
            {
                d[to] = d[v] + len;
                if (id[to] == 0)
                    q.push_back (to);
                else if (id[to] == 1)
                    q.push_front (to);
                p[to] = v;
                id[to] = 1;
            }
        }
    }
}
```

```
    }
}

... вывод результата ...

}
```

Алгоритм Флойда-Уоршола

Авторы алгоритма: Роберт Флойд (Robert Floyd) и Стивен Уоршол (Stephen Warshall), которые (независимо друг от друга?) разработали его в 1962 году.

С помощью алгоритма Флойда-Уоршола можно найти кратчайшие пути между всему парами вершин за $O(N^3)$, где N - количество вершин в графе.

На вход подаётся любой граф (ориентированный или неориентированный), заданный своей матрицей смежности. На выходе будет матрица $D[1..N][1..N]$, в каждом элементе $D[i][j]$ которой будет длина кратчайшего пути между вершинами i и j .

Алгоритм

Изначально присвоим матрице D матрицу смежности графа (однако, если две вершины не связаны ребром, то соответствующие значения в матрице сделаем равными бесконечности). Теперь будет перебирать все K от 1 до N , а затем для каждого K перебирать всевозможные пары вершин (i, j) и пытаться уменьшить длину пути между вершинами i и j с помощью вершины K (т.е. будем брать путь из i в K и из K в j и пытаться этим путём улучшить путь между i и j). По окончании этого процесса мы получим искомую матрицу D .

Код

```
const int INF = 100*1000*1000;

int main()
{
    // считываем матрицу графа
    int n;
    cin >> n;
    vector<vector<int>> g (n, vector<int> (n));
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
    {
        int t;
        cin >> t;
        g[i][j] = t ? t : INF;
    }

    // собственно алгоритм
    // храним две матрицы: для текущего шага и от предыдущего шага
    vector<vector<int>> d (n), d2;
    d2 = g;
    for (int i=0; i<n; i++)
        d[i].resize (n+1);
    for (int k=0; k<n; k++)
    {
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                d[i][j] = min (d2[i][j], d2[i][k]+d2[k][j]);
        d.swap (d2);
    }
    d.swap (d2);

    // выводим результат
    for (int i=0; i<n; i++)
    {
        for (int j=0; j<n; j++)
            cout << (d[i][j]<INF ? d[i][j] : 0) << ' ';
        cout << endl;
    }
}
```

Минимальное оствовное дерево. Алгоритм Прима

Дан взвешенный неориентированный граф. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим весом (т.е. суммой весов рёбер) из всех возможных. Такое поддерево называется минимальным оствовым деревом или простом минимальным оством.

Здесь будут рассмотрены несколько важных фактов, связанных с минимальными оствами, затем будет рассмотрен алгоритм Прима в его простейшей реализации, а затем слегка улучшенный вариант.

Свойства минимального оства

- Минимальный остав **уникален, если веса всех рёбер различны**. В противном случае, может существовать несколько минимальных оствов (конкретные алгоритмы обычно получают один из возможных оствов).
- Минимальный остав является также и **остовом с минимальным произведением весов рёбер**. (доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)
- Минимальный остав является также и **остовом с минимальным весом самого тяжелого ребра**. (это утверждение следует из справедливости [алгоритма Крускала](#))
- **Остав максимального веса** является аналогично оству минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритмов минимального оства.

Алгоритм Прима

Этот алгоритм был описан в статье Прима (Prim) в 1957 г., хотя этот алгоритм был открыт ещё в 1930 г. Ярником (Jarnik).

Алгоритм Прима постепенно строит искомый минимальный остав, добавляя в него по одному ребру на каждом шаге.

(Это означает, что алгоритм Прима является жадным. Более того, справедливость алгоритма Прима легко устанавливается в рамках теории матроидов.)

В начале работы алгоритма результирующее дерево состоит из одной вершины (её можно выбирать произвольно).

Алгоритм состоит из $N-1$ итерации, на каждой из которых к дереву добавляется ровно одно ребро, не нарушающее свойства дерева (т.е. один конец добавляемого ребра принадлежит дереву, а другой - не принадлежит). Ключевой момент - из всех таких рёбер каждый раз выбирается ребро с минимальным весом.

Простейшая реализация

Этот код самым непосредственным образом реализует описанный выше алгоритм, и выполняется за $O(NM)$.

```
int n;
vector<vector<pair<int,int>>> g; // пары вершина-вес

... чтение графа ...

vector<char> used (n);
used[0] = true;
vector<pair<int,int>> result;
for (int i=0; i<n-1; ++i)
{
    int minv = -1; size_t minr;
    for (int j=0; j<n; ++j)
        if (used[j])
            for (size_t k=0; k<g[j].size(); ++k)
                if (!used[g[j][k].first])
                    if (minv == -1 || g[j][k].second < g[minv][minr].second)
                        minv = j, minr = k;
    used[g[minv][minr].first] = true;
    result.push_back (make_pair (minv, g[minv][minr].first));
}

cout << "Result (all ribs):\n";
for (int i=0; i<n-1; ++i)
    cout << result[i].first << ' ' << result[i].second << '\n';
```

Улучшенная реализация

Эта реализация будет выполнять заметно быстрее - за $O(M \log N + N^2)$.

Для этого мы отсортируем все рёбра в списках смежности каждой вершины по увеличению веса (потребуется $O(M \log M) = O(M \log N)$). Кроме того, для каждой вершины заведем указатель, указывающий на первое доступное ребро в её списке смежности. Изначально все указатели указывают на начала списков, т.е. равны 0. На i -ой итерации алгоритма Прима мы перебираем все вершины, и выбираем наименьшее по весу ребро среди доступных. Поскольку всё рёбра уже отсортированы по весу, а указатели указывают на первые доступные рёбра, то выбор наименьшего ребра осуществляется за $O(N)$. Теперь нам следует обновить указатели, поскольку некоторые из них указывают на ставшие недоступными рёбра (оба конца которых оказались внутри дерева), т.е. сдвинуть некоторые из них вправо. Однако, поскольку во всех списках смежности в сумме $2 * M$ элементов, а указатели сдвигаются только вправо, то получается, что на поддержание всех

указателей потребуется $O(M)$ действий. Итого - время выполнения алгоритма $O(M \log M + N^2 + M)$, т.е. $O(M \log N + N^2)$.

```
int n;
vector < vector < pair<int,int> > > g; // пары вес-вершина
... чтение графа ...
for (int i=0; i<n; ++i)
    sort (g[i].begin(), g[i].end());
vector<char> used (n);
used[0] = true;
vector < pair<int,int> > result;
vector<size_t> ptr (n);
for (int i=0; i<n-1; ++i)
{
    int minv = -1;
    for (int j=0; j<n; ++j)
        if (used[j])
            if (ptr[j] < g[j].size())
                if (minv == -1 || g[j][ptr[j]].first < g[minv][ptr[minv]].first)
                    minv = j;
    used [ g[minv][ptr[minv]].second ] = true;
    result.push_back (make_pair (minv, g[minv][ptr[minv]].second));
    for (int j=0; j<n; ++j)
        while (ptr[j] < g[j].size() && used [ g[j][ptr[j]].second ])
            ++ptr[j];
}
cout << "Result (all ribs):\n";
for (int i=0; i<n-1; ++i)
    cout << result[i].first << ' ' << result[i].second << '\n';
```

Нахождение максимального остова

Как уже говорилось выше, нахождение максимального остова можно свести к нахождению минимального остова, если заменить веса всех рёбер на противоположные. Однако в случае алгоритма Прима можно поступить ещё проще - достаточно просто изменить порядок сортировки на обратный, и операцию '`<`' на '`>`' при сравнении весов рёбер.

Минимальное оствовное дерево. Алгоритм Крускала

Дан взвешенный неориентированный граф. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим весом (т.е. суммой весов рёбер) из всех возможных. Такое поддерево называется минимальным оствовым деревом или простым минимальным оством.

Здесь будут рассмотрены несколько важных фактов, связанных с минимальными оствами, затем будет рассмотрен алгоритм Крускала в его простейшей реализации.

Свойства минимального оства

- Минимальный остав **уникален, если веса всех рёбер различны**. В противном случае, может существовать несколько минимальных оствов (конкретные алгоритмы обычно получают один из возможных оствов).
- Минимальный остав является также и **остовом с минимальным произведением весов рёбер**. (доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)
- Минимальный остав является также и **остовом с минимальным весом самого тяжелого ребра**. (это утверждение следует из справедливости алгоритма Крускала)
- **Остав максимального веса** ищется аналогично оству минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритмов минимального оства.

Алгоритм Крускала

Данный алгоритм был описан Крускалом (Kruskal) в 1956 г.

Алгоритм Крускала изначально помещает каждую вершину в своё дерево, а затем постепенно объединяет эти деревья, объединяя на каждой итерации два некоторых дерева некоторым ребром. Перед началом выполнения алгоритма, все рёбра сортируются по весу (в порядке неубывания). Затем начинается процесс объединения: перебираются все рёбра от первого до последнего (в порядке сортировки), и если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются, а ребро добавляется к ответу. По окончании перебора всех рёбер все вершины окажутся принадлежащими одному поддереву, и ответ найден.

Простейшая реализация

Этот код самым непосредственным образом реализует описанный выше алгоритм, и выполняется за $O(M \log N + N^2)$. Сортировка рёбер потребует $O(M \log N)$ операций. Принадлежность вершины тому или иному поддереву хранится просто с помощью массива `tree_id` - в нём для каждой вершины хранится номер дерева, которому она принадлежит. Для каждого ребра мы за $O(1)$ определяем, принадлежат ли его концы разным деревьям. Наконец, объединение двух деревьев осуществляется за $O(N)$ простым проходом по массиву `tree_id`. Учитывая, что всего операций объединения будет $N-1$, мы и получаем асимптотику $O(M \log N + N^2)$.

```
int m;
vector<pair<int, pair<int,int>>> g (m); // вес - вершина 1 - вершина 2

int cost = 0;
vector<pair<int,int>> res;

sort (g.begin(), g.end());
vector<int> tree_id (n);
for (int i=0; i<n; ++i)
    tree_id[i] = i;
for (int i=0; i<m; ++i)
{
    int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
    if (tree_id[a] != tree_id[b])
    {
        cost += l;
        res.push_back (make_pair (a, b));
        int old_id = tree_id[b], new_id = tree_id[a];
        for (int j=0; j<n; ++j)
            if (tree_id[j] == old_id)
                tree_id[j] = new_id;
    }
}
```

Улучшенная реализация

С использованием структуры данных "Система непересекающихся множеств" можно написать более быструю реализацию алгоритма Крускала с асимптотикой $O(M \log N)$.

Минимальное оставное дерево. Алгоритм Крускала с системой непересекающихся множеств

Постановку задачи и описание алгоритма Крускала см. [здесь](#).

Здесь будет рассмотрена реализация с использованием структуры данных "система непересекающихся множеств" (DSU), которая позволит достигнуть асимптотики $O(M \log N)$.

Описание

Так же, как и в простой версии алгоритма Крускала, отсортируем все рёбра по неубыванию веса. Затем поместим каждую вершину в своё дерево (т.е. своё множество) с помощью вызова функции DSU `MakeSet` - на это уйдёт в сумме $O(N)$. Перебираем все рёбра (в порядке сортировки) и для каждого ребра за $O(1)$ определяем, принадлежат ли его концы разным деревьям (с помощью двух вызовов `FindSet` за $O(1)$). Наконец, объединение двух деревьев будет осуществляться вызовом `Union` - также за $O(1)$. Итого мы получаем асимптотику $O(M \log N + N + M) = O(M \log N)$.

Реализация

Для уменьшения объёма кода реализуем все операции не в виде отдельных функций, а прямо в коде алгоритма Крускала.

Здесь будет использоваться рандомизированная версия DSU.

```
vector<int> p (n);

int dsu_get (int v) {
    return (v == p[v]) ? v : (p[v] = dsu_get (p[v]));
}

void dsu_unite (int a, int b) {
    a = dsu_get (a);
    b = dsu_get (b);
    if (rand() & 1)
        swap (a, b);
    if (a != b)
        p[a] = b;
```

```

}

... в функции main(): ...

int m;
vector < pair < int, pair<int,int> > > g; // вес - вершина 1 - вершина 2
... чтение графа ...

int cost = 0;
vector < pair<int,int> > res;

sort (g.begin(), g.end());
p.resize (n);
for (int i=0; i<n; ++i)
p[i] = i;
for (int i=0; i<m; ++i) {
int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
if (dsu_get(a) != dsu_get(b)) {
cost += l;
res.push_back (g[i].second);
dsu_unite (a, b);
}
}

```

Матричная теорема Кирхгофа. Нахождение количества остовных деревьев

Задан связный неориентированный граф своей матрицей смежности. Кратные рёбра в графе допускаются. Требуется посчитать количество различных остовных деревьев этого графа.

Приведённая ниже формула принадлежит Кирхгофу (Kirchhoff), который доказал её в 1847 г.

Матричная теорема Кирхгофа

Возьмём матрицу смежности графа G , заменим каждый элемент этой матрицы на противоположный, а на диагонале вместо элемента $A_{i,i}$ поставим степень вершины i (если имеются кратные рёбра, то в степени вершины они учитываются со своей кратностью). Тогда, согласно матричной теореме Кирхгофа, все алгебраические дополнения этой матрицы равны между собой, и равны количеству остовных деревьев этого графа. Например, можно удалить последнюю строку и последний столбец этой матрицы, и модуль её определителя будет равен искомому количеству.

Определитель матрицы можно найти за $O(N^3)$ с помощью метода Гаусса или метода Крауга.

Доказательство этой теоремы достаточно сложно и здесь не приводится (см., например, Приезжев В.Б. "Задача о димерах и теорема Кирхгофа").

Связь с законами Кирхгофа в электрической цепи

Между матричной теоремой Кирхгофа и законами Кирхгофа для электрической цепи имеется удивительная связь.

Можно показать (как следствие из закона Ома и первого закона Кирхгофа), что сопротивление R_{ij} между точками i и j электрической цепи равно:

$$R_{ij} = |T^{(i,j)}| / |T^j|$$

где матрица T получена из матрицы A обратных сопротивлений проводников (A_{ij} - обратное число к сопротивлению проводника между точками i и j) преобразованием, описанным в матричной теореме Кирхгофа, а обозначение $T^{(i)}$ обозначает вычёркивание строки и столбца с номером i , а $T^{(i,j)}$ - вычёркивание двух строк и столбцов i и j .

Теорема Кирхгофа придаёт этой формуле геометрический смысл.

Нахождение отрицательного цикла

Для решения этой задачи мы воспользуемся [алгоритмом Форда-Беллмана](#), т.е. эту задачу мы решим за $O(NM)$.

Известно, что алгоритм Форда-Беллмана в "чистом" виде некорректно работает на графе с отрицательным циклом. Тем не менее, результатом работы этого алгоритма можно воспользоваться для нахождения этого отрицательного цикла.

Алгоритм

Выполняем все $N-1$ итерации алгоритма Форда-Беллмана, сохраняя дополнительно массив предков. Затем дополнительно выполняем ещё одну итерацию. Если на ней оказывается, что какое-то значение массива D было улучшено в некоторой вершине X , то в графе имеется отрицательный цикл, иначе отрицательного цикла нет. Далее, чтобы найти сам цикл, пройдёмся по массиву предков от вершины X до тех пор, пока не зациклимся - этот цикл и будет искомым отрицательным циклом (следует заметить, что сама вершина X далеко не всегда входит в этот цикл).

Код

```
typedef pair<int,int> rib;
typedef vector<rib> graf_line;
typedef graf_line::iterator graf_iter;
typedef vector<graf_line> graf;

const int inf = 1000*1000*1000;

int main()
{
    int n;
    graf g;
    ... чтение графа ...

    vector<long long> d (n, inf);
    d[0] = 0;
    vector<int> from (n, -1);
    from[0] = 0;
    bool anychanged;
    int firstchanged;
    for (int count=1; count<=n; count++)
    {
        anychanged = false;
        for (int v=0; v<n; v++)
            if (d[v] < inf)
                for (graf_iter i=g[v].begin(); i!=g[v].end(); ++i)
                {
                    int to = i->first, l = i->second;
                    if (d[to] > d[v]+1)
                    {
                        d[to] = d[v]+1;
                        from[to] = v;
                        if (!anychanged)
                        {
                            anychanged = true;
                            firstchanged = to;
                        }
                    }
                }
        if (!anychanged)
            puts ("NO");
        else
        {
            puts ("YES");

            vector<int> path;
            path.reserve (n+1);
            path.push_back (firstchanged);
            vector<char> used (n);
            used[firstchanged] = true;
            int last = firstchanged;
            for (int cur=from[firstchanged]; !used[cur]; last=cur=from[cur])
            {
                path.push_back (cur);
                used[cur] = true;
            }
            path.push_back (last);
            path.erase (path.begin(), find (path.begin(), path.end(), last));
            reverse (path.begin(), path.end());

            printf ("%d\n", (int)path.size());
            for (size_t i=0; i<path.size(); i++)
```

```
    printf ("%d ", path[i]+1);
}
}
```

Нахождение Эйлерова пути за O (M)

Эйлеров путь - это путь в графе, проходящий через все его рёбра. Эйлеров цикл - это эйлеров путь, являющийся циклом.

Задача заключается в том, чтобы найти эйлеров путь в **неориентированном мультиграфе с петлями**.

Алгоритм

Сначала проверим, существует ли эйлеров путь. Затем найдём все простые циклы и объединим их в один - это и будет эйлеровым циклом. Если граф таков, что эйлеров путь не является циклом, то, добавим недостающее ребро, найдём эйлеров цикл, потом удалим лишнее ребро.

Чтобы проверить, существует ли эйлеров путь, нужно воспользоваться следующей теоремой. Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны. Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум (или нулю, в случае существования эйлерова цикла).

Кроме того, конечно, граф должен быть достаточно связным (т.е. если удалить из него все изолированные вершины, то должен получиться связный граф).

Искать все циклы и объединять их будем одной рекурсивной процедурой:

```
procedure FindEulerPath (V)
1. перебрать все рёбра, выходящие из вершины V;
   каждое такое ребро удаляем из графа, и
   вызываем FindEulerPath из второго конца этого ребра;
2. добавляем вершину V в ответ.
```

Сложность этого алгоритма, очевидно, является линейной относительно числа рёбер.

Но этот же алгоритм мы можем записать в **нерекурсивном** варианте:

```
stack St;
в St кладём любую вершину (стартовая вершина);
пока St не пустой
  пусть V - значение на вершине St;
  если степень(V) = 0, то
    добавляем V к ответу;
    снимаем V с вершины St;
  иначе
    находим любое ребро, выходящее из V;
    удаляем его из графа;
    второй конец этого ребра кладём в St;
```

Несложно проверить эквивалентность этих двух форм алгоритма. Однако вторая форма, очевидно, быстрее работает, причём кода будет не больше.

Задача о домино

Приведём здесь классическую задачу на эйлеров цикл - задачу о домино.

Имеется N доминошек, как известно, на двух концах доминошки записано по одному числу (обычно от 1 до 6, но в нашем случае не важно). Требуется выложить все доминошки в ряд так, чтобы у любых двух соседних доминошек числа, записанные на их общей стороне, совпадали. Доминошки разрешается переворачивать.

Переформулируем задачу. Пусть числа, записанные на доминошках, - вершины графа, а доминошки - рёбра этого графа (каждая доминошка с числами (a,b) - это ребра (a,b) и (b,a)). Тогда наша задача **сводится к** задаче нахождения **эйлерова пути** в этом графе.

Реализация

Приведенная ниже программа ищет и выводит эйлеров цикл или путь в графе, или выводит -1, если его не существует.

Сначала программа проверяет степени вершин: если вершин с нечётной степенью нет, то в графе есть эйлеров цикл, если есть 2 вершины с нечётной степенью, то в графе есть только эйлеров путь (эйлерова цикла нет), если же таких вершин больше 2, то в графе нет ни эйлерова цикла, ни эйлерова пути. Чтобы найти эйлеров путь (не цикл), поступим таким образом: если V1 и V2 - это две вершины нечётной степени, то просто добавим ребро (V1,V2), в полученном графе найдём эйлеров цикл (он, очевидно, будет существовать), а затем удалим из ответа "фактивное" ребро (V1,V2). Эйлеров цикл будем искать в точности так, как описано выше (нерекурсивной версией), и заодно по окончании этого алгоритма проверим, связный был граф или нет (если граф был не связный, то по окончании работы алгоритма в графе останутся некоторые рёбра, и в этом случае нам надо вывести -1). Наконец, программа учитывает, что в графе могут быть изолированные вершины.

```
int main() {

    int n;
    vector < vector<int> > g (n, vector<int> (n));
    ... чтение графа в матрицу смежности ...

    vector<int> deg (n);
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            deg[i] += g[i][j];

    int first = 0;
    while (!deg[first]) ++first;

    int v1 = -1, v2 = -1;
    bool bad = false;
    for (int i=0; i<n; ++i)
        if (deg[i] & 1)
            if (v1 == -1)
                v1 = i;
            else if (v2 == -1)
                v2 = i;
            else
                bad = true;

    if (v1 != -1)
        ++g[v1][v2], ++g[v2][v1];

    stack<int> st;
    st.push (first);
    vector<int> res;
    while (!st.empty())
    {
        int v = st.top();
        int i;
        for (i=0; i<n; ++i)
            if (g[v][i])
                break;
        if (i == n)
        {
            res.push_back (v);
            st.pop();
        }
        else
        {
            --g[v][i];
            --g[i][v];
            st.push (i);
        }
    }

    if (v1 != -1)
        for (size_t i=0; i+1<res.size(); ++i)
            if (res[i] == v1 && res[i+1] == v2 || res[i] == v2 && res[i+1] == v1)
            {
                vector<int> res2;
                for (size_t j=i+1; j<res.size(); ++j)
                    res2.push_back (res[j]);
                for (size_t j=1; j<=i; ++j)
                    res2.push_back (res[j]);
                res = res2;
                break;
            }

        for (int i=0; i<n; ++i)
            for (int j=0; j<n; ++j)
                if (g[i][j])
                    bad = true;

        if (bad)
            puts ("-1");
        else
            for (size_t i=0; i<res.size(); ++i)
                printf ("%d ", res[i]+1);
    }
}
```

Проверка графа на ацикличность и нахождение цикла

Пусть дан ориентированный или неориентированный граф без петель и кратных рёбер. Требуется проверить, является ли он ациклическим, а если не является, то найти любой цикл.

Решим эту задачу с помощью [поиска в глубину](#) за $O(M)$.

Алгоритм

Произведём серию поисков в глубину в графе. Т.е. из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе - в чёрный. И если поиск в глубину пытается пойти в серую вершину, то это означает, что мы нашли цикл (если граф неориентированный, то случаи, когда поиск в глубину из какой-то вершины пытается пойти в предка, не считаются).

Сам цикл можно восстановить проходом по массиву предков.

Реализация

Здесь приведена реализация для случая ориентированного графа.

```
int n;
vector<vector<int>> g;
vector<char> cl;
vector<int> p;
int cycle_st, cycle_end;

bool dfs(int v) {
    cl[v] = 1;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (cl[to] == 0) {
            p[to] = v;
            if (dfs(to)) return true;
        }
        else if (cl[to] == 1) {
            cycle_end = v;
            cycle_st = to;
            return true;
        }
    }
    cl[v] = 2;
    return false;
}

int main() {
    ... чтение графа ...

    p.assign(n, -1);
    cl.assign(n, 0);
    cycle_st = -1;
    for (int i=0; i<n; ++i)
        if (dfs(i))
            break;

    if (cycle_st == -1)
        puts("Acyclic");
    else {
        puts("Cyclic");
        vector<int> cycle;
        cycle.push_back(cycle_st);
        for (int v=cycle_end; v!=cycle_st; v=p[v])
            cycle.push_back(v);
        reverse(cycle.begin(), cycle.end());
        for (size_t i=0; i<cycle.size(); ++i)
            printf("%d ", cycle[i]+1);
    }
}
```

Наименьший общий предок. Нахождение за $O(\sqrt{N})$ и $O(\log N)$ с препроцессингом $O(N)$

Пусть дано дерево G. На вход поступают запросы вида (V_1, V_2) , для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V, которая лежит на пути от корня до V_1 , на пути от корня до V_2 , и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V_1 , и V_2 , и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V_1 и V_2 - это их общий предок, лежащий на кратчайшем пути из V_1 в V_2 . В частности, например, если V_1 является предком V_2 , то V_1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Идея алгоритма

Перед тем, как отвечать на запросы, выполним так называемый **препроцессинг**. Запустим обход в глубину из корня, который будет строить список посещения вершин Order (текущая вершина добавляется в список при входе в эту вершину, а также после каждого возвращения из её сына), нетрудно заметить, что итоговый размер этого списка будет $O(N)$. И построим массив First[1..N], в котором для каждой вершины будет указана позиция в массиве Order, в которой стоит эта вершина, т.е. $Order[First[i]] = i$ для всех i . Также с помощью поиска в глубину найдём высоту каждой вершины (расстояние от корня до неё) - H[1..N].

Как теперь отвечать на запросы? Пусть имеется текущий запрос - пара вершин V_1 и V_2 . Рассмотрим список Order между индексами $First[V_1]$ и $First[V_2]$. Нетрудно заметить, что в этом диапазоне будет находиться искомое LCA (V_1, V_2), а также множество других вершин. Однако LCA (V_1, V_2) будет отличаться от остальных вершин тем, что это будет вершина с наименьшей высотой.

Таким образом, чтобы ответить на запрос, нам нужно просто **найти вершину с наименьшей высотой** в массиве Order в диапазоне между $First[V_1]$ и $First[V_2]$. Таким образом, задача LCA сводится к задаче RMQ ("минимум на отрезке"). А последняя задача решается с помощью структур данных (см. задача RMQ).

Если использовать **sqrt-декомпозицию**, то можно получить решение, отвечающее на запрос за $O(\sqrt{N})$ и выполняющее препроцессинг за $O(N)$.

Если использовать **дерево отрезков**, то можно получить решение, отвечающее на запрос за $O(\log N)$ и выполняющее препроцессинг за $O(N)$.

Реализация

Здесь будет приведена готовая реализация LCA с использованием дерева отрезков:

```
typedef vector<vector<int>> graph;
typedef vector<int>::const_iterator const_graph_iter;

vector<int> lca_h, lca_dfs_list, lca_first, lca_tree;
vector<char> lca_dfs_used;

void lca_dfs (const graph & g, int v, int h = 1)
{
    lca_dfs_used[v] = true;
    lca_h[v] = h;
    lca_dfs_list.push_back (v);
    for (const_graph_iter i = g[v].begin(); i != g[v].end(); ++i)
        if (!lca_dfs_used[*i])
        {
            lca_dfs (g, *i, h+1);
            lca_dfs_list.push_back (v);
        }
}

void lca_build_tree (int i, int l, int r)
{
    if (l == r)
        lca_tree[i] = lca_dfs_list[l];
    else
    {
        int m = (l + r) >> 1;
        lca_build_tree (i+i, l, m);
        lca_build_tree (i+i+1, m+1, r);
        if (lca_h[lca_tree[i+i]] < lca_h[lca_tree[i+i+1]])
            lca_tree[i] = lca_tree[i+i];
        else
            lca_tree[i] = lca_tree[i+i+1];
    }
}

void lca_prepare (const graph & g, int root)
{
    int n = (int) g.size();
    lca_h.resize (n);
    lca_dfs_list.reserve (n*2);
    lca_dfs_used.assign (n, 0);

    lca_dfs (g, root);
```

```

int m = (int) lca_dfs_list.size();
lca_tree.assign (lca_dfs_list.size() * 4 + 1, -1);
lca_build_tree (1, 0, m-1);

lca_first.assign (n, -1);
for (int i = 0; i < m; ++i)
{
    int v = lca_dfs_list[i];
    if (lca_first[v] == -1)
        lca_first[v] = i;
}

int lca_tree_min (int i, int sl, int sr, int l, int r)
{
    if (sl == l && sr == r)
        return lca_tree[i];
    int sm = (sl + sr) >> 1;
    if (r <= sm)
        return lca_tree_min (i+i, sl, sm, l, r);
    if (l > sm)
        return lca_tree_min (i+i+1, sm+1, sr, l, r);
    int ans1 = lca_tree_min (i+i, sl, sm, l, sm);
    int ans2 = lca_tree_min (i+i+1, sm+1, sr, sm+1, r);
    return lca_h[ans1] < lca_h[ans2] ? ans1 : ans2;
}

int lca (int a, int b)
{
    int left = lca_first[a],
        right = lca_first[b];
    if (left > right) swap (left, right);
    return lca_tree_min (1, 0, (int)lca_dfs_list.size()-1, left, right);
}

int main()
{
    graph g;
    int root;
    ... чтение графа ...

    lca_prepare (g, root);

    for (;;)
    {
        int v1, v2; // поступил запрос
        int v = lca (v1, v2); // ответ на запрос
    }
}

```

Наименьший общий предок. Нахождение за $O(\log N)$ (метод двоичного подъёма)

Пусть дано дерево G. На вход поступают запросы вида (V_1, V_2) , для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V , которая лежит на пути от корня до V_1 , на пути от корня до V_2 , и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V_1 , и V_2 , и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V_1 и V_2 - это их общий предок, лежащий на кратчайшем пути из V_1 в V_2 . В частности, например, если V_1 является предком V_2 , то V_1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Здесь будет рассмотрен алгоритм, который пишется намного быстрее, чем описанный [здесь](#).

Асимптотика полученного алгоритма будет равна: препроцессинг за $O(N \log N)$ и ответ на каждый запрос за $O(\log N)$.

Алгоритм

Предпосчитаем для каждой вершины её 1-го предка, 2-го предка, 4-го, и т.д. Обозначим этот массив через P , т.е. $P[i][j]$ - это 2^j -й предок вершины i , $i = 1..N$, $j = 0..[\log N]$. Так же для каждой вершины найдём времена захода в неё и выхода поиска в глубину (см. "Поиск в глубину") - это нам понадобится, чтобы определять за $O(1)$, является ли одна вершина предком другой (не обязательно непосредственным). Такой препроцессинг можно выполнить за $O(N \log N)$.

Пусть теперь поступил очередной запрос - пара вершин (A,B). Сразу проверим, не является ли одна вершина предком другой - в таком случае она и является результатом. Если A не предок B, и B не предок A, то будем подниматься по предкам A, пока не найдём самую высокую (т.е. наиболее близкую к корню) вершину, которая ещё не является предком (не обязательно непосредственным) B (т.е. такую вершину X, что X не предок B, а P[X][0] - предок B). При этом находить эту вершину X будем за O(log N), пользуясь массивом P.

Опишем этот процесс подробнее. Пусть L = [logN]. Пусть сначала l = L. Если P[A][l] не является предком B, то присваиваем A = P[A][l], и уменьшаем l. Если же P[A][l] является предком B, то просто уменьшаем l. Очевидно, что когда l станет меньше нуля, вершина A как раз и будет являться искомой вершиной - т.е. такой, что A не предок B, но P[A][0] - предок B.

Теперь, очевидно, ответом на LCA будет являться P[A][0] - т.е. наименьшая вершина среди предков исходной вершины A, являющаяся также и предком B.

Асимптотика. Весь алгоритм ответа на запрос состоит из изменения l от L = [logN] до 0, а также проверки на каждом шаге за O(1), является ли одна вершина предком другой. Следовательно, на каждый запрос будет найден ответ за O(log N).

Реализация

```
int n, l;
vector < vector<int> > g;
vector<char> used;
vector<int> tin, tout;
int timer;
vector < vector<int> > up;

void dfs (int v, int p = 0) {
    used[v] = true;
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i=1; i<=l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (!used[to])
            dfs (to, v);
    }
    tout[v] = ++timer;
}

bool upper (int a, int b) {
    return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int lca (int a, int b) {
    if (upper (a, b))    return a;
    if (upper (b, a))    return b;
    for (int i=1; i>=0; --i)
        if (! upper (up[a][i], b))
            a = up[a][i];
    return up[a][0];
}

int main() {
    ... чтение n и g ...

    used.resize (n),  tin.resize (n),  tout.resize (n),  up.resize (n);
    l = 1;
    while ((1<<l) <= n)  ++l;
    for (int i=0; i<n; ++i)  up[i].resize (l+1);
    dfs (0);

    for (;;) {
        int a, b; // текущий запрос
        int res = lca (a, b); // ответ на запрос
    }
}
```

Наименьший общий предок. Нахождение за O (1) с препроцессингом O (N) (алгоритм Фарах-Колтона и Бендера)

Пусть дано дерево G. На вход поступают запросы вида (V1, V2), для каждого запроса требуется найти их наименьшего общего предка, т.е. вершину V, которая лежит на пути от корня до V1, на пути от корня до V2, и из всех таких вершин следует выбирать самую нижнюю. Иными словами, искомая вершина V - предок и V1, и V2, и среди всех таких общих предков выбирается нижний. Очевидно, что наименьший общий предок вершин V1 и V2 - это их общий предок, лежащий на кратчайшем пути из V1 в V2. В частности, например, если V1 является предком V2, то V1 является их наименьшим общим предком.

На английском эта задача называется задачей LCA - Least Common Ancestor.

Описываемый здесь алгоритм Фарах-Колтона и Бендер (Farach-Colton, Bender) является асимптотически оптимальным, и при этом сравнительно простым (по сравнению с другими алгоритмами, например, Шибера-Вишкина).

Алгоритм

Воспользуемся классическим сведением задачи LCA к задаче RMQ (минимум на отрезке) (более подробно см. [Наименьший общий предок. Нахождение за O \(sqrt \(N\)\) и O \(log N\) с препроцессингом O \(N\)](#)). Научимся теперь решать задачу RMQ в данном частном случае с препроцессингом O (N) и O (1) на запрос.

Заметим, что задача RMQ, к которой мы свели задачу LCA, является весьма специфичной: любые два соседних элемента в массиве **отличаются ровно на единицу** (поскольку элементы массива - это не что иное как высоты вершин, посещаемых в порядке обхода, и мы либо идём в потомка, тогда следующий элемент будет на 1 больше, либо идём в предка, тогда следующий элемент будет на 1 меньше). Собственно алгоритм Фарах-Колтона и Бендер как раз и представляет собой решение такой задачи RMQ.

Обозначим через A массив, над которым выполняются запросы RMQ, а N - размер этого массива.

Построим сначала алгоритм, решающий эту задачу **с препроцессингом O (N log N) и O (1) на запрос**. Это сделать легко: создадим так называемую Sparse Table T[l,i], где каждый элемент T[l,i] равен минимуму A на промежутке [l; l+2ⁱ]. Очевидно, 0 <= i <= ⌈log N⌉, и потому размер Sparse Table будет O (N log N). Построить её также легко за O (N log N), если заметить, что T[l,i] = min (T[l,i-1], T[l+2ⁱ⁻¹,i-1]). Как теперь отвечать на каждый запрос RMQ за O (1)? Пусть поступил запрос (l,r), тогда ответом будет min (T[l,sz], T[r-2^{sz}+1,sz]), где sz - наибольшая степень двойки, не превосходящая r-l+1. Действительно, мы как бы берём отрезок (l,r) и покрываем его двумя отрезками длины 2^{sz} - один начинающийся в l, а другой заканчивающийся в r (причём эти отрезки перекрываются, что в данном случае нам нисколько не мешает). Чтобы действительно достигнуть асимптотики O (1) на запрос, мы должны предпосчитать значения sz для всех возможных длин, которых есть O (log N) штук.

Теперь опишем, как улучшить этот алгоритм до асимптотики O (N).

Разобьём массив A на блоки размером K = 0.5 log₂ N. Для каждого блока посчитаем минимальный элемент в нём и его позицию (поскольку для решения задачи LCA нам важны не сами минимумы, а их позиции). Пусть B - это массив размером N / K, составленный из этих минимумов в каждом блоке. Построим по массиву B Sparse Table, как описано выше, при этом размер Sparse Table и время её построения будут равны:

$$\begin{aligned} \frac{N}{K} \log \frac{N}{K} &= (2N / \log N) \log (2N / \log N) = \\ &= (2N / \log N) (1 + \log (N / \log N)) \leq 2N / \log N + 2N = O (N) \end{aligned}$$

Теперь нам осталось только научиться быстро отвечать на запросы RMQ **внутри каждого блока**. В самом деле, если поступил запрос RMQ(l,r), то, если l и r находятся в разных блоках, то ответом будет минимум из следующих значений: минимум в блоке l, начиная с l и до конца блока, затем минимум в блоках после l и до r (не включительно), и наконец минимум в блоке r, от начала блока до r. На запрос "минимум в блоках" мы уже можем отвечать за O (1) с помощью Sparse Table, остались только запросы RMQ внутри блоков.

Здесь мы воспользуемся "+-1 свойством". Заметим, что, если внутри каждого блока от каждого его элемента отнять первый элемент, то все блоки будут однозначно определяться последовательностью длины K-1, состоящей из чисел +1. Следовательно, количество различных блоков будет равно:

$$2^{K-1} = 2^{0.5 \log N - 1} = 0.5 \sqrt{N}$$

Итак, количество различных блоков будет O (sqrt (N)), и потому мы можем предпосчитать результаты RMQ внутри всех различных блоков за O (sqrt(N) K²) = O (sqrt(N) log² N) = O (N). С точки зрения реализации, мы можем каждый блок характеризовать битовой маской длины K-1 (которая, очевидно, поместится в стандартный тип int), и хранить предпосчитанные RMQ в некотором массиве R[mask,l,r] размера O (sqrt(N) log² N).

Итак, мы научились предпосчитывать результаты RMQ внутри каждого блока, а также RMQ над самими блоками, всё в сумме за O (N), а отвечать на каждый запрос RMQ за O (1) - пользуясь только предвычисленными значениями, в худшем случае четырьмя: в блоке l, в блоке r, и на блоках между l и r не включительно.

Реализация

В начале программы указаны константы MAXN, LOG_MAXLIST и SQRT_MAXLIST, определяющие максимальное число вершин в графе, которые при необходимости надо увеличить.

```
const int MAXN = 100*1000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST+1)/2) + 1;

int n, root;
vector<int> g[MAXN];
int h[MAXN]; // vertex height
vector<int> a; // dfs list
int a_pos[MAXN]; // positions in dfs list
int block; // block size = 0.5 log A.size()
int bt[MAXBLOCKS][LOG_MAXLIST+1]; // sparse table on blocks (relative minimum positions in blocks)
int bhash[MAXBLOCKS]; // block hashes
int brmq[SQRT_MAXLIST][LOG_MAXLIST/2][LOG_MAXLIST/2]; // rmq inside each block, indexed by block hash
int log2[2*MAXN]; // precalced logarithms (floored values)

// walk graph
void dfs (int v, int curh) {
```

```

h[v] = curh;
a_pos[v] = (int)a.size();
a.push_back (v);
for (size_t i=0; i<g[v].size(); ++i)
    if (h[g[v][i]] == -1) {
        dfs (g[v][i], curh+1);
        a.push_back (v);
    }
}

int log (int n) {
    int res = 1;
    while (1<<res < n)   ++res;
    return res;
}

// compares two indices in a
inline int min_h (int i, int j) {
    return h[a[i]] < h[a[j]] ? i : j;
}

// O(N) preprocessing
void build_lca() {
    int sz = (int)a.size();
    block = (log(sz) + 1) / 2;
    int blocks = sz / block + (sz % block ? 1 : 0);

    // precalc in each block and build sparse table
    memset (bt, 255, sizeof bt);
    for (int i=0, bl=0, j=0; i<sz; ++i, ++j) {
        if (j == block)
            j = 0, ++bl;
        if (bt[bl][0] == -1 || min_h (i, bt[bl][0]) == i)
            bt[bl][0] = i;
    }
    for (int j=1; j<=log(sz); ++j)
        for (int i=0; i<blocks; ++i) {
            int ni = i + (1<<(j-1));
            if (ni >= blocks)
                bt[i][j] = bt[i][j-1];
            else
                bt[i][j] = min_h (bt[i][j-1], bt[ni][j-1]);
        }

    // calc hashes of blocks
    memset (bhash, 0, sizeof bhash);
    for (int i=0, bl=0, j=0; i<sz || j<block; ++i, ++j) {
        if (j == block)
            j = 0, ++bl;
        if (j > 0 && (i >= sz || min_h (i-1, i) == i-1))
            bhash[bl] += 1<<(j-1);
    }

    // precalc RMQ inside each unique block
    memset (brmq, 255, sizeof brmq);
    for (int i=0; i<blocks; ++i) {
        int id = bhash[i];
        if (brmq[id][0][0] != -1)  continue;
        for (int l=0; l<block; ++l) {
            brmq[id][1][l] = 1;
            for (int r=l+1; r<block; ++r) {
                brmq[id][l][r] = brmq[id][l][r-1];
                if (i*block+r < sz)
                    brmq[id][l][r] =
                        min_h (i*block+brmq[id][l][r], i*block+r) - i*block;
            }
        }
    }

    // precalc logarithms
    for (int i=0, j=0; i<sz; ++i) {
        if (1<<(j+1) <= i)  ++j;
        log2[i] = j;
    }

    // answers RMQ in block #bl [l;r] in O(1)
    inline int lca_in_block (int bl, int l, int r) {
        return brmq[bhash[bl]][l][r] + bl*block;
    }

    // answers LCA in O(1)
    int lca (int v1, int v2) {
        int l = a_pos[v1], r = a_pos[v2];
        if (l > r)  swap (l, r);

```

```

int bl = l/block, br = r/block;
if (bl == br)
    return a[lca_in_block(bl, l%block, r%block)];
int ans1 = lca_in_block(bl, l%block, block-1);
int ans2 = lca_in_block(br, 0, r%block);
int ans = min_h (ans1, ans2);
if (bl < br - 1) {
    int pw2 = log2[br-bl-1];
    int ans3 = bt[bl+1][pw2];
    int ans4 = bt[br-(1<<pw2)][pw2];
    ans = min_h (ans, min_h (ans3, ans4));
}
return a[ans];
}

```

Задача RMQ (Range Minimum Query - минимум на отрезке). Решение за O (1) с препроцессингом O (N)

Дан массив A[1..N]. Поступают запросы вида (L, R), на каждый запрос требуется найти минимум в массиве A, начиная с позиции L и заканчивая позицией R. Массив A изменяться в процессе работы не может, т.е. здесь описано решение статической задачи RMQ.

Здесь описано асимптотически оптимальное решение. Оно несколько отличается от других алгоритмов решения RMQ, поскольку оно сильно отличается от них: оно сводит задачу RMQ к задаче LCA, а затем использует [алгоритм Фарах-Колтона и Бендера](#), который сводит задачу LCA обратно к RMQ (но уже частного вида) и решает её.

Алгоритм

Построим по массиву A декартово дерево, где у каждой вершины ключом будет позиция i, а приоритетом - само число A[i] (предполагается, что в декартовом дереве приоритеты упорядочены от меньшего в корне к большим). Такое дерево можно построить за O (N). Тогда запрос RMQ(l,r) эквивалентен запросу LCA(l',r'), где l' - вершина, соответствующая элементу A[l], r' - соответствующая A[r]. Действительно, LCA найдёт вершину, которая по ключу находится между l' и r', т.е. по позиции в массиве A будет между l и r, и при этом вершину, наиболее близкую к корню, т.е. с наименьшим приоритетом, т.е. наименьшим значением.

Задачу LCA мы можем решать за O (1) с препроцессингом O (N) с помощью [алгоритма Фарах-Колтона и Бендера](#), который, что интересно, сводит задачу LCA обратно к задаче RMQ, но уже частного вида.

Наименьший общий предок. Нахождение за O(1) в оффлайн (алгоритм Тарьяна)

Дано дерево G с n вершинами и дано m запросов вида (a_i, b_i) . Для каждого запроса (a_i, b_i) требуется найти наименьшего общего предка вершин a_i и b_i , т.е. такую вершину c_i , которая наиболее удалена от корня дерева, и при этом является предком обеих вершин a_i и b_i .

Мы рассматриваем задачу в режиме оффлайн, т.е. считая, что все запросы известны заранее. Описываемый ниже алгоритм позволяет ответить на все m запросов за суммарное время $O(n + m)$, т.е. при достаточно большом m за $O(1)$ на запрос.

Алгоритм Тарьяна

Основой для алгоритма является структура данных "[Система непересекающихся множеств](#)", которая и была изобретена Тарьянном (Tarjan).

Алгоритм фактически представляет собой обход в глубину из корня дерева, в процессе которого постепенно находятся ответы на запросы. А именно, ответ на запрос (v, u) находится, когда обход в глубину находится в вершине u , а вершина v уже была посещена, или наоборот.

Итак, пусть обход в глубину находится в вершине v (и уже были выполнены переходы в её сыновей), и оказалось, что для какого-то запроса (v, u) вершина u уже была посещена обходом в глубину. Научимся тогда находить LCA этих двух вершин.

Заметим, что $LCA(v, u)$ является либо самой вершиной v , либо одним из её предков. Получается, нам надо найти самую нижнюю вершину

среди предков v (включая её саму), для которой вершина u является потомком. Заметим, что при фиксированном v по такому признаку (т.е. какой наименьший предок v является и предком какой-то вершины) вершины дерева дерева распадаются на совокупность непересекающихся классов. Для каждого предка $p \neq v$ вершины v её класс содержит саму эту вершину, а также все поддеревья с корнями в тех её сыновьях, которые лежат "слева" от пути до v (т.е. которые были обработаны ранее, чем была достигнута v).

Нам надо научиться эффективно поддерживать все эти классы, для чего мы и применим структуру данных "Система непересекающихся множеств". Каждому классу будет соответствовать в этой структуре множество, причём для представителя этого множества мы определим величину ANCESTOR — ту вершину p , которая и образует этот класс.

Рассмотрим подробно реализацию обхода в глубину. Пусть мы стоим в некоторой вершине v . Поместим её в отдельный класс в структуре непересекающихся множеств, $\text{ANCESTOR}[v] = v$. Как обычно в обходе в глубину, перебираем все исходящие рёбра (v, to) . Для каждого такого to мы сначала должны вызвать обход в глубину из этой вершины, а потом добавить эту вершину со всем её поддеревом в класс вершины v . Это реализуется операцией Union структуры данных "система непересекающихся множеств", с последующей установкой $\text{ANCESTOR} = v$ для представителя множества (т.к. после объединения представитель класса мог измениться). Наконец, после обработки всех рёбер мы перебираем все запросы вида (v, u) , и если u была помечена как посещённая обходом в глубину, то ответом на этот запрос будет вершина $\text{LCA}(v, u) = \text{ANCESTOR}[\text{FindSet}(u)]$. Нетрудно заметить, что для каждого запроса это условие (что одна вершина запроса является текущей, а другая была посещена ранее) выполнится ровно один раз.

Оценим асимптотику. Она складывается из нескольких частей. Во-первых, это асимптотика обхода в глубину, которая в данном случае составляет $O(n)$. Во-вторых, это операции по объединению множеств, которые в сумме для всех разумных n затрачивают $O(n)$ операций. В-третьих, это для каждого запроса проверка условия (два раза на запрос) и определение результата (один раз на запрос), каждое, опять же, для всех разумных n выполняется за $O(1)$. Итоговая асимптотика получается $O(n + m)$, что означает для достаточно больших m ($n = O(m)$) ответ за $O(1)$ на один запрос.

Реализация

Приведём полную реализацию данного алгоритма, включая слегка изменённую (с поддержкой ANCESTOR) реализацию системы пересекающихся множеств (рандомизированный варианта).

```
const int MAXN = максимальное число вершин в графе;
vector<int> g[MAXN], q[MAXN]; // граф и все запросы
int dsu[MAXN], ancestor[MAXN];
bool u[MAXN];

int dsu_get (int v) {
    return v == dsu[v] ? v : dsu_get (dsu[v]);
}

void dsu_unite (int a, int b, int new_ancestor) {
    a = dsu_get (a), b = dsu_get (b);
    if (rand() & 1) swap (a, b);
    dsu[a] = b, ancestor[b] = new_ancestor;
}

void dfs (int v) {
    dsu[v] = v, ancestor[v] = v;
    u[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!u[g[v][i]])
            dfs (g[v][i]);
    dsu_unite (v, g[v][i], v);
}
for (size_t i=0; i<q[v].size(); ++i)
    if (u[q[v][i]])
        printf ("%d %d -> %d\n", v+1, q[v][i]+1,
               ancestor[dsu_get (q[v][i]) ]+1);
}

int main() {
    ... чтение графа ...

    // чтение запросов
    for (;;) {
        int a, b = ...; // очередной запрос
        --a, --b;
        q[a].push_back (b);
        q[b].push_back (a);
    }

    // обход в глубину и ответ на запросы
    dfs (0);
}
```

Максимальный поток методом Эдмондса-Карпа за $O(NM^2)$

Пусть дан граф G , в котором выделены две вершины: исток S и сток T , а у каждого ребра определена пропускная способность $C_{u,v}$. Поток F можно представить как поток вещества, которое могло бы пройти по сети от истока к стоку, если рассматривать граф как сеть труб с некоторыми пропускными способностями. Т.е. поток - функция $F_{u,v}$ определённая на множестве рёбер графа.

Задача заключается в нахождении максимального потока. Здесь будет рассмотрен метод Эдмондса-Карпа, работающий за $O(NM^2)$, или (другая оценка) $O(FM)$, где F - величина искомого потока. Алгоритм был предложен в 1972 году.

Алгоритм

Остаточной пропускной способностью называется пропускная способность ребра за вычетом текущего потока вдоль этого ребра. При этом надо помнить, что если некоторый поток протекает по ориентированному ребру, то возникает так называемое обратное ребро (направленное в обратную сторону), которое будет иметь нулевую пропускную способность, и по которому будет протекать тот же по величине поток, но со знаком минус. Если же ребро было неориентированным, то оно как бы распадается на два ориентированных ребра с одинаковой пропускной способностью, и каждое из этих рёбер является обратным для другого (если по одному протекает поток F , то по другому протекает $-F$).

Общая схема алгоритма Эдмондса-Карпа такова. Сначала полагаем поток равным нулю. Затем ищём дополняющий путь, т.е. простой путь из S в T по тем рёбрам, у которых остаточная пропускная способность строго положительна. Если дополняющий путь был найден, то производится увеличение текущего потока вдоль этого пути. Если же пути не было найдено, то текущий поток является максимальным. Для поиска дополняющего пути может использоваться как [Обход в ширину](#), так и [Обход в глубину](#).

Рассмотрим более точно процедуру увеличения потока. Пусть мы нашли некоторый дополняющий путь, тогда пусть C - наименьшая из остаточных пропускных способностей рёбер этого пути. Процедура увеличения потока заключается в следующем: для каждого ребра (u, v) дополняющего пути выполним: $F_{u,v} += C$, а $F_{v,u} = -F_{u,v}$ (или, что то же самое, $F_{v,u} -= C$).

Величиной потока будет сумма всех неотрицательных величин $F_{S,v}$, где v - любая вершина, соединённая с истоком.

Реализация

```
const int inf = 1000*1000*1000;

typedef vector<int> graf_line;
typedef vector<graf_line> graf;

typedef vector<int> vint;
typedef vector<vint> vvint;

int main()
{
    int n;
    cin >> n;
    vvint c (n, vint(n));
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            cin >> c[i][j];
    // исток - вершина 0, сток - вершина n-1

    vvint f (n, vint(n));
    for (;;)
    {
        vint from (n, -1);
        vint q (n);
        int h=0, t=0;
        q[t++] = 0;
        from[0] = 0;
        for (int cur; h<t;)
        {
            cur = q[h++];
            for (int v=0; v<n; v++)
                if (from[v] == -1 &&
                    c[cur][v]-f[cur][v] > 0)
                {
                    q[t++] = v;
                    from[v] = cur;
                }
        }

        if (from[n-1] == -1)
            break;
        int cf = inf;
        for (int cur=n-1; cur!=0; )
        {
            int prev = from[cur];
            cf = min (cf, c[prev][cur]-f[prev][cur]);
            cur = prev;
        }
    }

    if (from[n-1] == -1)
        break;
    int cf = inf;
    for (int cur=n-1; cur!=0; )
    {
        int prev = from[cur];
        cf = min (cf, c[prev][cur]-f[prev][cur]);
        cur = prev;
    }
}
```

```

    }

    for (int cur=n-1; cur!=0; )
    {
        int prev = from[cur];
        f[prev][cur] += cf;
        f[cur][prev] -= cf;
        cur = prev;
    }

    int flow = 0;
    for (int i=0; i<n; i++)
        if (c[0][i])
            flow += f[0][i];

    cout << flow;
}

```

Максимальный поток методом Проталкивания предпотока за $O(N^4)$

Пусть дан граф G , в котором выделены две вершины: исток S и сток T , а у каждого ребра определена пропускная способность $C_{u,v}$. Поток F можно представить как поток вещества, которое могло бы пройти по сети от истока к стоку, если рассматривать граф как сеть труб с некоторыми пропускными способностями. Т.е. поток - функция $F_{u,v}$, определённая на множестве рёбер графа.

Задача заключается в нахождении максимального потока. Здесь будет рассмотрен метод Проталкивания предпотока, работающий за $O(N^4)$, или, точнее, за $O(N^2 M)$. Алгоритм был предложен Гольдбергом в 1985 году.

Алгоритм

Общая схема алгоритма такова. На каждом шаге будем рассматривать некоторый предпоток - т.е. функцию, которая по свойствам напоминает поток, но не обязательно удовлетворяет закону сохранения потока. На каждом шаге будем пытаться применить какую-либо из двух операций: проталкивание потока или поднятие вершины. Если на каком-то шаге станет невозможно применить какую-либо из двух операций, то мы нашли требуемый поток.

Для каждой вершины определена её высота H_u , причём $H_S = N$, $H_T = 0$, и для любого остаточного ребра (u, v) имеем $H_u \leq H_v + 1$.

Для каждой вершины (кроме S) можно определить её избыток: $E_u = F_{V,u}$. Вершина с положительным избытком называется переполненной.

Операция проталкивания $Push(u, v)$ применима, если вершина u переполнена, остаточная пропускная способность $Cf_{u,v} > 0$ и $H_u = H_v + 1$.

Операция проталкивания заключается в максимальном увеличении потока из u в v , ограниченном избытком E_u и остаточной пропускной способностью $Cf_{u,v}$.

Операция поднятия $Lift(u)$ поднимает переполненную вершину u на максимально допустимую высоту. Т.е. $H_u = 1 + \min\{H_v\}$, где (u, v) - остаточное ребро.

Осталось только рассмотреть инициализацию потока. Нужно инициализировать только следующие значения: $F_{S,v} = C_{S,v}$, $F_{u,S} = -C_{u,S}$, остальные значения положить равными нулю.

Реализация

```

const int inf = 1000*1000*1000;

typedef vector<int> graf_line;
typedef vector<graf_line> graf;

typedef vector<int> vint;
typedef vector<vint> vvint;

void push (int u, int v, vvint & f, vint & e, const vvint & c)
{
    int d = min (e[u], c[u][v] - f[u][v]);
    f[u][v] += d;
    f[v][u] = - f[u][v];
}

```

```

e[u] -= d;
e[v] += d;
}

void lift (int u, vint & h, const vvint & f, const vvint & c)
{
    int d = inf;
    for (int i = 0; i < (int)f.size(); i++)
        if (c[u][i]-f[u][i] > 0)
            d = min (d, h[i]);
    if (d == inf)
        return;
    h[u] = d + 1;
}

int main()
{
    int n;
    cin >> n;
    vvint c (n, vint(n));
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            cin >> c[i][j];
    // исток - вершина 0, сток - вершина n-1

    vvint f (n, vint(n));
    for (int i=1; i<n; i++)
    {
        f[0][i] = c[0][i];
        f[i][0] = -c[0][i];
    }

    vint h (n);
    h[0] = n;

    vint e (n);
    for (int i=1; i<n; i++)
        e[i] = f[0][i];

    for ( ; ; )
    {
        int i;
        for (i=1; i<n-1; i++)
            if (e[i] > 0)
                break;
        if (i == n-1)
            break;

        int j;
        for (j=0; j<n; j++)
            if (c[i][j]-f[i][j] > 0 && h[i]==h[j]+1)
                break;
        if (j < n)
            push (i, j, f, e, c);
        else
            lift (i, h, f, c);
    }

    int flow = 0;
    for (int i=0; i<n; i++)
        if (c[0][i])
            flow += f[0][i];

    cout << max(flow,0);
}

```

Модификация метода Проталкивания предпотока для нахождения максимального потока за $O(N^3)$

Предполагается, что вы уже прочитали [Метод Проталкивания](#) предпоптока нахождения максимального потока за $O(N^4)$.

Описание

Модификация чрезвычайно проста: на каждой итерации среди всех переполненных вершин мы выбираем только те вершины, которые имеют **наибольшую высоту**, и применяем проталкивание/поднятие только к этим вершинам. Более того, для выбора вершин с наибольшей высотой нам не понадобятся никакие структуры данных, достаточно просто хранить список вершин с наибольшей высотой и просто пересчитывать его, если все вершины из этого списка были обработаны (тогда в список добавляются вершины с уже меньшей высотой), а при появлении новой переполненной вершины с большей высотой, чем в списке, очищать список и добавлять вершину в список.

Несмотря на простоту, эта модификация позволяет снизить асимптотику на целый порядок. Если быть точным, асимптотика получившего алгоритма равна $O(NM + N^2 \sqrt{M})$, что в худшем случае составляет $O(N^3)$.

Эта модификация была предложена Черияном (Cherian) и Махешвари (Maheshvari) в 1989 г.

Реализация

Здесь приведена готовая реализация этого алгоритма.

Отличие от обычного алгоритма проталкивания - только в наличии массива `maxh`, в котором будут храниться номера переполненных вершин с максимальной высотой. Размер массива указан в переменной `sz`. Если на какой-то итерации оказывается, что этот массив пустой (`sz==0`), то мы заполняем его (просто проходя по всем вершинам); если после этого массив по-прежнему пустой, то переполненных вершин нет, и алгоритм останавливается. Иначе мы проходим по вершинам в этом списке, применяя к ним проталкивание или поднятие. После выполнения операции проталкивания текущая вершина может перестать быть переполненной, в этом случае удаляем её из списка `maxh`. Если после какой-то операции поднятия высота текущей вершины становится больше высоты вершин в списке `maxh`, то мы очищаем список (`sz=0`), и сразу переходим к следующей итерации алгоритма проталкивания (на которой будет построен новый список `maxh`).

```
const int INF = 1000*1000*1000;

int main() {

    int n;
    vector < vector<int> > c (n, vector<int> (n));
    int s, t;
    ... чтение n, c, s, t ...

    vector<int> e (n);
    vector<int> h (n);
    h[s] = n-1;
    vector < vector<int> > f (n, vector<int> (n));

    for (int i=0; i<n; ++i) {
        f[s][i] = c[s][i];
        f[i][s] = -f[s][i];
        e[i] = c[s][i];
    }

    vector<int> maxh (n);
    int sz = 0;
    for (;;) {
        if (!sz)
            for (int i=0; i<n; ++i)
                if (i != s && i != t && e[i] > 0) {
                    if (!sz && h[i] > h[maxh[0]]) sz = 0;
                    maxh[sz++] = i;
                }
        if (!sz) break;
        while (sz) {
            int i = maxh[sz-1];
            bool pushed = false;
            for (int j=0; j<n && e[i]; ++j)
                if (c[i][j]-f[i][j] > 0 && h[i] == h[j]+1) {
                    pushed = true;
                    int addf = min (c[i][j]-f[i][j], e[i]);
                    f[i][j] += addf, f[j][i] -= addf;
                    e[i] -= addf, e[j] += addf;
                    if (e[i] == 0) --sz;
                }
            if (!pushed) {
                h[i] = INF;
                for (int j=0; j<n; ++j)
                    if (c[i][j]-f[i][j] > 0 && h[j]+1 < h[i])
                        h[i] = h[j]+1;
                if (h[i] > h[maxh[0]]) {
                    sz = 0;
                    break;
                }
            }
        }
    ...
    ... вывод потока f ...
}
```

Нахождение потока в графе, в котором у каждого ребра указано минимальное и максимальное значение потока

Пусть дан граф G , в котором для каждого ребра помимо пропускной способности (максимального значения потока вдоль этого ребра) указано и минимальное значение потока, который должен проходить по этому ребру.

Здесь мы рассмотрим две задачи: 1) требуется найти произвольный поток, удовлетворяющий всем ограничениям, и 2) требуется найти минимальный поток, удовлетворяющий всем ограничениям.

Решение задачи 1

Обозначим через L_i минимальную величину потока, которая может проходить по i -му ребру, а через R_i - его максимальная величина.

Произведём в графе следующие **изменения**. Добавим новый исток S' и сток T' . Рассмотрим все рёбра, у которых L_i отлично от нуля. Пусть i - номер такого ребра. Пусть концы этого ребра (ориентированного) - это вершины A_i и B_i . Добавим ребро (S', B_i) , у которого $L = 0$, $R = L_i$, добавим ребро (A_i, T') , у которого $L = 0$, $R = L_i$, а у самого i -го ребра положим $R_i = R_i - L_i$, а $L_i = 0$. Наконец, добавим в граф ребро из T в S (старых стока и истока), у которого $L = 0$, $R = \text{INF}$.

После выполнения этих преобразований все рёбра графа будут иметь $L_i = 0$, т.е. мы свели эту задачу к обычной задаче нахождения максимального потока (но уже в модифицированном графе с новыми истоком и стоком) (чтобы понять, почему именно максимального - читайте нижеследующее объяснение).

Корректность этих преобразований понять сложнее. Неформальное **объяснение** такое. Каждое ребро, у которого L_i отлично от нуля, мы заменяем на два ребра: одно с пропускной способностью L_i , а другое - с $R_i - L_i$. Нам требуется найти поток, который бы обязательно насытил первое ребро из этой пары (т.е. поток вдоль этого ребра должен быть равен L_i); второе ребро нас волнует меньше - поток вдоль него может быть любым, лишь бы он не превосходил его пропускной способности. Итак, нам требуется найти такой поток, который бы обязательно насытил некоторое множество рёбер. Рассмотрим каждое такое ребро, и выполним такую операцию: подведём к его концу ребро из нового истока S' , подведём ребро из его начала к стоку T' , само ребро удалим, а из старого стока T к старому истоку S проведём ребро бесконечной пропускной способности. Этими действиями мы имитируем тот факт, что это ребро насыщено - из ребра будет вытекать L_i единиц потока (мы имитируем это с помощью нового истока, который подаёт на конец ребра нужное количество потока), а втекать в него будет опять же L_i единиц потока (но вместо ребра этот поток попадёт в новый сток). Поток из нового истока протекает по одной части графа, дотекает до старого стока T , из него протекает в старый исток S , затем течёт по другой части графа, и наконец приходит к началу нашего ребра, и попадает в новый сток T' . Т.е., если мы найдём в этом модифицированном графе максимальный поток (и в сток попадёт нужное количество потока, т.е. сумма всех значений L_i - иначе величина потока будет меньше, и ответа попросту не существует), то мы одновременно найдём поток в исходном графе, который будет удовлетворять все ограничениям минимума, и, разумеется, всем ограничениям максимума.

Решение задачи 2

Заметим, что по ребру из старого стока в старый исток с пропускной способностью INF протекает весь старый поток, т.е. пропускная способность этого ребра влияет на величину старого потока. При достаточно большой величине пропускной способности этого ребра (т.е. INF) старый поток ничем не ограничен. Если мы будем уменьшать пропускную способность, то и, начиная с некоторого момента, будет уменьшаться и величина старого потока. Но при слишком малом значении величина потока станет недостаточной, чтобы обеспечить выполнение ограничений (на минимальное значение потока вдоль рёбер). Очевидно, здесь можно применить **бинарный поиск по значению INF** , и найти такое её наименьшее значение, при котором все ограничения ещё будут удовлетворяться, но старый поток будет иметь минимальное значение.

Поток минимальной стоимости (min-cost-flow). Алгоритм увеличивающих путей

Дана сеть G , состоящая из N вершин и M рёбер. У каждого ребра (вообще говоря, ориентированному, но по этому поводу см. ниже) указана пропускная способность (целое неотрицательное число) и стоимость единицы потока вдоль этого ребра (некоторое целое число). В графе указан исток S и сток T . Даётся некоторая величина K потока, требуется найти поток этой величины, причём среди всех потоков этой величины выбрать поток с наименьшей стоимостью ("задача min-cost-flow").

Иногда задачу ставят немного по-другому: требуется найти максимальный поток наименьшей стоимости ("задача min-cost-max-flow").

Обе эти задачи достаточно эффективно решаются описанным ниже алгоритмом увеличивающих путей.

Описание

Алгоритм очень похож на [алгоритм Эдмондса-Карпа вычисления максимального потока](#).

Простейший случай

Рассмотрим для начала простейший случай, когда граф - ориентированный, и между любой парой вершин не более одного ребра (если есть ребро (i,j) , то ребра (j,i) быть не должно).

Пусть U_{ij} - пропускная способность ребра (i,j) , если это ребро существует. Пусть C_{ij} - стоимость единицы потока вдоль ребра (i,j) . Пусть F_{ij} - величина потока вдоль ребра (i,j) , изначально все величины потоков равны нулю.

Модифицируем сеть следующим образом: для каждого ребра (i,j) добавим в сеть так называемое **обратное** ребро (j,i) с пропускной способностью $U_{ji} = 0$ и стоимостью $C_{ji} = -C_{ij}$. Поскольку, по нашему предположению, ребра (j,i) до этого в сети не было, то модифицированная таким образом сеть по-прежнему не будет мультиграфом. Кроме того, на всём протяжении работы алгоритма будем поддерживать верным условие: $F_{ji} = -F_{ij}$.

Определим **остаточную сеть** для некоторого зафиксированного потока F следующим образом (собственно, так же, как и в алгоритме Форда-Фалкерсона): остаточной сети принадлежат только ненасыщенные рёбра (т.е. у которых $F_{ij} < U_{ij}$), а остаточную пропускную способность каждого такого ребра как $U_{ij}' = U_{ij} - F_{ij}$.

Собственно **алгоритм min-cost-flow** заключается в следующем. На каждой итерации алгоритма находим кратчайший путь в остаточной сети из S в T (кратчайший относительно стоимостей C_{ij}). Если путь не был найден, то алгоритм завершается, поток F - искомый. Если же путь был найден, то мы увеличиваем поток вдоль него настолько, насколько это возможно (т.е. проходим вдоль этого пути, находим минимальную остаточную пропускную способность MIN_UPI среди рёбер этого пути, и затем увеличиваем поток вдоль каждого ребра пути на величину MIN_UPI , не забывая уменьшать на такую же величину поток вдоль обратных рёбер). Если в какой-то момент величина потока достигла величины K (данной нам по условию величины потока), то мы также останавливаем алгоритм (следует учесть, что тогда на последней итерации алгоритма при увеличении потока вдоль пути нужно увеличивать поток на такую величину, чтобы итоговый поток не превзошёл K , но это выполнить легко).

Нетрудно заметить, что если положить K равным бесконечности, то алгоритм найдёт максимальный поток минимальной стоимости, т.е. один и тот же алгоритм без изменений решает обе задачи min-cost-flow и min-cost-max-flow.

Случай неориентированных графов, мультиграфов

Случай неориентированных графов и мультиграфов в концептуальном плане ничем не отличается от вышеописанного, поэтому собственно алгоритм будет работать и на таких графах. Однако возникают некоторые сложности в реализации, на которые следует обратить внимание.

Неориентированное ребро (i,j) - это фактически два ориентированных ребра (i,j) и (j,i) с одинаковыми пропускными способностями и стоимостями. Поскольку вышеописанный алгоритм min-cost-flow требует для каждого неориентированного ребра создать обратное ему ребро, то в итоге получается, что неориентированное ребро расщепляется на 4 ориентированных ребра, и мы фактически получаем случай **мультиграфа**.

Какие проблемы вызывают **кратные рёбра**? Во-первых, поток по каждому из кратных рёбер должен сохраняться отдельно. Во-вторых, при поиске кратчайшего пути нужно учитывать, что важно то, какое именно из кратных рёбер выбрать при восстановлении пути по предкам. Т.е. вместо обычного массива предков для каждой вершины мы должны хранить вершину-предка и номер ребра, по которому мы из неё пришли. В-третьих, при увеличении потока вдоль некоторого ребра нужно, согласно алгоритму, уменьшить поток вдоль обратного ребра. Поскольку у нас могут быть кратные рёбра, то придётся для каждого ребра хранить номер ребра, обратного ему.

Других сложностей с неориентированными графиками и мультиграфами нет.

Анализ времени работы

По аналогии с анализом алгоритма Эдмондса-Карпа, мы получаем такую оценку: $O(NM) * T(N, M)$, где $T(N, M)$ - время, необходимое для нахождения кратчайшего пути в графе с N вершинами и M рёбрами. Если это реализовать с помощью [простейшего варианта алгоритма Дейкстры](#), то для всего алгоритма min-cost-flow получится оценка $O(N^3M)$, правда, алгоритм Дейкстры придётся модифицировать, чтобы он работал на графах с отрицательными весами (это называется алгоритм Дейкстры с потенциалами).

Вместо этого можно использовать [алгоритм Левита](#), который, хотя и асимптотически намного хуже, но на практике работает очень быстро (примерно за то же время, что и алгоритм Дейкстры).

Реализация

Здесь приведена реализация алгоритма min-cost-flow, базирующаяся на [алгоритме Левита](#).

На вход алгоритма подаётся сеть (неориентированный мультиграф) с N вершинами и M рёбрами, и K - величина потока, который нужно найти. Алгоритм находит поток величины K минимальной стоимости, если такой существует. Иначе он находит поток максимальной величины минимальной стоимости.

В программе есть специальная функция для добавления ориентированного ребра. Если нужно добавить неориентированное ребро, то эту функцию нужно вызывать для каждого ребра (i,j) дважды: от (i,j) и от (j,i) .

```
const int INF = 1000*1000*1000;

struct rib {
    int b, u, c, f;
    size_t back;
};

void add_rib (vector < vector<rib> > & g, int a, int b, int u, int c) {
    rib r1 = { b, u, c, 0, g[b].size() };
    rib r2 = { a, 0, -c, 0, g[a].size() };
    g[a].push_back (r1);
    g[b].push_back (r2);
}
```

```

int main()
{
    int n, m, k;
    vector < vector<rib> > g (n);
    int s, t;
    ... чтение графа ...

    int flow = 0, cost = 0;
    while (flow < k) {
        vector<int> id (n, 0);
        vector<int> d (n, INF);
        vector<int> q (n);
        vector<int> p (n);
        vector<size_t> p_rib (n);
        int qh=0, qt=0;
        q[qt++] = s;
        d[s] = 0;
        while (qh != qt) {
            int v = q[qh++];
            id[v] = 2;
            if (qh == n) qh = 0;
            for (size_t i=0; i<g[v].size(); ++i) {
                rib & r = g[v][i];
                if (r.f < r.u && d[v] + r.c < d[r.b]) {
                    d[r.b] = d[v] + r.c;
                    if (id[r.b] == 0) {
                        q[qt++] = r.b;
                        if (qt == n) qt = 0;
                    }
                    else if (id[r.b] == 2) {
                        if (--qh == -1) qh = n-1;
                        q[qh] = r.b;
                    }
                    id[r.b] = 1;
                    p[r.b] = v;
                    p_rib[r.b] = i;
                }
            }
        }
        if (d[t] == INF) break;
        int addflow = k - flow;
        for (int v=t; v!=s; v=p[v]) {
            int pv = p[v]; size_t pr = p_rib[v];
            addflow = min (addflow, g[pv][pr].u - g[pv][pr].f);
        }
        for (int v=t; v!=s; v=p[v]) {
            int pv = p[v]; size_t pr = p_rib[v], r = g[pv][pr].back;
            g[pv][pr].f += addflow;
            g[v][r].f -= addflow;
            cost += g[pv][pr].c * addflow;
        }
        flow += addflow;
    }
    ... вывод результата ...
}

```

Задача о назначениях. Решение с помощью min-cost-flow

Задача имеет две эквивалентные постановки:

- Дана квадратная матрица $A[1..N, 1..N]$. Нужно выбрать в ней N элементов так, чтобы в каждой строке и столбце был выбран ровно один элемент, а сумма значений этих элементов была наименьшей.
- Имеется N заказов и N станков. Про каждый заказ известна стоимость его изготовления на каждом станке. На каждом станке можно выполнять только один заказ. Требуется распределить все заказы по станкам так, чтобы минимизировать суммарную стоимость.

Здесь мы рассмотрим решение задачи на основе алгоритма [нахождения потока минимальной стоимости \(min-cost-flow\)](#), решив задачу о назначениях за $O(N^5)$.

Описание

Построим двудольную сеть: имеется исток S, сток T, в первой доле находятся N вершин (соответствующие строкам матрицы или заказам), во второй - тоже N вершин (соответствующие столбцам матрицы или станкам). Между каждой вершиной i первой доли и каждой вершиной j второй доли проведём ребро с пропускной способностью 1 и стоимостью A_{ij} . От истока S проведём ребра ко всем вершинам i первой доли с пропускной способностью 1 и стоимостью 0. От каждой вершины второй доли j к стоку T проведём ребра с пропускной способностью 1 и стоимостью 0.

Найдём в полученной сети максимальный поток минимальной стоимости. Очевидно, величина потока будет равна N. Далее, очевидно, что для каждой вершины i из первой доли найдётся ровно одна вершина j из второй доли, такая, что поток $F_{ij} = 1$. Наконец, очевидно, это взаимно однозначное соответствие между вершинами первой доли и вершинами второй доли является решением задачи (поскольку найденный поток имеет минимальную стоимость, то сумма стоимостей выбранных рёбер будет наименьшей из возможных, что и является критерием оптимальности).

Поскольку максимальный поток минимальной стоимости ищется за $O(N^3 M)$, то асимптотика этого решения задачи о назначениях равна $O(N^5)$.

Реализация

Приведённая здесь реализация длинноватая, возможно, её можно значительно сократить.

```
typedef vector<int> vint;
typedef vector<vint> vvint;

const int INF = 1000*1000*1000;

int main()
{
    int n;
    vvint a (n, vint (n));
    ... чтение a ...

    int m = n * 2 + 2;
    vvint f (m, vint (m));
    int s = m-2, t = m-1;
    int cost = 0;
    for (;;)
    {
        vector<int> dist (m, INF);
        vector<int> p (m);
        vector<int> type (m, 2);
        deque<int> q;
        dist[s] = 0;
        p[s] = -1;
        type[s] = 1;
        q.push_back (s);
        for (; !q.empty(); )
        {
            int v = q.front(); q.pop_front();
            type[v] = 0;
            if (v == s)
            {
                for (int i=0; i<n; ++i)
                    if (f[s][i] == 0)
                {
                    dist[i] = 0;
                    p[i] = s;
                    type[i] = 1;
                    q.push_back (i);
                }
            }
            else
            {
                if (v < n)
                {
                    for (int j=n; j<n+n; ++j)
                        if (f[v][j] < 1 && dist[j] > dist[v] + a[v][j-n])
                    {
                        dist[j] = dist[v] + a[v][j-n];
                        p[j] = v;
                        if (type[j] == 0)
                            q.push_front (j);
                        else if (type[j] == 2)
                            q.push_back (j);
                        type[j] = 1;
                    }
                }
                else
                {
                    for (int j=0; j<n; ++j)
                        if (f[v][j] < 0 && dist[j] > dist[v] - a[j][v-n])
                    {
                        dist[j] = dist[v] - a[j][v-n];
                        p[j] = v;
                    }
                }
            }
        }
    }
}
```

```

        if (type[j] == 0)
            q.push_front (j);
        else if (type[j] == 2)
            q.push_back (j);
        type[j] = 1;
    }
}
}

int curcost = INF;
for (int i=n; i<n+n; ++i)
    if (f[i][t] == 0 && dist[i] < curcost)
    {
        curcost = dist[i];
        p[t] = i;
    }
if (curcost == INF) break;
cost += curcost;
for (int cur=t; cur!=-1; cur=p[cur])
{
    int prev = p[cur];
    if (prev!=-1)
        f[cur][prev] = - (f[prev][cur] = 1);
}

printf ("%d\n", cost);
for (int i=0; i<n; ++i)
    for (int j=0; j<n; ++j)
        if (f[i][j+n] == 1)
            printf ("%d ", j+1);
}

```

Задача о назначениях. Венгерский алгоритм (алгоритм Куна) за $O(N^4)$

Задача имеет две эквивалентные постановки:

- Дана квадратная матрица $A[1..N, 1..N]$. Нужно выбрать в ней N элементов так, чтобы в каждой строке и столбце был выбран ровно один элемент, а сумма значений этих элементов была наименьшей.
- Имеется N заказов и N станков. Про каждый заказ известна стоимость его изготовления на каждом станке. На каждом станке можно выполнять только один заказ. Требуется распределить все заказы по станкам так, чтобы минимизировать суммарную стоимость.

Здесь рассмотрен венгерский алгоритм (изобретённый Куном), который позволяет решать задачу о назначениях очень эффективно (по сравнению с алгоритмом min-cost-flow).

Описание

[описание пока отсутствует]

Реализация

```

const int INF = 1000*1000*1000;

int n;
vector < vector<int> > a;
vector<int> xy, yx;
vector<char> vx, vy;
vector<int> minrow, mincol;

bool dotry (int i) {
    if (vx[i]) return false;
    vx[i] = true;
    for (int j=0; j<n; ++j)
        if (a[i][j]-minrow[i]-mincol[j] == 0)
            vy[j] = true;
    for (int j=0; j<n; ++j)
        if (a[i][j]-minrow[i]-mincol[j] == 0 && yx[j] == -1) {

```

```

    xy[i] = j;
    yx[j] = i;
    return true;
}
for (int j=0; j<n; ++j)
if (a[i][j]-minrow[i]-mincol[j] == 0 && dotry (yx[j])) {
    xy[i] = j;
    yx[j] = i;
    return true;
}
return false;
}

int main() {
    ... чтение a ...

    mincol.assign (n, INF);
    minrow.assign (n, INF);
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            minrow[i] = min (minrow[i], a[i][j]);
    for (int j=0; j<n; ++j)
        for (int i=0; i<n; ++i)
            mincol[j] = min (mincol[j], a[i][j] - minrow[i]);

    xy.assign (n, -1);
    yx.assign (n, -1);
    for (int c=0; c<n; ) {
        vx.assign (n, 0);
        vy.assign (n, 0);
        int k = 0;
        for (int i=0; i<n; ++i)
            if (xy[i] == -1 && dotry (i))
                ++k;
        c += k;
        if (k == 0) {
            int z = INF;
            for (int i=0; i<n; ++i)
                if (vx[i])
                    for (int j=0; j<n; ++j)
                        if (!vy[j])
                            z = min (z, a[i][j]-minrow[i]-mincol[j]);
            for (int i=0; i<n; ++i) {
                if (vx[i]) minrow[i] += z;
                if (vy[i]) mincol[i] -= z;
            }
        }
    }

    int ans = 0;
    for (int i=0; i<n; ++i)
        ans += a[i][xy[i]];
    printf ("%d\n", ans);
    for (int i=0; i<n; ++i)
        printf ("%d ", xy[i]+1);
}
}

```

Нахождение минимального разреза. Алгоритм Штор-Вагнера

Постановка задачи

Дан неориентированный взвешенный граф G с n вершинами и m рёбрами. Разрезом C называется некоторое подмножество вершин (фактически, разрез — разбиение вершин на два множества: принадлежащие C и все остальные). Весом разреза называется сумма весов рёбер, проходящих через разрез, т.е. таких рёбер, ровно один конец которых $\in C$. Требуется найти разрез минимального веса.

Иногда эту задачу называют "глобальным минимальным разрезом", в отличие от задачи, когда заданы вершины-сток и исток. Глобальный минимальный разрез равен минимуму среди разрезов минимальной стоимости по всевозможным парам исток-сток. Впрочем, как нетрудно заметить, достаточно лишь перебирать всевозможные истоки, а в качестве стока брать любую отличную вершину.

Хотя эту задачу можно решить с помощью алгоритма нахождения максимального потока (он решает неглобальную задачу минимального разреза), однако ниже описан гораздо более простой и быстрый алгоритм, предложенный Матильдой Штор (Mechthild Stoer) и Франком Вагнером (Frank Wagner) в 1994 г.

В общем случае допускаются петли и кратные рёбра, хотя, понятно, петли абсолютно никак не влияют на результат, а все кратные рёбра можно заменить одним ребром с их суммарным весом. Поэтому мы для простоты будем считать, что во входном графе петли и кратные рёбра отсутствуют.

Описание алгоритма

Базовая идея алгоритма очень проста. Будем итеративно повторять следующий процесс: находить минимальный разрез между какой-нибудь парой вершин s и t , а затем объединять эти две вершины в одну (соединяя списки смежности). В конце концов, после $n - 1$ итерации, граф сожмётся в единственную вершину и процесс остановится. После этого ответом будет являться минимальный среди всех $n - 1$ найденных разрезов. Действительно, на каждой i -й стадии найденный минимальный разрез C_i между вершинами s_i и t_i либо окажется искомым глобальным минимальным разрезом, либо же, напротив, вершины s_i и t_i невыгодно относить к разным множествам, поэтому мы ничего не ухудшаем, объединяя эти две вершины в одну.

Таким образом, мы свели задачу к следующей: для данного графа найти **минимальный разрез между какой-нибудь произвольной парой вершин s и t** . Для решения этой задачи был предложен следующий, тоже итеративный процесс. Вводим некоторое множество вершин A , которое изначально содержит единственную произвольную вершину. На каждом шаге находится вершина, **наиболее сильно связанныя** с множеством A , т.е. вершина $v \notin A$, для которой следующая величина максимальна:

$$w(v, A) = \sum_{\substack{(v,u) \in G, \\ u \in A}} c(v, u)$$

(т.е. сумма весов рёбер, один конец которых v , а другой принадлежит A)

Опять же, этот процесс завершится через $n - 1$ итерацию, когда все вершины перейдут в множество A (кстати говоря, этот процесс очень напоминает [алгоритм Прима](#)). Тогда, как утверждает **теорема Штор-Вагнера**, если мы обозначим через s и t последние две добавленные в A вершины, то минимальный разрез между вершинами s и t будет состоять из единственной вершины — t . Доказательство этой теоремы будет приведено в следующем разделе (как это часто бывает, само по себе оно никак не способствует пониманию алгоритма).

Таким образом, общая **схема алгоритма** Штор-Вагнера такова. Алгоритм состоит из $n - 1$ фазы. На каждой фазе множество A сначала полагается состоящим из какой-либо вершины; подсчитываются стартовые веса вершин $w(v, A)$. Затем происходит $n - 1$ итерация, на каждой из которых выбирается вершина u с наибольшим значением $w(v, A)$ и добавляется в множество A , после чего пересчитываются значения w для оставшихся вершин (для чего, очевидно, надо пройтись по всем рёбрам списка смежности выбранной вершины u). После выполнения всех итераций мы запоминаем в s и t номера последних двух добавленных вершин, а в качестве стоимости найденного минимального разреза между s и t можно взять значение $w(t, A \setminus t)$. Затем надо сравнить найденный минимальный разрез с текущим ответом, если меньше, то обновить ответ. Перейти к следующей фазе.

Если не использовать никаких сложных структур данных, то самой критичной частью будет нахождение вершины с наибольшей величиной w . Если производить это за $O(n)$, то, учитывая, что всего фаз $n - 1$, и по $n - 1$ итерации в каждой, итоговая **асимптотика алгоритма** получается $O(n^3)$.

Если для нахождения вершины с наибольшей величиной w использовать **Фibonacciевы кучи** (которые позволяют увеличивать значение ключа за $O(1)$ в среднем и извлекать максимум за $O(\log n)$ в среднем), то все связанные с множеством A операции на одной фазе выполняются за $O(m + n \log n)$. Итоговая асимптотика алгоритма в таком случае составит $O(nm + n^2 \log n)$.

Доказательство теоремы Штор-Вагнера

Напомним условие этой теоремы. Если добавить в множество A по очереди все вершины, каждый раз добавляя вершину, наиболее сильно связанную с этим множеством, то обозначим предпоследнюю добавленную вершину через s , а предпоследнюю — через t . Тогда минимальный s - t разрез состоит из единственной вершины — t .

Для доказательства рассмотрим произвольный s - t разрез C и покажем, что его вес не может быть меньше веса разреза, состоящего из единственной вершины t :

$$w(\{t\}) \leq w(C)$$

Для этого докажем следующий факт. Пусть A_v — состояние множества A на момент добавления вершины v (непосредственно перед добавлением). Пусть C_v — разрез множества $A_v \cup v$, индуцированный разрезом C (проще говоря, C_v равно пересечению этих двух множеств вершин). Далее, вершина v называется активной (по отношению к разрезу C), если вершина v и предыдущая добавленная в A вершина принадлежат разным частям разреза C . Тогда, утверждается, для любой активной вершины v выполняется неравенство:

$$w(v, A_v) \leq w(C_v)$$

В частности, t является активной вершиной (т.к. перед ним добавлялась вершина s), и при $v = t$ это неравенство превращается в утверждение теоремы:

$$w(t, A_t) = w(\{t\}) \leq w(C_t) = w(C)$$

Итак, будем доказывать неравенство, для чего воспользуемся методом математической индукции.

Для первой активной вершины v это неравенство верно (более того, оно обращается в равенство) — поскольку все вершины A_v принадлежат одной части разреза, а v — другой.

Пусть теперь это неравенство выполнено для всех активных вершин вплоть до некоторой вершины v , докажем его для следующей активной вершины u . Для этого преобразуем левую часть:

$$w(u, A_u) \equiv w(u, A_v) + w(u, A_u \setminus A_v)$$

Во-первых, заметим, что:

$$w(u, A_v) \leq w(v, A_v)$$

это следует из того, что когда множество A было равно A_v , в него была добавлена именно вершина v , а не u , значит, она имела наибольшее значение w .

Далее, поскольку $w(v, A_v) \leq w(C_v)$ по предположению индукции, то получаем:

$$w(u, A_v) \leq w(C_v)$$

откуда

$$w(u, A_u) \leq w(C_v) + w(u, A_u \setminus A_v)$$

Теперь заметим, что вершина u и все вершины $A_u \setminus A_v$ находятся в разных частях разреза C , поэтому эта величина $w(u, A_u \setminus A_v)$ обозначает сумму весов рёбер, которые учтены в $w(C_u)$, но не ещё не были учтены в $w(C_v)$, откуда получаем:

$$w(u, A_u) \leq w(C_v) + w(u, A_u \setminus A_v) \leq w(C_u)$$

что и требовалось доказать.

Мы доказали соотношение $w(v, A_v) \leq w(C_v)$, а из него, как уже говорилось выше, следует и вся теорема.

Реализация

Для наиболее простой и ясной реализации (с асимптотикой $O(n^3)$) было выбрано представление графа в виде матрицы смежности. Ответ хранится в переменных `best_cost` и `best_cut` (искомые стоимость минимального разреза и сами вершины, содержащиеся в нём).

Для каждой вершины в массиве `exist` хранится, существует ли она, или она была объединена с какой-то другой вершиной. В списке `v[i]` для каждой сжатой вершины i хранятся номера исходных вершин, которые были скаты в эту вершину i .

Алгоритм состоит из $n - 1$ фазы (цикл по переменной `ph`). На каждой фазе сначала все вершины находятся вне множества A , для чего массив `in_a` заполняется нулями, и связности w всех вершин нулевые. На каждой из $n - ph$ итерации находится вершина `sel` с наибольшей величиной w . Если это итерация последняя, то ответ, если надо, обновляется, а предпоследняя `prev` и последняя `sel` выбранные вершины объединяются в одну. Если итерация не последняя, то `sel` добавляется в множество A , после чего пересчитываются веса всех остальных вершин.

Следует заметить, что алгоритм в ходе своей работы "портит" граф G , поэтому, если он ещё понадобится позже, надо сохранять его копию перед вызовом функции.

```
const int MAXN = 500;
int n, g[MAXN][MAXN];
int best_cost = 10000000000;
vector<int> best_cut;

void mincut() {
    vector<int> v[MAXN];
    for (int i=0; i<n; ++i)
        v[i].assign (1, i);
    int w[MAXN];
    bool exist[MAXN], in_a[MAXN];
    memset (exist, true, sizeof exist);
    for (int ph=0; ph<n-1; ++ph) {
        memset (in_a, false, sizeof in_a);
        memset (w, 0, sizeof w);
        for (int it=0, prev; it<n-ph; ++it) {
            int sel = -1;
            for (int i=0; i<n; ++i)
                if (exist[i] && !in_a[i] && (sel == -1 || w[i] > w[sel]))
                    sel = i;
            if (it == n-ph-1) {
                if (w[sel] < best_cost)
                    best_cost = w[sel], best_cut = v[sel];
                v[prev].insert (v[prev].end(), v[sel].begin(), v[sel].end());
                for (int i=0; i<n; ++i)
                    g2[prev][i] = g2[i][prev] += g2[sel][i];
                exist[sel] = false;
            }
            else {
                in_a[sel] = true;
                for (int i=0; i<n; ++i)
                    w[i] += g2[sel][i];
                prev = sel;
            }
        }
    }
}
```

Литература

- Mechthild Stoer, Frank Wagner. [A Simple Min-Cut Algorithm](#) [1997]
- Kurt Mehlhorn, Christian Uhrig. [The minimum cut algorithm of Stoer and Wagner](#) [1995]

Поток минимальной стоимости, циркуляция минимальной стоимости. Алгоритм удаления циклов отрицательного веса

Постановка задач

Пусть G — сеть (network), то есть ориентированный граф, в котором выбраны вершины-исток s и сток t . Множество вершин обозначим через V , множество рёбер — через E . Каждому ребру $(i, j) \in E$ сопоставлены его пропускная способность $u_{ij} \geq 0$ и стоимость единицы потока c_{ij} . Если какого-то ребра (i, j) в графе нет, то предполагается, что $u_{ij} = c_{ij} = 0$.

Потоком (flow) в сети G называется такая действительнозначная функция f , сопоставляющая каждой паре вершин (i, j) поток f_{ij} между ними, и удовлетворяющая трём условиям:

- Ограничение пропускной способности (выполняется для любых $i, j \in V$):

$$f_{ij} \leq u_{ij}$$

- Антисимметричность (выполняется для любых $i, j \in V$):

$$f_{ij} = -f_{ji}$$

- Сохранение потока (выполняется для любых $i \in V$, кроме $i = s, i = t$):

$$\sum_{j \in V} f_{ij} = 0$$

Величиной потока называется величина

$$|f| = \sum_{i \in V} f_{si}$$

Стоимостью потока называется величина

$$z(f) = \sum_{i, j \in V} c_{ij} f_{ij}$$

Задача нахождения **потока минимальной стоимости** заключается в том, что по заданной величине потока $|f|$ требуется найти поток, обладающий минимальной стоимостью $z(f)$. Стоит обратить внимание на то, что стоимости c_{ij} , приписанные рёбрам, отвечают за стоимость единицы потока вдоль этого ребра; иногда встречается задача, когда рёбрам сопоставляются стоимости протекания потока вдоль этого ребра (т.е. если протекает поток любой величины, то взимается эта стоимость, независимо от величины потока) — эта задача не имеет ничего общего с рассматриваемой здесь и, более того, является NP-полной.

Задача нахождения **максимального потока минимальной стоимости** заключается в том, чтобы найти поток наибольшей величины, а среди всех таких — с минимальной стоимостью. В частном случае, когда веса всех рёбер одинаковы, эта задача становится эквивалентной обычной задаче о максимальном потоке.

Задача нахождения **циркуляции минимальной стоимости** заключается в том, чтобы найти поток нулевой величины с минимальной стоимостью. Если все стоимости неотрицательные, то, понятно, ответом будет нулевой поток $f_{ij} = 0$; если же есть рёбра отрицательного веса (а, точнее, циклы отрицательного веса), то даже при нулевом потоке возможно найти поток отрицательной стоимости. Задачу нахождения циркуляции минимальной стоимости можно, разумеется, поставить и на сети без истока и стока, поскольку никакой смысловой нагрузки они не несут (впрочем, в такой график можно добавить исток и сток в виде изолированных вершин и получить обычную по формулировке задачу). Иногда ставится задача нахождения циркуляции максимальной стоимости — понятно, достаточно изменить стоимости рёбер на противоположные и получим задачу нахождения циркуляции уже минимальной стоимости.

Все эти задачи, разумеется, можно перенести и на неориентированные графы. Впрочем, перейти от неориентированного графа к ориентированному легко: каждое неориентированное ребро (i, j) с пропускной способностью u_{ij} и стоимостью c_{ij} следует заменить двумя ориентированными рёбрами (i, j) и (j, i) с одинаковыми пропускными способностями и стоимостями.

Остаточная сеть

Концепция **остаточной сети** G^f основана на следующей простой идее. Пусть есть некоторый поток f ; вдоль каждого ребра $(i, j) \in E$ протекает некоторый поток $f_{ij} \leq u_{ij}$. Тогда вдоль этого ребра можно (теоретически) пустить ещё $u_{ij} - f_{ij}$ единиц потока; эту величину и назовём **остаточной пропускной способностью**:

$$r_{ij}^f = u_{ij} - f_{ij}$$

Стоимость этих дополнительных единиц потока будет такой же:

$$c_{ij}^f = c_{ij}$$

Однако помимо этого, **прямого** ребра (i, j) , в остаточной сети G^f появляется и **обратное ребро** (j, i) . Интуитивный смысл этого ребра в том, что мы можем в будущем отменить часть потока, протекавшего по ребру (i, j) . Соответственно, пропускание потока вдоль этого обратного ребра (j, i) фактически, и формально, означает уменьшение потока вдоль ребра (i, j) . Обратное ребро имеет пропускную способность, равную нулю (чтобы, например, при $f_{ij} = 0$ и по обратному ребру невозможно было бы пропустить поток; при положительной

величине $f_{ij} > 0$ для обратного ребра по свойству антисимметричности станет $f_{ji} < 0$, что меньше $c_{ji}^f = 0$, т.е. можно будет пропускать какой-то поток вдоль обратного ребра), остаточную пропускную способность — равную потоку вдоль прямого ребра, а стоимость — противоположную (ведь после отмены части потока мы должны соответственно уменьшить и стоимость):

$$\begin{aligned} u_{ji}^f &= 0 \\ r_{ji}^f &= f_{ij} \\ c_{ji}^f &= -c_{ij} \end{aligned}$$

Таким образом, каждому ориентированному ребру в G соответствует два ориентированных ребра в остаточной сети G^f , и у каждого ребра остаточной сети появляется дополнительная характеристика — остаточная пропускная способность. Впрочем, нетрудно заметить, что выражения для остаточной пропускной способности r_{ij}^f по сути одинаковы как для прямого, так и для обратного ребра, т.е. мы можем записать для любого ребра (i, j) остаточной сети:

$$r_{ij}^f = u_{ij}^f - f_{ij}^f$$

Кстати, при реализации это свойство позволяет не хранить остаточные пропускные способности, а просто вычислять их при необходимости для ребра.

Следует отметить, что из остаточной сети удаляются все рёбра, имеющие нулевую остаточную пропускную способность. Остаточная сеть G^f должна содержать **только рёбра с положительной остаточной пропускной способностью** r_{ij}^f .

Здесь стоит обратить внимание на такой важный момент: если в сети G были одновременно оба ребра (i, j) и (j, i) , то в остаточной сети у каждого из них появится по обратному ребру, и в итоге появятся **кратные рёбра**. Например, такая ситуация часто возникает, когда сеть строится по неориентированному графу (и, получается, каждое неориентированное ребро в итоге приведёт к появлению четырёх рёбер в остаточной сети). Эту особенность нужно всегда помнить, она приводит к небольшому усложнению программирования, хотя в общем ничего не меняет. Кроме того, обозначение ребра (i, j) в таком случае становится неоднозначным, поэтому ниже мы везде будем считать, что такой ситуации в сети нет (исключительно в целях простоты и корректности описаний); на правильность идеи это никак не влияет.

Критерий оптимальности по наличию циклов отрицательного веса

Теорема. Некоторый поток f является оптимальным (т.е. имеет наименьшую стоимость среди всех потоков такой же величины) тогда и только тогда, когда остаточная сеть G^f не содержит циклов отрицательного веса.

Доказательство: необходимость. Пусть поток f является оптимальным. Предположим, что остаточная сеть G^f содержит цикл отрицательного веса. Возьмём этот цикл отрицательного веса и выберем минимум k среди остаточных пропускных способностей рёбер этого цикла (k будет больше нуля). Но тогда можно увеличить поток вдоль каждого ребра цикла на величину k , при этом никакие свойства потока не нарушаются, величина потока не изменится, однако стоимость потока уменьшится (уменьшится на стоимость цикла, умноженную на k). Таким образом, если есть цикл отрицательного веса, то f не может быть оптимальным, ч.т.д.

Доказательство: достаточность. Для этого сначала докажем вспомогательные факты.

Лемма 1 (о декомпозиции потока): любой поток f можно представить в виде совокупности путей из истока в сток и циклов, все — имеющие положительный поток. Докажем эту лемму конструктивно: покажем, как разбить поток на совокупность путей и циклов. Если поток имеет ненулевую величину, то, очевидно, из истока s выходит хотя бы одно ребро с положительным потоком; пройдём по этому ребру, окажемся в какой-то вершине v_1 . Если эта вершина $v_1 = t$, то останавливаемся — нашли путь из s в t . Иначе, по свойству сохранения потока, из v_1 должно выходить хотя бы одно ребро с положительным потоком; пройдём по нему в какую-то вершину v_2 . Повторяя этот процесс, мы либо придём в сток t , либо же придём в какую-то вершину во второй раз. В первом случае мы обнаружим путь из s в t , во втором — цикл. Найденный путь/цикл будет иметь положительный поток k (минимум из потоков рёбер этого пути/цикла). Тогда уменьшим поток вдоль каждого ребра этого пути/цикла на величину k , в результате получим снова поток, к которому снова применим этот процесс. Рано или поздно поток вдоль всех рёбер станет нулевым, и мы найдём его декомпозицию на пути и циклы.

Лемма 2 (о разности потоков): для любых двух потоков f и g одной величины ($|f| = |g|$) поток g можно представить как поток f плюс несколько циклов в остаточной сети G^f . Действительно, рассмотрим разность этих потоков $g - f$ (вычитание потоков — это почлененное вычитание, т.е. вычитание потоков вдоль каждого ребра). Нетрудно убедиться, что в результате получится некоторый поток нулевой величины, т.е. циркуляция. Произведём декомпозицию этой циркуляции согласно предыдущей лемме. Очевидно, это декомпозиция не может содержать путей (т.к. наличие s - t -пути с положительным потоком означает, что и величина потока в сети положительна). Таким образом, разность потоков g и f можно представить в виде суммы циклов в сети G . Более того, это будут и циклы в остаточной сети G^f , т.к.

$$g_{ij} - f_{ij} \leq u_{ij} - f_{ij} = r_{ij}^f, \text{ ч.т.д.}$$

Теперь, вооружившись этими леммами, мы легко можем **доказать достаточность**. Итак, рассмотрим произвольный поток f , в остаточной сети которого нет циклов отрицательной стоимости. Рассмотрим также поток той же величины, но минимальной стоимости f^* ; докажем, что f и f^* имеют одинаковую стоимость. Согласно лемме 2, поток f^* можно представить в виде суммы потока f и нескольких циклов. Но раз стоимости всех циклов неотрицательны, то и стоимость потока f^* не может быть меньше стоимости потока f : $z(f^*) \geq z(f)$. С другой стороны, т.к. поток f^* является оптимальным, то его стоимость не может быть выше стоимости потока f . Таким образом, $z(f) = z(f^*)$, ч.т.д.

Алгоритм удаления циклов отрицательного веса

Только что доказанная теорема даёт нам простой **алгоритм**, позволяющий найти поток минимальной стоимости: если у нас есть какой-то поток f , то построить для него остаточную сеть, проверить, есть ли в ней цикл отрицательного веса. Если такого цикла нет, то поток f является оптимальным (имеет наименьшую стоимость среди всех потоков такой же величины). Если же был найден цикл отрицательной стоимости, то посчитать поток k , который можно пропустить дополнительно через этот цикл (это k будет равно минимуму из остаточных пропускных способностей рёбер цикла). Увеличив поток на k вдоль каждого ребра цикла, мы, очевидно, не нарушим свойства потока, не изменим величину потока, но уменьшим стоимость этого потока, получив новый поток f' , для которого надо повторить весь процесс.

Таким образом, чтобы запустить процесс улучшения стоимости потока, нам предварительно нужно найти **произвольный поток нужной величины** (каким-нибудь стандартным алгоритмом нахождения максимального потока, см., например, [алгоритм Эдмонда-Карпа](#)). В частности, если требуется найти циркуляцию наименьшей стоимости, то начать можно просто с нулевого потока.

Оценим **асимптотику** алгоритма. Поиск цикла отрицательной стоимости в графе с n вершинами и m рёбрами производится за $O(nm)$ (см. [соответствующую статью](#)). Если мы обозначим через C наибольшее из стоимостей рёбер, через U — наибольшую из пропускных способностей, то максимальное значение стоимости потока не превосходит mCU . Если все стоимости и пропускные способности — целые

числа, то каждая итерация алгоритма уменьшает стоимость потока как минимум на единицу; следовательно, всего алгоритм совершил $O(mCU)$ итераций, а итоговая асимптотика составит:

$$O(nm^2CU)$$

Эта асимптотика — не строго полиномиальна (strong polynomial), поскольку зависит от величин пропускных способностей и стоимостей.

Впрочем, если искать не произвольный отрицательный цикл, а использовать какой-то более чёткий подход, то асимптотика может значительно уменьшиться. Например, если каждый раз искать цикл с минимальной средней стоимостью (что также можно производить за $O(nm)$), то время работы всего алгоритма уменьшится до $O(nm^2 \log n)$, и эта асимптотика уже является строго полиномиальной.

Реализация

Сначала введём структуры данных и функции для хранения графа. Каждое ребро хранится в отдельной структуре `edge`, все рёбра лежат в общем списке `edges`, а для каждой вершины i в векторе `g[i]` хранятся номера рёбер, выходящих из неё. Такая организация позволяет легко находить номер обратного ребра по номеру прямого ребра — они оказываются в списке `edges` соседними, и номер одного можно получить по номеру другого операцией "`^1`" (она инвертирует младший бит). Добавление ориентированного ребра в граф осуществляется функция `add_edge`, которая добавляет сразу прямое и обратное рёбра.

```
const int MAXN = 100*2;
int n;
struct edge {
    int v, to, u, f, c;
};
vector<edge> edges;
vector<int> g[MAXN];

void add_edge (int v, int to, int cap, int cost) {
    edge e1 = { v, to, cap, 0, cost };
    edge e2 = { to, v, 0, 0, -cost };
    g[v].push_back ((int) edges.size());
    edges.push_back (e1);
    g[to].push_back ((int) edges.size());
    edges.push_back (e2);
}
```

В основной программе после чтения графа идёт бесконечный цикл, внутри которого выполняется алгоритм Форда-Беллмана, и если он обнаруживает цикл отрицательной стоимости, то вдоль этого цикла увеличивается поток. Поскольку остаточная сеть может представлять собой несвязный граф, то алгоритм Форда-Беллмана запускается из каждой не достигнутой ещё вершины. В целях оптимизации алгоритм использует очередь (текущая очередь `q` и новая очередь `nq`), чтобы не перебирать на каждой стадии все рёбра. Вдоль обнаруженного цикла каждый раз проталкивается ровно единица потока, хотя, понятно, в целях оптимизации величину потока можно определять как минимум остаточных пропускных способностей.

```
const int INF = 1000000000;
for (;;) {
    bool found = false;

    vector<int> d (n, INF);
    vector<int> par (n, -1);
    for (int i=0; i<n; ++i)
        if (d[i] == INF) {
            d[i] = 0;
            vector<int> q, nq;
            q.push_back (i);
            for (int it=0; it<n && q.size(); ++it) {
                nq.clear();
                sort (q.begin(), q.end());
                q.erase (unique (q.begin(), q.end()), q.end());
                for (size_t j=0; j<q.size(); ++j) {
                    int v = q[j];
                    for (size_t k=0; k<g[v].size(); ++k) {
                        int id = g[v][k];
                        if (edges[id].f < edges[id].u)
                            if (d[v] + edges[id].c < d[edges[id].to]) {
                                d[edges[id].to] = d[v] + edges[id].c;
                                par[edges[id].to] = v;
                                nq.push_back (edges[id].to);
                            }
                    }
                }
                swap (q, nq);
            }
            if (q.size()) {
                int leaf = q[0];
                vector<int> path;
                for (int v=leaf; v!= -1; v=par[v])
                    if (find (path.begin(), path.end(), v) == path.end())
                        path.push_back (v);
                    else {
                        path.erase (path.begin(), find (path.begin(), path.end(), v));
                        break;
                    }
                for (size_t j=0; j<path.size(); ++j)
                    int to = path[j], v = path[(j+1)%path.size()];
```

```

        for (size_t k=0; k<g[v].size(); ++k)
            if (edges[ g[v][k] ].to == to) {
                int id = g[v][k];
                edges[id].f += 1;
                edges[id^1].f -= 1;
            }
        found = true;
    }

    if (!found) break;
}

```

Литература

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. **Алгоритмы: Построение и анализ** [2005]
- Ravindra Ahuja, Thomas Magnanti, James Orlin. **Network flows** [1993]
- Andrew Goldberg, Robert Tarjan. **Finding Minimum-Cost Circulations by Cancelling Negative Cycles** [1989]

Алгоритм Диница

Постановка задачи

Пусть дана сеть, т.е. ориентированный граф G , в котором каждому ребру (u, v) приписана пропускная способность c_{uv} , а также выделены две вершины — исток s и сток t . Требуется найти в этой сети поток f_{uv} из истока s в сток t максимальной величины.

Немного истории

Этот алгоритм был опубликован советским (израильским) учёным Ефимом **Диницем** (Yefim Dinic, иногда Dinitz) в 1970 г., т.е. даже на два года раньше опубликования алгоритма Эдмондса-Карпа (впрочем, оба алгоритма были независимо открыты в 1968 г.).

Кроме того, следует отметить, что некоторые упрощения алгоритма были произведены Шимоном Ивеном (Shimon Even) и его учеником Алоном Итаи (Alon Itai) в 1979 г. Именно благодаря им алгоритм получил свой современный облик: они применили к идеи Диница концепцию блокирующих потоков Александра Карзанова (Alexander Karzanov, 1974 г.), а также переформулировали алгоритм к той комбинации обхода в ширину и в глубину, в которой сейчас этот алгоритм и излагается везде.

Развитие идей по отношению к потоковым алгоритмам крайне интересно рассматривать, учитывая "железный занавес" тех лет, разделявший СССР и Запад. Видно, как иногда похожие идеи появлялись почти одновременно (как в случае алгоритма Диница и алгоритма Эдмондса-Карпа), правда, имея при этом разную эффективность (алгоритм Диница на один порядок быстрее); иногда же, наоборот, появление идеи по одну сторону "занавеса" опережало аналогичный ход по другую сторону более чем на десятилетие (как алгоритм Карзанова проталкивания в 1974 г. и алгоритм Гольдберга (Goldberg) проталкивания в 1985 г.).

Необходимые определения

Введём три необходимых определения (каждое из них является независимым от остальных), которые затем будут использоваться в алгоритме Диница.

Остаточной сетью G^R по отношению к сети G и некоторому потоку f в ней называется сеть, в которой каждому ребру $(u, v) \in G$ с пропускной способностью c_{uv} и потоком f_{uv} соответствуют два ребра:

- (u, v) с пропускной способностью $c_{uv}^R = c_{uv} - f_{uv}$
- (v, u) с пропускной способностью $c_{vu}^R = f_{uv}$

Стоит отметить, что при таком определении в остаточной сети могут появляться кратные рёбра: если в исходной сети было как ребро (u, v) , так и (v, u) .

Остаточное ребро можно интуитивно понимать как меру того, насколько ещё можно увеличить поток вдоль какого-то ребра. В самом деле, если по ребру (u, v) с пропускной способностью c_{uv} протекает поток f_{uv} , то потенциально по нему можно пропустить ещё $c_{uv} - f_{uv}$ единиц потока, а в обратную сторону можно пропустить до f_{uv} единиц потока, что будет означать отмену потока в первоначальном направлении.

Блокирующим потоком в данной сети называется такой поток, что любой путь из истока s в сток t содержит насыщено этим потоком ребро. Иными словами, в данной сети не найдётся такого пути из истока в сток, вдоль которого можно беспрепятственно увеличить поток.

Блокирующий поток не обязательно максимальен. Теорема Форда-Фалкерсона говорит о том, что поток будет максимальным тогда и только тогда, когда в остаточной сети не найдётся $s - t$ пути; в блокирующем же потоке ничего не утверждается о существовании пути по рёбрам, появляющимся в остаточной сети.

Слоистая сеть для данной сети строится следующим образом. Сначала определяются длины кратчайших путей из истока s до всех остальных вершин; назовём уровнем $\text{level}[v]$ вершины её расстояние от истока. Тогда в слоистую сеть включают все те рёбра (u, v) исходной сети, которые ведут с одного уровня на какой-либо другой, более поздний, уровень, т.е. $\text{level}[u] + 1 = \text{level}[v]$ (почему в этом

случае разница расстояний не может превосходить единицы, следует из свойства кратчайших расстояний). Таким образом, удаляются все рёбра, расположенные целиком внутри уровней, а также рёбра, ведущие назад, к предыдущим уровням.

Очевидно, слоистая сеть ациклична. Кроме того, любой $s - t$ путь в слоистой сети является кратчайшим путём в исходной сети.

Построить слоистую сеть по данной сети очень легко: для этого надо запустить обход в ширину по рёбрам этой сети, посчитав тем самым для каждой вершины величину $\text{level}[]$, и затем внести в слоистую сеть все подходящие рёбра.

Примечание. Термин "слоистая сеть" в русскоязычной литературе не употребляется; обычно эта конструкция называется просто "вспомогательным графом". Связано это с трудностью перевода исходного термина "layered network".

Алгоритм

Схема алгоритма

Алгоритм представляет собой несколько фаз. На каждой фазе сначала строится остаточная сеть, затем по отношению к ней строится слоистая сеть (обходом в ширину), а в ней ищется произвольный блокирующий поток. Найденный блокирующий поток прибавляется к текущему потоку, и на этом очередная итерация заканчивается.

Этот алгоритм схож с алгоритмом Эдмондса-Карпа, но основное отличие можно понимать так: на каждой итерации поток увеличивается не вдоль одного кратчайшего $s - t$ пути, а вдоль целого набора таких путей (ведь именно такими путями являются пути в блокирующем потоке слоистой сети).

Корректность алгоритма

Покажем, что если алгоритм завершается, то на выходе у него получается поток именно максимальной величины.

В самом деле, предположим, что в какой-то момент в слоистой сети, построенной для остаточной сети, не удалось найти блокирующий поток. Это означает, что сток t вообще не достижим в слоистой сети из истока s . Но поскольку слоистая сеть содержит в себе все кратчайшие пути из истока в остаточной сети, это в свою очередь означает, что в остаточной сети нет пути из истока в сток. Следовательно, применяя теорему Форда-Фалкерсона, получаем, что текущий поток в самом деле максимальен.

Оценка числа фаз

Покажем, что алгоритм Диница всегда выполняет **менее n фаз**. Для этого докажем две леммы:

Лемма 1. Кратчайшее расстояние от истока до каждой вершины не уменьшается с выполнением каждой итерации, т.е.

$$\text{level}_{i+1}[v] \geq \text{level}_i[v]$$

где нижний индекс обозначает номер фазы, перед которой взяты значения этих переменных.

Доказательство. Зафиксируем произвольную фазу i и произвольную вершину v и рассмотрим любой кратчайший $s - v$ путь P в сети G_{i+1}^R (напомним, так мы обозначаем остаточную сеть, взятую перед выполнением $i + 1$ -ой фазы). Очевидно, длина пути P равна $\text{level}_{i+1}[v]$.

Заметим, что в остаточную сеть G_{i+1}^R могут входить только рёбра из G^R , а также рёбра, обратные рёбрам из G^R (это следует из определения остаточной сети). Рассмотрим два случая:

- Путь P содержит только рёбра из G^R . Тогда, понятно, длина пути P больше либо равна $\text{level}_i[v]$ (потому что $\text{level}_i[v]$ по определению — длина кратчайшего пути), что и означает выполнение неравенства.
- Путь P содержит как минимум одно ребро, не содержащееся в G^R (но обратное какому-то ребру из G^R). Рассмотрим первое такое ребро; пусть это будет ребро (u, w) .

$$s \Rightarrow u \rightarrow w \Rightarrow v$$

Мы можем применить нашу лемму к вершине u , потому что она подпадает под первый случай; итак, мы получаем неравенство $\text{level}_{i+1}[u] \geq \text{level}_i[u]$.

Теперь заметим, что поскольку ребро (u, w) появилось в остаточной сети только после выполнения i -ой фазы, то отсюда следует, что вдоль ребра (w, u) был дополнительно пропущен какой-то поток; следовательно, ребро (w, u) принадлежало слоистой сети перед i -ой фазой, а потому $\text{level}_i[u] = \text{level}_i[w] + 1$. Учтём, что по свойству кратчайших путей $\text{level}_{i+1}[w] = \text{level}_{i+1}[u] + 1$, и объединяя это равенство с двумя предыдущими неравенствами, получаем:

$$\text{level}_{i+1}[w] \geq \text{level}_i[w] + 2.$$

Теперь мы можем применять те же самые рассуждения ко всему оставшемуся пути до v (т.е. что каждое инвертированное ребро добавляет к level как минимум два), и в итоге получим требуемое неравенство.

Лемма 2. Расстояние между истоком и стоком строго увеличивается после каждой фазы алгоритма, т.е.:

$$\text{level}'[t] > \text{level}[t],$$

где штрихом помечено значение, полученное на следующей фазе алгоритма.

Доказательство: от противного. Предположим, что после выполнения текущей фазы оказалось, что $\text{level}'[t] = \text{level}[t]$. Рассмотрим кратчайший путь из истока в сток; по предположению, его длина должна сохраниться неизменной. Однако остаточная сеть на следующей фазе содержит только рёбра остаточной сети перед выполнением текущей фазы, либо обратные к ним. Таким образом, пришли к противоречию: нашёлся $s - t$ путь, который не содержит насыщенных рёбер, и имеет ту же длину, что и кратчайший путь. Этот путь должен был быть "заблокирован" блокирующим потоком, чего не произошло, в чём и заключается противоречие, что и требовалось доказать.

Эту лемму интуитивно можно понимать следующим образом: на i -ой фазе алгоритм Диница выявляет и насыщает все $s - t$ пути длины i .

Поскольку длина кратчайшего пути из s в t не может превосходить $n - 1$, то, следовательно, алгоритм Диница совершаёт **не более $n - 1$ фазы**.

Поиск блокирующего потока

Чтобы завершить построение алгоритма Диница, надо описать алгоритм нахождения блокирующего потока в слоистой сети — ключевое

место алгоритма.

- Искать $s - t$ пути по одному, пока такие пути находятся. Путь можно найти за $O(m)$ обходом в глубину, а всего таких путей будет $O(m)$ (поскольку каждый путь насыщает как минимум одно ребро). Итоговая асимптотика поиска одного блокирующего потока составит $O(m^2)$.
- Аналогично предыдущей идее, однако удалять в процессе обхода в глубину из графа все "лишние" рёбра, т.е. рёбра, вдоль которых не получится дойти до стока.

Это очень легко реализовать: достаточно удалять ребро после того, как мы просмотрели его в обходе в глубину (кроме того случая, когда мы прошли вдоль ребра и нашли путь до стока). С точки зрения реализации, надо просто поддерживать в списке смежности каждой вершины указатель на первое неудалённое ребро, и увеличивать этот указатель в цикле внутри обхода в глубину.

Оценим асимптотику этого решения. Каждый обход в глубину завершается либо насыщением как минимум одного ребра (если этот обход достиг стока), либо продвижением вперёд как минимум одного указателя (в противном случае). Можно понять, что один запуск обхода в глубину из основной программы работает за $O(k + n)$, где k — число продвижений указателей. Учитывая, что всего запусков обхода в глубину в рамках поиска одного блокирующего потока будет $O(p)$, где p — число рёбер, насыщенных этим блокирующим потоком, то весь алгоритм поиска блокирующего потока отработает за $O(pk + pn)$, что, учитывая, что все указатели в сумме прошли расстояние $O(m)$, даёт асимптотику $O(m + pn)$. В худшем случае, когда блокирующий поток насыщает все рёбра, асимптотика получается $O(nm)$; эта асимптотика будет использоваться далее.

Можно сказать, что этот способ нахождения блокирующего потока чрезвычайно эффективен в том смысле, что на поиск одного увеличивающего пути он тратит $O(n)$ операций в среднем. Именно в этом и кроется разность на целый порядок эффективностей алгоритма Диница и Эммондса-Карпа (который ищёт один увеличивающий путь за $O(m)$).

Этот способ решения является по-прежнему простым для реализации, но достаточно эффективным, и потому наиболее часто применяется на практике.

- Можно применить специальные структуры данных — динамические деревья Слетора (Sleator) и Тарьяна (Tarjan)). Тогда каждый блокирующий поток можно найти за время $O(m \log n)$.

Асимптотика

Таким образом, весь алгоритм Диница выполняется за $O(n^2m)$, если блокирующий поток искать описанным выше способом за $O(nm)$. Реализация с использованием динамических деревьев Слетора и Тарьяна будет работать за время $O(nm \log n)$.

Единичные сети

Единичной называется такая сеть, в которой все пропускные способности равны 0 либо 1, и у любой вершины либо входящее, либо исходящее ребро единственно.

Этот случай является достаточно важным, поскольку в задаче поиска **максимального паросочетания** построенная сеть является именно единичной.

Докажем, что на единичных сетях алгоритм Диница даже в простой реализации (которая на произвольных графах отрабатывает за $O(n^2m)$) работает за время $O(m\sqrt{n})$, достигая на задаче поиска наибольшего паросочетания наилучший из известных алгоритмов — алгоритм Хопкрофта-Карпа. Чтобы доказать эту оценку, надо рассмотреть два случая:

- Если величина искомого потока не превосходит \sqrt{n} , то, значит, число фаз и запусков обхода в глубину есть величина $O(\sqrt{n})$. Вспоминая, что одна фаза в этой реализации работает за $O(m + pn)$, получаем итоговую асимптотику $O(\sqrt{nm} + \sqrt{pn}) = O(\sqrt{nm})$.
- Если величина $|f|$ искомого потока больше \sqrt{n} . Заметим, что поток в единичной сети можно представить в виде суммы $|f|$ вершинно-непересекающихся путей, а потому максимальная длина $s - t$ пути имеет величину $O(\sqrt{n})$. Учитывая, что одна фаза алгоритма Диница целиком обрабатывает все пути какой-либо длины, мы снова получаем, что число фаз есть величина $O(\sqrt{n})$. Суммируя асимптотику одной фазы $O(m + pn)$ по всем фазам, получаем $O(\sqrt{nm} + n^2) = O(\sqrt{nm})$, что и требовалось доказать.

Реализация

Приведём две реализации алгоритма за $O(n^2m)$, работающие на сетях, заданных матрицами смежности и списками смежности соответственно.

Реализация над графиками в виде матриц смежности

```
const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

int n, c[MAXN][MAXN], f[MAXN][MAXN], s, t, d[MAXN], ptr[MAXN], q[MAXN];

bool bfs() {
    int qh=0, qt=0;
    q[qt++] = s;
    memset(d, -1, n * sizeof d[0]);
    d[s] = 0;
    while (qh < qt) {
        int v = q[qh++];
        for (int to=0; to<n; ++to)
            if (d[to] == -1 && f[v][to] < c[v][to]) {
                q[qt++] = to;
                d[to] = d[v] + 1;
            }
    }
    return d[t] != -1;
}

int dfs (int v, int flow) {
    if (!flow) return 0;
    for (int to=0; to<n; ++to)
        if (d[to] == d[v] + 1 && f[v][to] < c[v][to]) {
            int new_flow = min(flow, c[v][to] - f[v][to]);
            if (dfs(to, new_flow)) {
                f[v][to] += new_flow;
                f[to][v] -= new_flow;
                return new_flow;
            }
        }
    return 0;
}
```

```

if (v == t) return flow;
for (int & to=ptr[v]; to<n; ++to) {
    if (d[to] != d[v] + 1) continue;
    int pushed = dfs (to, min (flow, c[v][to] - f[v][to]));
    if (pushed) {
        f[v][to] += pushed;
        f[to][v] -= pushed;
        return pushed;
    }
}
return 0;
}

int dinic () {
    int flow = 0;
    for (;;) {
        if (!bfs ()) break;
        memset (ptr, 0, n * sizeof ptr[0]);
        while (int pushed = dfs (s, INF))
            flow += pushed;
    }
    return flow;
}

```

Сеть должна быть предварительно считана: должны быть заданы переменные n, s, t , а также считана матрица пропускных способностей $c[\cdot][\cdot]$. Основная функция решения — `dinic()`, которая возвращает величину найденного максимального потока.

Реализация над графами в виде списков смежности

```

const int MAXN = ...; // число вершин
const int INF = 1000000000; // константа-бесконечность

struct edge {
    int a, b, cap, flow;
};

int n, s, t, d[MAXN], ptr[MAXN], q[MAXN];
vector<edge> e;
vector<int> g[MAXN];

void add_edge (int a, int b, int cap) {
    edge e1 = { a, b, cap, 0 };
    edge e2 = { b, a, 0, 0 };
    g[a].push_back ((int) e.size ());
    e.push_back (e1);
    g[b].push_back ((int) e.size ());
    e.push_back (e2);
}

bool bfs () {
    int qh=0, qt=0;
    q[qt++] = s;
    memset (d, -1, n * sizeof d[0]);
    d[s] = 0;
    while (qh < qt && d[t] == -1) {
        int v = q[qh++];
        for (size_t i=0; i<g[v].size (); ++i) {
            int id = g[v][i],
                to = e[id].b;
            if (d[to] == -1 && e[id].flow < e[id].cap) {
                q[qt++] = to;
                d[to] = d[v] + 1;
            }
        }
    }
    return d[t] != -1;
}

int dfs (int v, int flow) {
    if (!flow) return 0;
    if (v == t) return flow;
    for (; ptr[v]<(int)g[v].size (); ++ptr[v]) {
        int id = g[v][ptr[v]],
            to = e[id].b;
        if (d[to] != d[v] + 1) continue;
        int pushed = dfs (to, min (flow, e[id].cap - e[id].flow));
        if (pushed) {
            e[id].flow += pushed;
            e[id^1].flow -= pushed;
            return pushed;
        }
    }
    return 0;
}

```

```

int dinic() {
    int flow = 0;
    for (;;) {
        if (!bfs()) break;
        memset (ptr, 0, n * sizeof ptr[0]);
        while (int pushed = dfs (s, INF))
            flow += pushed;
    }
    return flow;
}

```

Сеть должна быть предварительно считана: должны быть заданы переменные n , s , t , а также добавлены все рёбра (ориентированные) с помощью вызовов функции `add_edge`. Основная функция решения — `dinic()`, которая возвращает величину найденного максимального потока.

Алгоритм Куна нахождения наибольшего паросочетания в двудольном графе

Дан двудольный граф, содержащий N вершин и M рёбер. Требуется найти наибольшее паросочетание, т.е. выбрать как можно больше рёбер, чтобы ни одно выбранное ребро не имело общей вершины ни с каким другим выбранным ребром.

Описание

Цепью длины k назовём некоторый простой путь (т.е. не содержащий повторяющихся вершин или рёбер), содержащий k рёбер. **Чередующаяся цепь** (в двудольном графе, относительно некоторого паросочетания) назовём цепь, в которой рёбра поочередно принадлежат/не принадлежат паросочетанию. **Увеличивающей цепью** (в двудольном графе, относительно некоторого паросочетания) назовём цепь, у которой начальная и конечная вершины не принадлежат паросочетанию.

Теорема Бержа. Паросочетание является максимальным тогда и только тогда, когда не существует увеличивающих относительно него цепей.

Далее, заметим, что если найдена некоторая увеличивающая цепь, то с её помощью легко **увеличить мощность паросочетания на 1**. Пройдём вдоль этой цепи, и каждое ребро поочерёдно будем добавлять/удалять из паросочетания (т.е. первое ребро (которое по определению не принадлежит паросочетанию) добавим в паросочетание, второе удалим, третье добавим, и т.д.). Действительно, в результате этой операции мы увеличим мощность паросочетания на 1 (для этого достаточно заметить, что длина увеличивающей цепи всегда нечётно, а корректность вышеописанного преобразования очевидна).

Таким образом, мы получили **каркас алгоритма** построения максимального паросочетания: искать в графе увеличивающие цепи, пока они существуют, и увеличивать паросочетание вдоль них.

Осталось детализировать способ нахождения увеличивающих цепей. **Алгоритм Куна** основан на **поиске в глубину или в ширину**, и выбирает каждый раз любую из найденных увеличивающих цепей. Стоит заметить, что есть и другие способы, например, в более эффективном алгоритме Хопкрофта-Карпа.

Поскольку каждая увеличивающая цепь будет найдена за $O(N+M)$, а всего цепей понадобится найти не более $N/2$, то итоговая асимптотика алгоритма равна $O(N^2 + NM)$, т.е. $O(NM)$.

Рассмотрим **подробнее** алгоритм поиска увеличивающей цепи (пусть, для определённости, это поиск в глубину). Поиск начинает идти из вершины первой доли. Из первой доли во вторую он ходит только по рёбрам, не принадлежащим паросочетанию, а из второй доли в первую - наоборот, только по принадлежащим. С точки зрения реализации, поиск в глубину всегда находится в вершине первой доли, и он возвращает булево значение - найдена цепь или не найдена. Из текущей вершины V поиск в глубину пытается пойти по всем смежным рёбрам (кроме принадлежащего паросочетанию), и если он может пойти в вершину To второй доли, не принадлежащей паросочетанию, то возвращает `True` и добавляет ребро (V, To) в паросочетание. Если же он пытается пойти в вершину To , уже принадлежащую паросочетанию, то он вызывает себя из вершины $Mt[To]$ (соседа To в паросочетании), и если цепь была найдена, то добавляет ребро (V, To) в паросочетание.

Реализация

Здесь N - число вершин в первой доле, K - во второй доле.

```

int n, k;
vector < vector<int> > g;
vector<int> mt;
vector<char> used;

bool kuhn (int v) {
    if (used[v]) return false;
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (mt[to] == -1 || kuhn (mt[to])) {
            mt[to] = v;

```

```

        return true;
    }
}
return false;
}

int main() {
    ... чтение графа ...

    mt.assign (k, -1);
    for (int i=0; i<n; ++i) {
        used.assign (n, false);
        kuhn (i);
    }

    for (int i=0; i<k; ++i)
        if (mt[i] != -1)
            printf ("%d %d\n", mt[i]+1, i+1);
}

```

Улучшенная реализация

Модифицируем алгоритм следующим образом. До основного цикла алгоритма найдём каким-нибудь простым алгоритмом **произвольное паросочетание** (каким-либо **жадным алгоритмом**), и лишь затем будем выполнять цикл с вызовами функции `kuhn()`, который будет улучшать это паросочетание. В результате алгоритм будет работать заметно быстрее на случайных графах - потому что в большинстве графов можно легко набрать паросочетание достаточно большого веса.

Например, можно просто перебрать все вершины первой доли, и для каждой из них найти произвольное ребро, которое можно добавить в паросочетание, и добавить его. Даже такая простая эвристика способна ускорить алгоритм Куна в несколько раз.

Следует обратить внимание на то, что основной цикл придётся немного модифицировать. Поскольку при вызове функции `kuhn` в основном цикле предполагается, что текущая вершина ещё не входит в паросочетание, то нужно добавить соответствующую проверку.

В реализации изменится только код в функции `main()`:

```

int main() {
    ... чтение графа ...

    mt.assign (k, -1);
    vector<char> used1 (n);
    for (int i=0; i<n; ++i)
        for (size_t j=0; j<g[i].size(); ++j)
            if (mt[g[i][j]] == -1) {
                mt[g[i][j]] = i;
                used1[i] = true;
                break;
            }
    for (int i=0; i<n; ++i)
        if (used1[i]) continue;
        used.assign (n, false);
        kuhn (i);

    for (int i=0; i<k; ++i)
        if (mt[i] != -1)
            printf ("%d %d\n", mt[i]+1, i+1);
}

```

Другой хорошей эвристикой является следующая. На каждом шаге будет искать вершину наименьшей степени (но не изолированную), из неё выбирать любое (?) ребро и добавлять его в паросочетание, затем удаляя обе эти вершины со всеми инцидентными им рёбрами из графа. Такая жадность работает очень хорошо на случайных графах, даже в большинстве случаев строит максимальное паросочетание, однако на специально подобранных тестах она может найти паросочетание значительно меньшей величины, чем максимальное.

Проверка графа на двудольность и разбиение на две доли

Пусть дан неориентированный граф. Требуется проверить, является ли он двудольным, т.е. можно ли разделить его вершины на две доли так, чтобы не было рёбер, соединяющих две вершины одной доли. Если граф является двудольным, то вывести сами доли.

Решим эту задачу с помощью [поиска в ширину](#) за $O(M)$.

Признак двудольности

Теорема. Граф является двудольным тогда и только тогда, когда все его простые циклы имеют чётную длину.

Впрочем, с практической точки зрения искать все простые циклы неудобно. Намного проще проверять граф на двудольность следующим алгоритмом:

Алгоритм

Произведём серию поисков в ширину. Т.е. будем запускать поиск в ширину из каждой непосещённой вершины. Ту вершину, из которой мы начинаем идти, мы помещаем в первую долю. В процессе поиска в ширину, если мы идём в какую-то новую вершину, то мы помещаем её в долю, отличную от доли текущей вершины. Если же мы пытаемся пройти по ребру в вершину, которая уже посещена, то мы проверяем, чтобы эта вершина и текущая вершина находились в разных долях. В противном случае граф двудольным не является.

По окончании работы алгоритма мы либо обнаружим, что граф не двудолен, либо найдём разбиение вершин графа на две доли.

Реализация

```
int n;
vector < vector<int> > g;
... чтение графа ...

vector<char> part (n, -1);
bool ok = true;
vector<int> q (n);
for (int st=0; st<n; ++st)
    if (part[st] == -1) {
        int h=0, t=0;
        q[t++] = st;
        part[st] = 0;
        while (h<t) {
            int v = q[h++];
            for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (part[to] == -1)
                    part[to] = !part[v], q[t++] = to;
                else
                    ok &= part[to] != part[v];
            }
        }
    }
puts (ok ? "YES" : "NO");
```

Нахождение наибольшего по весу вершинно-взвешенного паросочетания

Дан двудольный граф G. Для каждой вершины первой доли указан её вес. Требуется найти паросочетание наибольшего веса, т.е. с наибольшей суммой весов насыщенных вершин.

Ниже мы опишем и докажем алгоритм, основанный на [алгоритме Куна](#), который будет находить оптимальное решение.

Алгоритм

Сам алгоритм чрезвычайно прост. Отсортируем вершины первой доли в порядке убывания (точнее говоря, невозрастания) весов, и применим к полученному графу [алгоритм Куна](#).

Утверждается, что полученное при этом максимальное (с точки зрения количества рёбер) паросочетание будет и оптимальным с точки зрения суммы весов насыщенных вершин (несмотря на то, что после сортировки мы фактически больше не используем эти веса).

Таким образом, реализация будет примерно такой:

```
int n;
vector < vector<int> > g (n);
vector used (n);
vector<int> order (n); // список вершин, отсортированный по весу
... чтение ...

for (int i=0; i<n; ++i) {
    int v = order[i];
    used.assign (n, false);
    try_kuhn (v);
}
```

Функция `try_kuhn()` берётся безо всяких изменений из алгоритма Куна.

Доказательство

Напомним основные положения теории матроидов.

Матроид M - это упорядоченная пара (S, I) , где S - некоторое множество, I - непустое семейство подмножеств множества S , которые удовлетворяют следующим условиям:

1. Множество S конечное.
2. Семейство I является наследственным, т.е. если какое-то множество принадлежит I , то все его подмножества также принадлежат I .
3. Структура M обладает свойством замены, т.е. если $A \in I$, и $B \in I$, и $|A| < |B|$, то найдётся такой элемент $x \in A - B$, что $A \cup x \in I$.

Элементы семейства I называются независимыми подмножествами.

Матроид называется взвешенным, если для каждого элемента $x \in S$ определён некоторый вес. Весом подмножества называется сумма весов его элементов.

Наконец, важнейшая теорема в теории взвешенных матроидов: чтобы получить оптимальный ответ, т.е. независимое подмножество с наибольшим весом, нужно действовать жадно: начиная с пустого подмножества, будем добавлять (если, конечно, текущий элемент можно добавить без нарушения независимости) все элементы по одному в порядке уменьшения (точнее, невозрастания) их весов:

```
отсортировать множество S по невозрастанию веса;
ans = [];
foreach (x in S)
    if (ans ∪ x ∈ I)
        ans = ans ∪ x;
```

Утверждается, что по окончании этого процесса мы получим подмножество с наибольшим весом.

Теперь докажем, что наша задача - не что иное, как взвешенный матроид.

Пусть S - множество всех вершин первой доли. Чтобы свести нашу задачу к двудольному графу к матроиду относительно вершин первой доли, поставим в соответствие каждому паросочетанию такое подмножество S , которое равно множеству насыщенных вершин первой доли. Можно также определить и обратное соответствие (из множества насыщенных вершин - в паросочетание), которое, хотя и не будет однозначным, однако вполне нас будет устраивать.

Тогда определим семейство I как семейство таких подмножеств множества S , для которых найдётся хотя бы одно соответствующее паросочетание.

Далее, для каждого элемента S , т.е. для каждой вершины первой доли, по условию определён некоторый вес. Причём вес подмножества, как нам и требуется в рамках теории матроидов, определяется как сумма весов элементов в нём.

Тогда задача о нахождении паросочетания наибольшего веса теперь переформулируется как задача нахождения независимого подмножества наибольшего веса.

Осталось проверить, что выполнены 3 вышеописанных условия, наложенных на матроид. Во-первых, очевидно, что S является конечным. Во-вторых, очевидно, что удаление ребра из паросочетания эквивалентно удалению вершины из множества насыщенных вершин, а потому свойство наследственности выполняется. В-третьих, как следует из корректности алгоритма Куна, если текущее паросочетание не максимально, то всегда найдётся такая вершина, которую можно будет насытить, не удаляя из множества насыщенных вершин другие вершины.

Итак, мы показали, что наша задача является взвешенным матроидом относительно множества насыщенных вершин первой доли, а потому к ней применим жадный алгоритм.

Осталось показать, что алгоритм Куна является этим жадным алгоритмом.

Однако это довольно очевидный факт. Алгоритм Куна на каждом шаге пытается насытить текущую вершину - либо просто проводя ребро в ненасыщенную вершину второй доли, либо находя удлиняющую цепь и чередуя паросочетание вдоль неё. И в том, и в другом случае никакие уже насыщенные вершины не перестают быть ненасыщенными, а ненасыщенные на предыдущих шагах вершины первой доли не насыщаются и на этом шаге. Таким образом, алгоритм Куна является жадным алгоритмом, строящим оптимальное независимое подмножество матроида, что и завершает наше доказательство.

Алгоритм Эдмондса нахождения наибольшего паросочетания в произвольных графах

Дан неориентированный невзвешенный граф G с N вершинами. Требуется найти в нём наибольшее паросочетание, т.е. такое наибольшее (по мощности) множество M его рёбер, что никакие два ребра из M не имеют общих вершин.

В отличие от случая двудольного графа (см. [Алгоритм Куна](#)), в графе G могут присутствовать циклы нечётной длины, что значительно усложняет поиск увеличивающих путей.

Приведём сначала теорему Бержа, из которой следует, что, как и в случае двудольных графов, наибольшее паросочетание можно находить при помощи увеличивающих путей.

Увеличивающие пути. Теорема Бержа

Пусть зафиксировано некоторое паросочетание M . Тогда простая цепь $P = (v_1, v_2, \dots, v_k)$ называется **чредующейся** цепью, если в ней рёбра по очереди принадлежат паросочетанию M . Чредующаяся цепь называется **увеличивающейся**, если её первая и последняя вершины не принадлежат паросочетанию. Иными словами, простая цепь P является увеличивающейся тогда и только тогда, когда вершина $v_1 \notin M$, ребро $(v_2, v_3) \in M$, ребро $(v_4, v_5) \in M$, ..., ребро $(v_{k-2}, v_{k-1}) \in M$, и вершина $v_k \notin M$.

Теорема Бержа (Berge, 1957 г.). Паросочетание M является наибольшим тогда и только тогда, когда для него не существует увеличивающейся цепи.

Доказательство необходимости. Пусть для паросочетания M существует увеличивающаяся цепь P . Покажем, как перейти к паросочетанию большей мощности. Выполним чередование паросочетания M вдоль этой цепи P , т.е. включим в паросочетание рёбра $(v_1, v_2), (v_3, v_4), \dots, (v_{k-1}, v_k)$, и удалим из паросочетания рёбра $(v_2, v_3), (v_4, v_5), \dots, (v_{k-2}, v_{k-1})$. В результате, очевидно, будет получено корректное паросочетание, мощность которого будет на единицу выше, чем у паросочетания M (т.к. мы добавили $k/2$ рёбер, а удалили $k/2 - 1$ ребро).

Доказательство достаточности. Пусть для паросочетания M не существует увеличивающейся цепи, докажем, что оно является наибольшим. Пусть \overline{M} — наибольшее паросочетание. Рассмотрим симметрическую разность $\overline{G} = M \oplus \overline{M}$ (т.е. множество рёбер, принадлежащих либо M , либо \overline{M} , но не обоим одновременно). Покажем, что \overline{G} содержит одинаковое число рёбер из M и \overline{M} (т.к. мы исключили из \overline{G} только общие для них рёбра, то отсюда будет следовать $|M| = |\overline{M}|$). Заметим, что \overline{G} состоит только из простых цепей и циклов (т.к. иначе одной вершине были бы инцидентны сразу два ребра какого-либо паросочетания, что невозможно). Далее, циклы не могут иметь нечётную длину (по той же самой причине). Цепь в \overline{G} также не может иметь нечётную длину (иначе бы она являлась увеличивающейся цепью для M , что противоречит условию, или для \overline{M} , что противоречит его максимальности). Наконец, в чётных циклах и цепях чётной длины в \overline{G} рёбра поочерёдно входят в M и \overline{M} , что означает, что в \overline{G} входит одинаковое количество рёбер от M и \overline{M} . Как уже упоминалось выше, отсюда следует, что $|M| = |\overline{M}|$, т.е. M является наибольшим паросочетанием.

Теорема Бержа даёт основу для алгоритма Эдмондса — поиск увеличивающих цепей и чередование вдоль них, пока увеличивающие цепи находятся.

Алгоритм Эдмондса. Сжатие цветков

Основная проблема заключается в том, как находить увеличивающий путь. Если в графе имеются циклы нечётной длины, то просто запускать обход в глубину/ширину нельзя.

Можно привести простой контрпример, когда при запуске из одной из вершин алгоритм, не обрабатывающий особо циклы нечётной длины (фактически, **Алгоритм Куна**) не найдёт увеличивающий путь, хотя должен. Это цикл длины 3 с висящим на нём ребром, т.е. граф 1-2, 2-3, 3-1, 2-4, и ребро 2-3 взято в паросочетание. Тогда при запуске из вершины 1, если обход пойдёт сначала в вершину 2, то он "упрётся" в вершину 3, вместо того чтобы найти увеличивающую цепь 1-3-2-4. Правда, на этом примере при запуске из вершины 4 алгоритм Куна всё же найдёт эту увеличивающую цепь.

Тем не менее, можно построить граф, на котором при определённом порядке в списках смежности алгоритм Куна зайдёт в тупик. В качестве примера можно привести такой граф с 6 вершинами и 7 рёбрами: 1-2, 1-6, 2-6, 2-4, 4-3, 1-5, 4-5. Если применить здесь алгоритм Куна, то он найдёт паросочетание 1-6, 2-4, после чего он должен будет обнаружить увеличивающую цепь 5-1-6-2-4-3, однако может так и не обнаружить её (если из вершины 5 он пойдёт сначала в 4, и только потом в 1, а при запуске из вершины 3 он из вершины 2 пойдёт сначала в 1, и только затем в 6).

Как мы увидели на этом примере, вся проблема в том, что при попадании в цикл нечётной длины обход может пойти по циклу в неправильном направлении. На самом деле, нас интересуют только "насыщенные" циклы, т.е. в которых имеется k насыщенных рёбер, где длина цикла равна $2k + 1$. В таком цикле есть ровно одна вершина, не насыщённая рёбрами этого цикла, назовём её **базой** (base). К базовой вершине подходит чредующийся путь чётной (возможно, нулевой) длины, начинающийся в свободной (т.е. не принадлежащей паросочетанию) вершине, и этот путь называется **стеблем** (stem). Наконец, подграф, образованный "насыщенным" нечётным циклом, называется **цветком** (blossom).

Идея алгоритма Эдмондса (Edmonds, 1965 г.) — в **сжатии цветков** (blossom shrinking). Сжатие цветка — это сжатие всего нечётного цикла в одну псевдо-вершину (соответственно, все рёбра, инцидентные вершинам этого цикла, становятся инцидентными псевдо-вершине).

Алгоритм Эдмондса ищёт в графе все цветки, сжимает их, после чего в графе не остаётся "плохих" циклов нечётной длины, и на таком графе (называемом "поверхностным" (surface) графом) уже можно искать увеличивающую цепь простым обходом в глубину/ширину. После нахождения увеличивающей цепи в поверхности графе необходимо "развернуть" цветки, восстановив тем самым увеличивающую цепь в исходном графе.

Однако неочевидно, что после сжатия цветка не нарушится структура графа, а именно, что если в графе G существовала увеличивающая цепь, то она существует и в графе \overline{G} , полученном после сжатия цветка, и наоборот.

Теорема Эдмондса. В графе \overline{G} существует увеличивающаяся цепь тогда и только тогда, когда существует увеличивающаяся цепь в G .

Доказательство. Итак, пусть граф \overline{G} был получен из графа G сжатием одного цветка (обозначим через B цикл цветка, и через \overline{B} соответствующую сжатую вершину), докажем утверждение теоремы. Вначале заметим, что достаточно рассматривать случай, когда база цветка является свободной вершиной (не принадлежащей паросочетанию). Действительно, в противном случае в базе цветка оканчивается чредующийся путь чётной длины, начинающийся в свободной вершине. Прочередовав паросочетание вдоль этого пути, мощность паросочетания не изменится, а база цветка станет свободной вершиной. Итак, при доказательстве можно считать, что база цветка является свободной вершиной.

Доказательство необходимости. Пусть путь P является увеличивающим в графе G . Если он не проходит через B , то тогда, очевидно, он будет увеличивающим и в графе \overline{G} . Пусть P проходит через B . Тогда можно не теряя общности считать, что путь P представляет собой некоторый путь P_1 , не проходящий по вершинам B , плюс некоторый путь P_2 , проходящий по вершинам B и, возможно, другим вершинам. Но тогда путь $P_1 + \overline{B}$ будет являться увеличивающим путём в графе \overline{G} , что и требовалось доказать.

Доказательство достаточности. Пусть путь \overline{P} является увеличивающим путём в графе \overline{G} . Снова, если путь \overline{P} не проходит через \overline{B} , то путь \overline{P} без изменений является увеличивающим путём в G , поэтому этот случай мы рассматривать не будем.

Рассмотрим отдельно случай, когда \overline{P} начинается со сжатого цветка \overline{B} , т.е. имеет вид (\overline{B}, c, \dots) . Тогда в цветке B найдётся соответствующая вершина v , которая связана (ненасыщенным) ребром с c . Осталось только заметить, что из базы цветка всегда найдётся чредующийся путь чётной длины до вершины v . Учитывая всё вышесказанное, получаем, что путь $P = (b, \dots, v, c, \dots)$ является увеличивающим путём в графе G .

Пусть теперь путь \overline{P} проходит через псевдо-вершину \overline{B} , но не начинается и не заканчивается в ней. Тогда в \overline{P} есть два ребра, проходящих

через \overline{B} , пусть это (a, \overline{B}) и (\overline{B}, c) . Одно из них обязательно должно принадлежать паросочетанию M , однако, т.к. база цветка не насыщена, а все остальные вершины цикла цветка B насыщены рёбрами цикла, то мы приходим к противоречию. Таким образом, этот случай просто невозможен.

Итак, мы рассмотрели все случаи и в каждом из них показали справедливость теоремы Эдмондса.

Общая схема алгоритма Эдмондса принимает следующий вид:

```
void edmonds() {
    for (int i=0; i<n; ++i)
        if (вершина i не в паросочетании) {
            int last_v = find_augment_path (i);
            if (last_v != -1)
                выполнить чередование вдоль пути из i в last_v;
        }
}

int find_augment_path (int root) {
    обход в ширину:
    int v = текущая_вершина;
    перебрать все рёбра из v
    если обнаружили цикл нечётной длины, сжать его
    если пришли в свободную вершину, return
    если пришли в несвободную вершину, то добавить
        в очередь смежную ей в паросочетании
    return -1;
}
```

Эффективная реализация

Сразу оценим асимптотику. Всего имеется N итераций, на каждой из которых выполняется обход в ширину за $O(M)$, кроме того, могут происходить операции сжатия цветков — их может быть $O(N)$. Таким образом, если мы научимся сжимать цветок за $O(N)$, то общая асимптотика алгоритма составит $O(N(M + N^2)) = O(N^3)$.

Основную сложность представляют операции сжатия цветков. Если выполнять их, непосредственно объединяя списки смежности в один и удаляя из графа лишние вершины, то асимптотика сжатия одного цветка будет $O(M)$, кроме того, возникнут сложности при "разворачивании" цветков.

Вместо этого будем для каждой вершины графа G поддерживать указатель на базу цветка, которому она принадлежит (или на себя, если вершина не принадлежит никакому цветку). Нам надо решить две задачи: сжатие цветка за $O(N)$ при его обнаружении, а также удобное сохранение всей информации для последующего чередования вдоль увеличивающего пути.

Итак, одна итерация алгоритма Эдмондса представляет собой обход в ширину, выполняемый из заданной свободной вершины $root$. Постепенно будет строиться дерево обхода в ширину, причём путь в нём до любой вершины будет являться чередующимся путём, начинающимся со свободной вершины $root$. Для удобства программирования будем кладь в очередь только те вершины, расстояние до которых в дереве путей чётно (будем называть такие вершины чётными — т.е. это корень дерева, и вторые концы рёбер в паросочетании). Само дерево будем хранить в виде массива предков $P[]$, в котором для каждой нечётной вершины (т.е. до которой расстояние в дереве путей нечётно, т.е. это первые концы рёбер в паросочетании) будем хранить предка — чётную вершину. Таким образом, для восстановления пути из дерева нам надо поочерёдно пользоваться массивами $P[]$ и $MATCH[]$, где $MATCH[]$ — для каждой вершины содержит смежную ей в паросочетании, или -1, если таковой нет.

Теперь становится понятно, как обнаруживать циклы нечётной длины. Если мы из текущей вершины v в процессе обхода в ширину приходим в такую вершину u , являющуюся корнем $root$ или принадлежащую паросочетанию и дереву путей (т.е. $P[MATCH[]]$ от которой не равно -1), то мы обнаружили цветок. Действительно, при выполнении этих условий и вершина v , и вершина u являются чётными вершинами. Расстояние от них до их наименьшего общего предка имеет одну чётность, поэтому найденный нами цикл имеет нечётную длину.

Научимся сжимать цикл. Итак, мы обнаружили нечётный цикл при рассмотрении ребра (v, u) , где u и v — чётные вершины. Найдём их наименьшего общего предка b , он и будет базой цветка. Нетрудно заметить, что база тоже является чётной вершиной (поскольку у нечётных вершин в дереве путей есть только один сын). Однако надо заметить, что b — это, возможно, псевдовершина, поэтому мы фактически найдём базу цветка, являющейся наименьшим общим предком вершин v и u . Реализуем сразу нахождение наименьшего общего предка (нас вполне устраивает асимптотика $O(N)$):

```
int lca (int a, int b) {
    bool used[MAXN] = { 0 };
    // поднимаемся от вершины a до корня, помечая все чётные вершины
    for (;;) {
        a = base[a];
        used[a] = true;
        if (match[a] == -1) break; // дошли до корня
        a = p[match[a]];
    }
    // поднимаемся от вершины b, пока не найдём помеченную вершину
    for (;;) {
        b = base[b];
        if (used[b]) return b;
        b = p[match[b]];
    }
}
```

Теперь нам надо выявить сам цикл — пройтись от вершин v и u до базы b цветка. Будет более удобно, если мы пока просто пометим в каком-то специальном массиве (назовём его $BLOSSOM[]$) вершины, принадлежащие текущему цветку. После этого нам надо будет симитировать обход в ширину из псевдо-вершины — для этого достаточно положить в очередь обхода в ширину все вершины, лежащие на цикле цветка. Тем самым мы избежим явного объединения списков смежности.

Однако остаётся ещё одна проблема: корректное восстановление путей по окончании обхода в ширину. Для него мы сохранили массив предков P . Но после сжатия цветков возникает единственная проблема: обход в ширину продолжился сразу из всех вершин цикла, в том числе и нечётных, а массив предков у нас предназначался для восстановления путей по чётным вершинам. Более того, когда в сжатом графе найдётся увеличивающая цепь, проходящая через цветок, она вообще будет проходить по этому циклу в таком направлении, что в дереве путей это будет представляться движением вниз. Однако все эти проблемы изящно решаются таким манёвром: при сжатии цикла, проставим предков для всех его чётных вершин (кроме базы), чтобы эти "предки" указывали на соседнюю вершину в цикле. Для вершин u и v , если они также не базы, направим указатели предков друг на друга. В результате, если при восстановлении увеличивающего пути мы придём в цикл цветка в нечётную вершину, путь по предкам будет восстановлен корректно, и приведёт в базу цветка (из которой он уже дальше будет восстанавливаться нормально).

Итак, мы готовы реализовать сжатие цветка:

```
int v, u; // ребро (v, u), при рассмотрении которого был обнаружен цветок
int b = lca(v, u);
memset(blossom, 0, sizeof blossom);
mark_path(v, b, u);
mark_path(u, b, v);
```

где функция `mark_path` проходит по пути от вершины до базы цветка, проставляет в специальном массиве $BLOSSOM$ для них `true` и проставляет предков для чётных вершин. Параметр `children` — сын для самой вершины v (с помощью этого параметра мы замкнём цикл в предках).

```
void mark_path(int v, int b, int children) {
    while (base[v] != b) {
        blossom[base[v]] = blossom[base[match[v]]] = true;
        p[v] = children;
        children = match[v];
        v = p[match[v]];
    }
}
```

Наконец, реализуем основную функцию — `find_path` (`int root`), которая будет искать увеличивающий путь из свободной вершины $root$ и возвращать последнюю вершину этого пути, или -1 , если увеличивающий путь не найден.

Вначале произведём инициализацию:

```
int find_path(int root) {
    memset(used, 0, sizeof used);
    memset(p, -1, sizeof p);
    for (int i=0; i<n; ++i)
        base[i] = i;
```

Далее идёт обход в ширину. Рассматривая очередное ребро (v, to) , у нас есть несколько вариантов:

- Ребро несуществующее. Под этим мы подразумеваем, что v и to принадлежат одной сжатой псевдо-вершине ($\text{BASE}[v] == \text{BASE}[to]$), поэтому в текущем поверхностном графе этого ребра нет. Кроме этого случая, есть ещё один случай: когда ребро (v, to) уже принадлежит текущему паросочетанию; т.к. мы считаем, что вершина v является чётной вершиной, то проход по этому ребру означает в дереве путей подъём к предку вершины v , что недопустимо.

```
if (base[v] == base[to] || match[v] == to) continue;
```

- Ребро замыкает цикл нечётной длины, т.е. обнаруживается цветок. Как уже упоминалось выше, цикл нечётной длины обнаруживается при выполнении условия:

```
if (to == root || match[to] != -1 && p[match[to]] != -1)
```

В этом случае нужно выполнить сжатие цветка. Выше уже подробно разбирался этот процесс, здесь приведём его реализацию:

```
int curbase = lca(v, to);
memset(blossom, 0, sizeof blossom);
mark_path(v, curbase, to);
mark_path(to, curbase, v);
for (int i=0; i<n; ++i)
    if (blossom[base[i]]) {
        base[i] = curbase;
        if (!used[i]) {
            used[i] = true;
            q[qt++] = i;
        }
    }
}
```

- Иначе — это "обычное" ребро, поступаем как и в обычном поиске в ширину. Единственная тонкость — при проверке, что эту вершину мы ещё не посещали, надо смотреть не в массив `used`, а в массив P — именно он заполняется для посещённых нечётных вершин. Если мы в вершину to ещё не заходили, и она оказалась ненасыщенной, то мы нашли увеличивающую цепь, заканчивающуюся на вершине to , возвращаем её.

```
if (p[to] == -1) {
    p[to] = v;
    if (match[to] == -1)
        return to;
    to = match[to];
    used[to] = true;
```

```
    q[qt++ ] = to;
}
```

Итак, полная реализация функции `find_path()`:

```
int find_path (int root) {
    memset (used, 0, sizeof used);
    memset (p, -1, sizeof p);
    for (int i=0; i<n; ++i)
        base[i] = i;

    used[root] = true;
    int qh=0, qt=0;
    q[qt++ ] = root;
    while (qh < qt) {
        int v = q[qh++];
        for (size_t i=0; i<g[v].size(); ++i) {
            int to = g[v][i];
            if (base[v] == base[to] || match[v] == to) continue;
            if (to == root || match[to] != -1 && p[match[to]] != -1) {
                int curbase = lca (v, to);
                memset (blossom, 0, sizeof blossom);
                mark_path (v, curbase, to);
                mark_path (to, curbase, v);
                for (int i=0; i<n; ++i)
                    if (blossom[base[i]]) {
                        base[i] = curbase;
                        if (!used[i]) {
                            used[i] = true;
                            q[qt++ ] = i;
                        }
                    }
            } else if (p[to] == -1) {
                p[to] = v;
                if (match[to] == -1)
                    return to;
                to = match[to];
                used[to] = true;
                q[qt++ ] = to;
            }
        }
    }
    return -1;
}
```

Наконец, приведём определения всех глобальных массивов, и реализацию основной программы нахождения наибольшего паросочетания:

```
const int MAXN = ...; // максимально возможное число вершин во входном графе

int n;
vector<int> g[MAXN];
int match[MAXN], p[MAXN], base[MAXN], q[MAXN];
bool used[MAXN], blossom[MAXN];

...

int main() {
    ... чтение графа ...

    memset (match, -1, sizeof match);
    for (int i=0; i<n; ++i)
        if (match[i] == -1) {
            int v = find_path (i);
            while (v != -1) {
                int pv = p[v], ppv = match[pv];
                match[v] = pv, match[pv] = v;
                v = ppv;
            }
        }
}
```

Оптимизация: предварительное построение паросочетания

Как и в случае [Алгоритма Куна](#), перед выполнением алгоритма Эдмондса можно каким-нибудь простым алгоритмом построить предварительное паросочетание. Например, таким жадным алгоритмом:

```
for (int i=0; i<n; ++i)
    if (match[i] == -1)
        for (size_t j=0; j<g[i].size(); ++j)
            if (match[g[i][j]] == -1) {
                match[g[i][j]] = i;
                match[i] = g[i][j];
```

```
    break;  
}
```

Такая оптимизация значительно (в несколько раз) ускорит работу алгоритма на случайных графах.

Случай двудольного графа

В двудольных графах отсутствуют циклы нечётной длины, и, следовательно, код, выполняющий сжатие цветков, никогда не выполнится. Удалив мысленно все части кода, обрабатывающие сжатие цветков, мы получим [Алгоритм Куна](#) практически в чистом виде. Таким образом, на двудольных графах алгоритм Эдмондса вырождается в [алгоритм Куна](#) и работает за $O(NM)$.

Дальнейшая оптимизация

Во всех вышеописанных операциях с цветками легко угадываются операции с непересекающимися множествами, которые можно выполнять намного эффективнее (см. [Система непересекающихся множеств](#)). Если переписать алгоритм с использованием этой структуры, то асимптотика алгоритма понизится до $O(NM)$. Таким образом, для произвольных графов мы получили ту же асимптотическую оценку, что и в случае двудольных графов (алгоритм Куна), но заметно более сложным алгоритмом.

Покрытие путями ориентированного ациклического графа

Дан ориентированный ациклический граф G . Требуется покрыть его наименьшим числом путей, т.е. найти наименьшее по мощности множество непересекающихся по вершинам простых путей, таких, что каждая вершина принадлежит какому-либо пути.

Сведение к двудольному графу

Пусть дан граф G с n вершинами. Построим соответствующий ему двудольный граф H стандартным образом, т.е.: в каждой доле графа H будет по n вершин, обозначим их через a_i и b_i соответственно. Тогда для каждого ребра (i, j) исходного графа G проведём соответствующее ребро (a_i, b_j) .

Каждому ребру G соответствует одно ребро H , и наоборот. Если мы рассмотрим в G любой путь $P = (v_1, v_2, \dots, v_k)$, то ему ставится в соответствие набор рёбер $(a_{v_1}, b_{v_2}), (a_{v_2}, b_{v_3}), \dots, (a_{v_{k-1}}, b_{v_k})$.

Более просто для понимания будет, если мы добавим "обратные" рёбра, т.е. образуем граф \overline{H} из графа H добавлением рёбер вида $(b_i, a_i), i = 1 \dots N$. Тогда пути $P = (v_1, v_2, \dots, v_k)$ в графе \overline{H} будет соответствовать путь $\overline{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, \dots, a_{v_{k-1}}, b_{v_k})$.

Обратно, рассмотрим любой путь \overline{Q} в графе \overline{H} , начинающийся в первой доле и заканчивающийся во второй доле. Очевидно, \overline{Q} снова будет иметь вид $\overline{Q} = (a_{v_1}, b_{v_2}, a_{v_2}, b_{v_3}, \dots, a_{v_{k-1}}, b_{v_k})$, и ему можно поставить в соответствие в графе G путь $P = (v_1, v_2, \dots, v_k)$. Однако здесь есть одна тонкость: v_1 могло совпадать с v_k , поэтому путь P получился бы циклом. Однако по условию граф G ациклический, поэтому это вообще невозможно (это единственное место, где используется ациклическость графа G ; тем не менее, на циклические графы описываемый здесь метод вообще нельзя обобщить).

Итак, всякому простому пути в графе \overline{H} , начинающемуся в первой доле и заканчивающемуся во второй, можно поставить в соответствие простой путь в графе G , и наоборот. Но заметим, что такой путь в графе \overline{H} — это **паросочетание** в графе H . Таким образом, любому пути из G можно поставить в соответствие паросочетание в графе H , и наоборот. Более того, непересекающимся путям в G соответствуют непересекающиеся паросочетания в H .

Последний шаг. Заметим, что чем больше путей есть в нашем наборе, тем меньше все эти пути содержат рёбер. А именно, если есть p непересекающихся путей, покрывающих все n вершин графа, то они вместе содержат $r = n - p + 1$ рёбер. Итак, чтобы минимизировать число путей, мы должны **максимизировать число рёбер** в них.

Итак, мы свели задачу к нахождению максимального паросочетания в двудольном графе H . После нахождения этого паросочетания (см. [Алгоритм Куна](#)) мы должны преобразовать его в набор путей в G (это делается тривиальным алгоритмом, неоднозначностей здесь не возникает). Некоторые вершины могут остаться ненасыщенным паросочетанием, в таком случае в ответ надо добавить пути нулевой длины из каждой из этих вершин.

Взвешенный случай

Взвешенный случай не сильно отличается от невзвешенного, просто в графе H на рёбрах появляются веса, и требуется найти уже паросочетание наименьшего веса. Восстанавливая ответ аналогично невзвешенному случаю, мы получим покрытие графа наименьшим числом путей, а при равенстве — наименьшим по стоимости.

Рёберная связность. Свойства и нахождение

Определение

Пусть дан граф G.

Рёберной связностью λ графа G называется наименьшее число рёбер, которое нужно удалить, чтобы граф перестал быть связным. Для несвязного графа рёберная связность равна нулю. Для графа с мостом рёберная связность равна единице.

Множество S рёбер **разделяет** вершины A и B, если при удалении этих рёбер из графа вершины A и B оказываются в разных компонентах связности.

Ясно, что рёберная связность графа равна минимуму наименьшего числа рёбер, разделяющих две вершины A и B, взятому среди всех пар (A,B).

Свойства

Соотношение Уитни (Whitney) (1932 г.) между рёберной связностью λ , вершинной связностью k и наименьшей из степеней вершин δ :

$$k \leq \lambda \leq \delta$$

Теорема Форда-Фалкерсона (1956 г.). Для любых двух вершин наибольшее число рёберно-непересекающихся цепей, соединяющих их, равно наименьшему числу рёбер, разделяющих эти вершины.

На этой теореме и основано **нахождение** величины рёберной связности.

Нахождение рёберной связности

Итак, мы должны перебрать все пары вершин (A,B), и между каждой парой найти наибольшее число непересекающихся по рёбрам путей. А найти эту величину уже можно с помощью алгоритма максимального потока: мы делаем A истоком, B стоком, а пропускную способность каждого ребра положив равной 1.

Таким образом, псевдокод алгоритма таков:

```
int ans = INF;
for (int s=0; s<n; ++s)
    for (int t=0; t<n; ++t) {
        int flow;
        ... находим величину flow максимального потока из s в t ...
        ans = min (ans, flow);
    }
```

Асимптотика алгоритма при использовании [алгоритма Эдмондса-Карпа](#) нахождения максимального потока получается $O(N^2 NM^2) = O(N^3 M^2)$, однако следует заметить, что скрытая в асимптотике константа весьма мала, поскольку практически невозможно создать такой граф, чтобы алгоритм нахождения максимального потока работал медленно сразу при всех стоках и истоках.

Вершинная связность. Свойства и нахождение

Определение

Пусть дан граф G.

Вершинной связностью к ("каппа") графа G называется наименьшее число вершин, которое нужно удалить, чтобы граф перестал быть связным. Для несвязного графа вершинная связность равна нулю. Для полносвязного графа вершинная связность полагается равной $N-1$ (поскольку, сколько вершин ни удалять из графа, - он всё равно не перестанет быть связным).

Множество S вершин **разделяет** вершины A и B, если при удалении этих вершин из графа вершины A и B оказываются в разных компонентах связности.

Ясно, что для всех графов, кроме полного, вершинная связность графа равна минимуму из всех наименьших чисел вершин, разделяющих две вершины A и B, взятому по всевозможным парам (A,B). Кроме того, можно заметить, что пары смежных вершин можно не рассматривать - они всё равно не смогут изменить ответ.

Свойства

Соотношение Уитни (Whitney) (1932 г.) между рёберной связностью λ , вершинной связностью k и наименьшей из степеней вершин δ :

$$k \leq \lambda \leq \delta$$

Теорема Менгера (1927 г.). Для любых двух вершин наибольшее число вершинно-непересекающихся простых цепей, соединяющих их, равно наименьшему числу вершин, разделяющих эти вершины.

На этой теореме и основано **нахождение** величины вершинной связности.

Нахождение вершинной связности

Итак, изначально мы полагаем ответ равным $N-1$. Затем мы перебираем всевозможные пары несмежных вершин (A, B), и между каждой парой найти наибольшее число непересекающихся по вершинам простых путей. А найти эту величину уже можно с помощью алгоритма максимального потока: мы делаем A истоком, B стоком, все остальные вершины разделяем (вершина I заменяется I^1 и I^2), причём все рёбра, ранее входившие в I , направляем в I^1 , а все рёбра, выходившие из I , направляем из I^2 , кроме того, проводим ребро из I^1 в I^2 ; пропускную способность каждого ребра положив равной 1.

Таким образом, псевдокод алгоритма таков:

```
int ans = n-1;
строим модифицированный граф с раздвоенными вершинами
for (int s=0; s<n; ++s)
    for (int t=0; t<n; ++t) {
        if (g[s][t]) continue;
        int flow;
        ... находим величину flow максимального потока из s2 в t1 ...
        ans = min (ans, flow);
    }
```

Асимптотика алгоритма при использовании [алгоритма Эдмондса-Карпа](#) нахождения максимального потока получается $O(N^2 NM^2) = O(N^3 M^2)$, однако следует заметить, что скрытая в асимптотике константа весьма мала, поскольку практически невозможно создать такой граф, чтобы алгоритм нахождения максимального потока работал медленно сразу при всех стоках и истоках.

Построение графа с указанными величинами вершинной и рёберной связностей и наименьшей из степеней вершин

Даны величины κ , λ , δ - вершинная связность ("каппа"), рёберная связность ("лямбда") и наименьшая из степеней вершин графа ("дельта"). Требуется построить граф, который бы обладал указанными значениями, или сказать, что такого графа не существует.

(см. "[вершинная связность](#)", "[рёберная связность](#)")

Решение

Заметим, что всегда выполняется условие:

```
 $\kappa \leq \lambda \leq \delta$ 
```

Это соотношение было найдено Уитни (Whitney) в 1932 году (впрочем, эти соотношения можно легко доказать эвристически)

Менее очевидный факт - что если κ , λ и δ удовлетворяют этому соотношению, то граф, обладающий этими значениями, обязательно существует. Таким образом, мы решили первую часть задачи - ответили на вопрос о существовании ответа.

Теперь построим сам график. Граф будет состоять из $2(\delta+1)$ вершин, причём первые $(\delta+1)$ вершины образуют полно связный подграф, и вторые $(\delta+1)$ вершины также образуют полно связный подграф. Кроме того, соединим эти две части λ рёбрами так, чтобы в первой части эти рёбра были смежны λ вершинам, а в другой части - к вершинам. Очевидно, полученный график будет обладать необходимыми свойствами.

Нахождение К-го кратчайшего пути без циклов с помощью бинарного поиска

Дан взвешенный график (ориентированный или неориентированный) с N вершинами, M рёбрами, любой путь в котором имеет длину не более W , а каждое ребро имеет неотрицательный вес. Кратные рёбра допускаются. Также дано число K , и указаны две вершины S и T . Требуется найти K -ый в порядке сортировки простой путь (т.е. без повторяющихся вершин), соединяющий вершины S и T .

Наиболее популярный алгоритм решения этой задачи - это алгоритм Йена, однако его асимптотика - $O(NMK)$. Здесь же мы рассмотрим алгоритм, который и идеально значительно проще, и асимптотически быстрее (в большинстве случаев) - за $O(N^2K \log W)$.

Алгоритм

Научимся для заданной верхней границы MaxLen проверять за $O(N^2K)$, есть ли K простых путей из S в T длины не более MaxLen. Однако это можно сделать достаточно просто - фактически с помощью перебора с отсечениями. Запустим перебор из вершины S , который будет искать всевозможные простые пути в вершину T длины не более MaxLen. Ясно, что чтобы перебор был эффективным, если текущая длина пути уже больше MaxLen, то из этой ветви перебора надо сразу выходить (здесь используется то, что рёбер с отрицательными весами нет). Если в какой-то момент мы найдём K путей, то мы уже можем останавливать перебор - ответ найден, как минимум K различных простых путей существуют. Последнее отсечение, которое надо добавить в перебор - чтобы он не заходил в тупики, т.е. в такие вершины, из которых он уже никогда не достигнет T . Математически это отсечение имеет вид: $\text{CurLen} + \text{Dist}(V,T) \leq \text{MaxLen}$, где CurLen - текущая длина пути, $\text{Dist}(A,B)$ - это расстояние между вершинами A и B , которое можно предвычислить алгоритмом [Дейкстры](#) (можно простейшим вариантом за $O(N^2 + M)$, всё равно это асимптотику не ухудшит).

Поймём, почему асимптотика этого перебора получится $O(N^2K)$. Действительно, поскольку все пути простые, то длина каждого найденного пути есть $O(N)$. Далее, поскольку алгоритм не заходит в тупики, то любая его ветвь завершается нахождением ещё одного пути из S в T , но путей будет найдено не более K . Наконец, каждый раз, находясь в какой-либо вершине, перебор выполняет $O(N)$ действий. Итого мы получаем асимптотику $O(N^2K)$.

Теперь применим этот алгоритм перебора к нашей задаче с помощью **бинарного поиска**. Подбирать бинарным поиском мы, очевидно, будем величину MaxLen - ограничение на длину пути. Мы хотим найти минимальную величину MaxLen, при которой в графе ещё есть как минимум K путей из S в T длины не более MaxLen. После окончания бинарного поиска мы можем восстановить и сам путь с помощью всего того же перебора (теперь мы уже знаем длину пути - MaxLen, и хотим найти любой путь этой длины).

Поскольку MaxLen может изменяться только от 0 до W , то бинарный поиск в целом отработает за $O(N^2K \log W)$, восстановление пути (если он, конечно, существует) - за $O(N^2K)$, итого асимптотика $O(N^2K \log W)$.

Реализация

```
const int INF = 1000*1000*1000;
const int W = ...; // максимальная длина пути

int n, s, t;
vector<vector<pair<int,int>>> g;
vector<int> dist;
vector<char> used;
vector<int> curpath, kth_path;

int kth_path_exists (int k, int maxlen, int v, int curlen = 0) {
    curpath.push_back (v);
    if (v == t) {
        if (curlen == maxlen)
            kth_path = curpath;
        curpath.pop_back();
        return 1;
    }
    used[v] = true;
    int found = 0;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i].first, len = g[v][i].second;
        if (!used[to] && curlen + len + dist[to] <= maxlen) {
            found += kth_path_exists (k - found, maxlen, to, curlen + len);
            if (found == k) break;
        }
    }
    used[v] = false;
    curpath.pop_back();
    return found;
}

int main() {
    ... чтение входных данных (n, k, g, s, t) ...

    dist.assign (n, INF);
    dist[t] = 0;
    used.assign (n, false);
    for (;;) {
        int sel = -1;
        for (int i=0; i<n; ++i)
            if (!used[i] && dist[i] < INF && (sel == -1 || dist[i] < dist[sel]))
                sel = i;
        if (sel == -1) break;
        used[sel] = true;
        for (size_t i=0; i<g[sel].size(); ++i) {
            int to = g[sel][i].first, len = g[sel][i].second;
            dist[to] = min (dist[to], dist[sel] + len);
        }
    }

    int minw = 0, maxw = W;
    while (minw < maxw) {
```

```

int wlimit = (minw + maxw) >> 1;
used.assign (n, false);
if (kth_path_exists (k, wlimit, s) == k)
    maxw = wlimit;
else
    minw = wlimit + 1;
}

used.assign (n, false);
if (kth_path_exists (k, minw, s) < k)
    puts ("NO SOLUTION");
else {
    cout << minw << ' ' << kth_path.size() << endl;
    for (size_t i=0; i<kth_path.size(); ++i)
        cout << kth_path[i]+1 << ' ';
}
}

```

Обратная задача SSSP (inverse-SSSP - обратная задача кратчайших путей из одной вершины)

Имеется взвешенный неориентированный мультиграф G из N вершин и M рёбер. Дан массив P[1..N] и указана некоторая начальная вершина S. Требуется изменить веса рёбер так, чтобы для всех $|P|$ было равно длине кратчайшего пути из S в I, причём сумма всех изменений (сумма модулей изменений весов рёбер) была бы наименьшей. Если этого сделать невозможно, то алгоритм должен выдать "No solution". Делать веса ребра отрицательным запрещено.

Описание решения

Мы решим эту задачу за линейное время, просто перебрав все рёбра (т.е. за один проход).

Пусть на текущем шаге мы рассматриваем ребро из вершины A в вершину B длиной R. Мы предполагаем, что для вершины A уже все условия выполнены (т.е. расстояние от S до A действительно равно $P[A]$), и будем проверять выполнение условий для вершины B. Имеем несколько вариантов ситуации:

- 1. $P[A] + R < P[B]$

Это означает, что мы нашли путь, более короткий, чем он должен быть. Поскольку $P[A]$ и $P[B]$ мы изменять не можем, то мы обязаны удлинить текущее ребро (независимо от остальных рёбер), а именно выполнить:

$$R \leftarrow P[B] - P[A] - R.$$

Кроме того, это означает, что мы нашли уже путь в вершину B из S, длина которого равна требуемому значению $P[B]$, поэтому на последующих шагах нам не придётся укорачивать какие-либо рёбра (см. вариант 2).

- 2. $P[A] + R \geq P[B]$

Это означает, что мы нашли путь, более длинный, чем требуемый. Поскольку таких путей может быть несколько, мы должны выбрать среди всех таких путей (рёбер) то, которое потребует наименьшего изменения. Повторимся, что если мы удлиняли какое-то ребро, ведущее в вершину B (вариант 1), то этим мы фактически построили кратчайший путь в вершину B, а потому укорачивать никакое ребро уже не надо будет. Таким образом, для каждой вершины мы должны хранить ребро, которое собираемся укорачивать, т.е. ребро с наименьшим весом изменения.

Таким образом, просто перебрав все рёбра, и рассмотрев для каждого ребра ситуацию (за $O(1)$), мы решим обратную задачу SSSP за линейное время.

Если в какой-то момент мы пытаемся изменить уже изменённое ребро, то, очевидно, этого делать нельзя, и следует выдать "No solution". Кроме того, у некоторых вершин может быть так и не достигнута требуемая оценка кратчайшего пути, тогда ответ тоже будет "No solution". Во всех остальных случаях (кроме, конечно, явно некорректных значений в массиве P, т.е. $P[S] \neq 0$ или отрицательные значения) ответ будет существовать.

Реализация

Программа выводит "No solution", если решения нет, иначе выводит в первой строке минимальную сумму изменений весов рёбер, а в последующих M строках - новые веса рёбер.

```

const int INF = 1000*1000*1000;
int n, m;
vector<int> p (n);

bool ok = true;
vector<int> cost (m), cost_ch (m), decrease (n, INF), decrease_id (n, -1);
decrease[0] = 0;
for (int i=0; i<m; ++i) {
    int a, b, c; // текущее ребро (a,b) с ценой c
    cost[i] = c;
}

```

```

for (int j=0; j<=1; ++j) {
    int diff = p[b] - p[a] - c;
    if (diff > 0) {
        ok &= cost_ch[i] == 0 || cost_ch[i] == diff;
        cost_ch[i] = diff;
        decrease[b] = 0;
    }
    else
        if (-diff <= c && -diff < decrease[b]) {
            decrease[b] = -diff;
            decrease_id[b] = i;
        }
    swap (a, b);
}
}

for (int i=0; i<n; ++i) {
    ok &= decrease[i] != INF;
    int r_id = decrease_id[i];
    if (r_id != -1) {
        ok &= cost_ch[r_id] == 0 || cost_ch[r_id] == -decrease[i];
        cost_ch[r_id] = -decrease[i];
    }
}

if (!ok)
    cout << "No solution";
else {
    long long sum = 0;
    for (int i=0; i<m; ++i) sum += abs (cost_ch[i]);
    cout << sum << '\n';
    for (int i=0; i<m; ++i)
        printf ("%d ", cost[i] + cost_ch[i]);
}

```

Обратная задача MST (inverse-MST - обратная задача минимального остова) за $O(N M^2)$

Дан взвешенный неориентированный граф G с N вершинами и M рёбрами (без петель и кратных рёбер). Известно, что граф связный. Также указан некоторый остов T этого графа (т.е. выбрано $N-1$ ребро, которые образуют дерево с N вершинами). Требуется изменить веса рёбер таким образом, чтобы указанный остов T являлся минимальным остовом этого графа (точнее говоря, одним из минимальных остовов), причём сделать это так, чтобы суммарное изменение всех весов было наименьшим.

Решение

Сведём задачу inverse-MST к задаче min-cost-flow, точнее, к задаче, двойственной min-cost-flow (в смысле двойственности задач линейного программирования); затем решим последнюю задачу.

Итак, пусть дан граф G с N вершинами, M рёбрами. Вес каждого ребра обозначим через C_i . Предположим, не теряя общности, что рёбра с номерами с 1 по $N-1$ являются рёбрами T .

1. Необходимое и достаточное условие MST

Пусть дан некоторый остов S (не обязательно минимальный).

Введём сначала одно обозначение. Рассмотрим некоторое ребро j , не принадлежащее S . Очевидно, в графе S имеется единственный путь, соединяющий концы этого ребра, т.е. единственный путь, соединяющий концы ребра j и состоящий только из рёбер, принадлежащих S .

Обозначим через $P[j]$ множество рёбер, образующих этот путь для j -го ребра.

Для того, чтобы некоторый остов S являлся минимальным, **необходимо и достаточно**, чтобы:

$$C_i \leq C_j \text{ для всех } j \notin S \text{ и каждого } i \in P[j]$$

Можно заметить, что, поскольку в **нашей задаче** остову T принадлежат рёбра $1..N-1$, то мы можем записать это условие таким образом:

$$C_i \leq C_j \text{ для всех } j = N..M \text{ и каждого } i \in P[j]$$

(причём все } i \text{ лежат в диапазоне } 1..N-1)

2. Граф путей

Понятие графа путей непосредственно связано с предыдущей теоремой.

Пусть дан некоторый остов S (не обязательно минимальный).

Тогда **графом путей H** для графа G будет следующий граф:

- Он содержит M вершин, каждая вершина в H взаимно однозначно соответствует некоторому ребру в G .
- Граф H двудольный. В первой его доле находятся вершины i , которые соответствуют рёбрам в G , принадлежащим остову S . Соответственно, во второй доле находятся вершины j , которые соответствуют рёбрам, не принадлежащим S .
- Ребро проводится из вершины i в вершину j тогда и только тогда, когда i принадлежит $P[j]$. Иными словами, для каждой вершины j из второй доли в неё входят рёбра из всех вершин первой доли, соответствующих множеству рёбер $P[j]$.

В случае **нашей задачи** мы можем немного упростить описание графа путей:

```
ребро (i, j) существует в H, если i ∈ P[j], j = N..M, i = 1..N-1
```

3. Математическая формулировка задачи

Чисто формально **задача inverse-MST** записывается таким образом:

```
найти массив A[1..M] такой, что  
Ci + Ai ≤ Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),  
и минимизировать сумму |A1| + |A2| + ... + |Am|
```

здесь под искомым массивом A мы подразумеваем те значения, которые нужно добавить к весам рёбер (т.е., решив задачу inverse-MST, мы заменяем вес C_i каждого ребра i на величину $C_i + A_i$).

Очевидно, что нет смысла увеличивать вес рёбер, принадлежащих T , т.е.

```
Ai <= 0, i = 1..N-1
```

и нет смысла укорачивать рёбра, не принадлежащие T :

```
Ai >= 0, i = N..M
```

(поскольку в противном случае мы только ухудшим ответ)

Тогда мы можем немного **упростить** постановку задачи, убрав из суммы модули:

```
найти массив A[1..M] такой, что  
Ci + Ai ≤ Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),  
Ai ≤ 0, i = 1..N-1,  
Ai ≥ 0, i = N..M,  
и минимизировать сумму An + ... + Am - (A1 + ... + An-1)
```

Наконец, просто изменим "минимизацию" на "максимизацию", а в самой сумме изменим все знаки на противоположные:

```
найти массив A[1..M] такой, что  
Ci + Ai ≤ Cj + Aj для всех j = N..M и каждого i ∈ P[j] (i в 1..N-1),  
Ai ≤ 0, i = 1..N-1,  
Ai ≥ 0, i = N..M,  
и максимизировать сумму A1 + ... + An-1 - (An + ... + Am)
```

4. Сведение задачи inverse-MST к задаче, двойственной задаче о назначениях

Формулировка задачи inverse-MST, которую мы только что дали, является формулировкой задачи **линейного программирования** с неизвестными $A_1..A_m$.

Применим классический приём - рассмотрим **двойственную** ей задачу.

По определению, чтобы получить двойственную задачу, нужно каждому неравенству сопоставить двойственную переменную X_{ij} , поменять ролями целевую функцию (которую нужно было минимизировать) и коэффициенты в правых частях неравенств, поменять знаки " \leq " на " \geq " и наоборот, поменять максимизацию на минимизацию.

Итак, **двойственная** к inverse-MST задача:

```
найти все Xij для каждого (i, j) ∈ H, такие что:  
все Xij ≥ 0,  
для каждого i=1..N-1 ∑ Xij по всем j: (i, j) ∈ H ≤ 1,  
для каждого j=N..M ∑ Xij по всем i: (i, j) ∈ H ≤ 1,  
и минимизировать ∑ Xij (Cj - Ci) для всех (i, j) ∈ H
```

Последняя задача является **задачей о назначениях**: нам нужно в графе путей H выбрать несколько рёбер так, чтобы ни одно ребро не пересекалось с другим в вершине, а сумма весов рёбер (вес ребра (i,j) определим как $C_j - C_i$) должна быть наименьшей.

Таким образом, **двойственная задача inverse-MST эквивалентна задаче о назначениях**. Если мы научимся решать двойственную задачу о назначениях, то мы автоматически решим задачу inverse-MST.

5. Решение двойственной задачи о назначениях

Сначала уделим немного внимания тому частному случаю задачи о назначениях, который мы получили. Во-первых, это несбалансированная задача о назначениях, поскольку в одной доле находится $N-1$ вершин, а в другой - M вершин, т.е. в общем случае число вершин во второй доле больше на целый порядок. Для решения такой двойственной задачи о назначениях есть специализированный алгоритм, который решит её за $O(N^3)$, но здесь этот алгоритм рассматриваться не будет. Во-вторых, такую задачу о назначениях можно назвать задачей о назначениях с взвешенными вершинами: веса рёбер положим равными 0, вес каждой вершины из первой доли положим равным $-C_i$, из второй доли - равным C_j , и решение полученной задачи будет тем же самым.

Мы будем решать задачу двойственную задачу о назначениях с помощью **модифицированного алгоритма min-cost-flow**, который будет находить поток минимальной стоимости и одновременно решение двойственной задачи.

Свести задачу о назначениях к задаче min-cost-flow очень легко, но для полноты картины мы опишем этот процесс.

Добавим в граф исток s и сток t . Из s к каждой вершине первой доли проведём ребро с пропускной способностью = 1 и стоимостью = 0. Из каждой вершины второй доли проведём ребро к t с пропускной способностью = 1 и стоимостью = 0. Пропускные способности всех рёбер между первой и второй долями также положим равными 1.

Наконец, чтобы модифицированный алгоритм min-cost-flow (описанный ниже) работал, нужно **добавить ребро из s в t** с пропускной способностью = $N+1$ и стоимостью = 0.

6. Модифицированный алгоритм min-cost-flow для решения задачи о назначениях

Здесь мы рассмотрим **алгоритм последовательных кратчайших путей с потенциалами**, который напоминает обычный алгоритм min-cost-flow, но использует также понятие **потенциалов**, которые к концу работы алгоритма будут содержать **решение двойственной задачи**.

Введём обозначения. Для каждого ребра (i,j) обозначим через U_{ij} его пропускную способность, через C_{ij} - его стоимость, через F_{ij} - поток вдоль этого ребра.

Также введём понятие потенциалов. Каждая вершина обладает своим потенциалом P_i . Остаточная стоимость ребра CPI_{ij} определяется как:

$$CPI_{ij} = C_{ij} - P_i + P_j$$

В любой момент работы алгоритма **потенциалы таковы**, что выполняются условия:

```
если  $F_{ij} = 0$ , то  $CPI_{ij} \geq 0$ 
если  $F_{ij} = U_{ij}$ , то  $CPI_{ij} \leq 0$ 
иначе  $CPI_{ij} = 0$ 
```

Алгоритм начинает с нулевого потока, и нам нужно найти некоторые начальные значения потенциалов, которые бы удовлетворяли указанным условиям. Нетрудно проверить, что такой способ является одним из возможных решений:

```
Pi_j = 0 для j = N..M
Pi_i = min Ci_j, где (i,j) ∈ H
Pi_s = min Pi_i, где i = 1..N-1
Pi_t = 0
```

Собственно сам алгоритм min-cost-flow состоит из нескольких итераций. **На каждой итерации** мы находим кратчайший путь из s в t в остаточной сети, причём в качестве весов рёбер используем остаточные стоимости CPI. Затем мы увеличиваем поток вдоль найденного пути на единицу, и обновляем потенциалы следующим образом:

```
Pi_i -= Di
```

где D_i - найденное кратчайшее расстояние от s до i (повторимся, в остаточной сети с весами рёбер CPI).

Рано или поздно мы найдём тот путь из s в t , который состоит из единственного ребра (s,t) . Тогда после этой итерации нам следует **завершить** работу алгоритма: действительно, если мы не остановим алгоритм, то дальше уже будут находиться пути с неотрицательной стоимостью, и добавлять их в ответ не надо.

К концу работы алгоритма мы получим решение задачи о назначениях (в виде потока F_{ij}) и решение двойственной задачи о назначениях (в массиве P_i).

(с P_i надо будет провести небольшую модификацию: от всех значений P_i отнять P_s , поскольку его значения имеют смысл только при $P_s = 0$)

6. Итог

Итак, мы решили двойственную задачу о назначениях, а, следовательно, и задачу inverse-MST.

Оценим **асимптотику** получившегося алгоритма.

Сначала мы должны будем построить граф путей. Для этого просто для каждого ребра $j \notin T$ обходом в ширину по оставу T найдём путь $P[j]$. Тогда граф путей мы построим за $O(M) * O(N) = O(NM)$.

Затем мы найдём начальные значения потенциалов за $O(N) * O(M) = O(NM)$.

Затем мы будем выполнять итерации min-cost-flow, всего итераций будет не более N (поскольку из истока выходит N рёбер, каждое с пропускной способностью = 1), на каждой итерации мы ищем в графе путей кратчайшие пути от истока до всех остальных вершин. Поскольку вершин в графе путей равно $M+2$, а число рёбер - $O(NM)$, то, если реализовать поиск кратчайших путей простейшим вариантом алгоритма Дейкстры, каждая итерация min-cost-flow будет выполнять за $O(M^2)$, а весь алгоритм min-cost-flow выполнится за $O(NM^2)$.

Итоговая асимптотика алгоритма равна $O(NM^2)$.

Реализация

Реализуем весь вышеописанный алгоритм. Единственное изменение - вместо алгоритма Дейкстры применяется алгоритм Левита, который на многих тестах должен работать несколько быстрее.

```

const int INF = 1000*1000*1000;

struct rib {
    int v, c, id;
};

struct rib2 {
    int a, b, c;
};

int main() {

    int n, m;
    cin >> n >> m;
    vector < vector<rib> > g (n); // граф в формате списков смежности
    vector<rib2> ribs (m); // все рёбра в одном списке
    ... чтение графа ...

    int nn = m+2, s = nn-2, t = nn-1;
    vector < vector<int> > f (nn, vector<int> (nn));
    vector < vector<int> > u (nn, vector<int> (nn));
    vector < vector<int> > c (nn, vector<int> (nn));
    for (int i=n-1; i<m; ++i) {
        vector<int> q (n);
        int h=0, t=0;
        rib2 & cur = ribs[i];
        q[t++] = cur.a;
        rib_id[cur.a] = -2;
        while (h < t) {
            int v = q[h++];
            for (size_t j=0; j<g[v].size(); ++j)
                if (g[v][j].id >= n-1)
                    break;
                else if (rib_id[g[v][j].v] == -1) {
                    rib_id[g[v][j].v] = g[v][j].id;
                    q[t++] = g[v][j].v;
                }
        }
        for (int v=cur.b, pv; v!=cur.a; v=pv) {
            int r = rib_id[v];
            pv = v != ribs[r].a ? ribs[r].a : ribs[r].b;
            u[r][i] = n;
            c[r][i] = ribs[i].c - ribs[r].c;
            c[i][r] = -c[r][i];
        }
    }
    u[s][t] = n+1;
    for (int i=0; i<n-1; ++i)
        u[s][i] = 1;
    for (int i=n-1; i<m; ++i)
        u[i][t] = 1;

    vector<int> pi (nn);
    pi[s] = INF;
    for (int i=0; i<n-1; ++i) {
        pi[i] = INF;
        for (int j=n-1; j<m; ++j)
            if (u[i][j])
                pi[i] = min (pi[i], ribs[j].c-ribs[i].c);
        pi[s] = min (pi[s], pi[i]);
    }

    for (;;) {
        vector<int> id (nn);
        deque<int> q;
        q.push_back (s);
        vector<int> d (nn, INF);
        d[s] = 0;
        vector<int> p (nn, -1);
        while (!q.empty()) {
            int v = q.front(); q.pop_front();
            id[v] = 2;
            for (int i=0; i<nn; ++i)
                if (f[v][i] < u[v][i]) {
                    int new_d = d[v] + c[v][i] - pi[v] + pi[i];
                    if (new_d < d[i]) {
                        d[i] = new_d;
                        if (id[i] == 0)
                            q.push_back (i);
                        else if (id[i] == 2)
                            q.push_front (i);
                        id[i] = 1;
                        p[i] = v;
                    }
                }
        }
    }
}

```

```

    }
}

for (int i=0; i<nn; ++i)
    pi[i] -= d[i];
for (int v=t; v!=s; v=p[v]) {
    int pv = p[v];
    ++f[pv][v], --f[v][pv];
}
if (p[t] == s) break;
}

for (int i=0; i<m; ++i)
    pi[i] -= pi[s];
for (int i=0; i<n-1; ++i)
    if (f[s][i])
        ribs[i].c += pi[i];
for (int i=n-1; i<m; ++i)
    if (f[i][t])
        ribs[i].c += pi[i];
...
вывод графа ...
}

```

Покраска рёбер дерева

Это достаточно часто встречающаяся задача. Дано дерево G. Поступают запросы двух видов: первый вид - покрасить некоторое ребро, второй вид - запрос количества покрашенных рёбер между двумя вершинами.

Здесь будет описано достаточно простое решение (с использованием [дерева отрезков](#)), которое будет отвечать на запросы за $O(\log N)$, с препроцессингом (предварительной обработкой дерева) за $O(M)$.

Решение

Для начала нам придётся реализовать [LCA](#), чтобы каждый запрос второго вида (i,j) сводить к двум запросам (a,b) , где a - предок b .

Теперь опишем [препроцессинг](#) собственно для нашей задачи. Запустим поиск в глубину из корня дерева, этот поиск в глубину составит некоторый список посещения вершин (каждая вершина добавляется в список, когда поиск заходит в неё, и каждый раз после того, как поиск в глубину возвращается из сына текущей вершины) - кстати говоря, этот же список используется алгоритмом LCA. В этом списке будет присутствовать каждое ребро (в том смысле, что если i и j - концы ребра, то в списке обязательно найдётся место, где i и j идут подряд друг за другом), причём присутствовать ровно 2 раза: в прямом направлении (из i в j , где вершина i ближе к корню, чем вершина j) и в обратном (из j в i).

Построим два дерева отрезков (для суммы, с единичной модификацией) по этому списку: T_1 и T_2 . Дерево T_1 будет учитывать каждое ребро только в прямом направлении, а дерево T_2 - наоборот, только в обратном.

Пусть поступил очередной **запрос** вида (i,j) , где i - предок j , и требуется определить, сколько рёбер покрашено на пути между i и j . Найдём i и j в списке обхода в глубину (нам обязательно нужны позиции, где они встречаются впервые), пусть это некоторые позиции p и q (это можно сделать за $O(1)$, если вычислить эти позиции заранее во время препроцессинга). Тогда **ответом будет сумма $T_1[p..q-1]$ - сумма $T_2[p..q-1]$** .

Почему? Рассмотрим отрезок $[p;q]$ в списке обхода в глубину. Он содержит рёбра нужного нам пути из i в j , но также содержит и множество рёбер, которые лежат на других путях из i . Однако между нужными нам рёбрами и остальными рёбрами есть одно большое отличие: нужные рёбра будут содержаться в этом списке только один раз, причём в прямом направлении, а все остальные рёбра будут встречаться дважды: и в прямом, и в обратном направлении. Следовательно, разность $T_1[p..q-1] - T_2[p..q-1]$ даст нам ответ (минус один нужно, потому что иначе мы захватим ещё лишнее ребро из вершины j куда-то вниз или вверх). Запрос суммы в дереве отрезков выполняется за $O(\log N)$.

Ответ на **запрос** вида 1 (о покраске какого-либо ребра) ещё проще - нам просто нужно обновить T_1 и T_2 , а именно выполнить единичную модификацию того элемента, который соответствует нашему ребру (найти ребро в списке обхода, опять же, можно за $O(1)$, если выполнить этот поиск в препроцессинге). Единичная модификация в дереве отрезков выполняется за $O(\log N)$.

Реализация

Здесь будет приведена полная реализация решения, включая LCA:

```

const int INF = 1000*1000*1000;

typedef vector<vector<int>> graph;

vector<int> dfs_list;
vector<int> ribs_list;
vector<int> h;

void dfs (int v, const graph & g, const graph & rib_ids, int cur_h = 1)

```

```

{
    h[v] = cur_h;
    dfs_list.push_back(v);
    for (size_t i=0; i<g[v].size(); ++i)
        if (h[g[v][i]] == -1)
    {
        ribs_list.push_back(rib_ids[v][i]);
        dfs(g[v][i], g, rib_ids, cur_h+1);
        ribs_list.push_back(rib_ids[v][i]);
        dfs_list.push_back(v);
    }
}

vector<int> lca_tree;
vector<int> first;

void lca_tree_build (int i, int l, int r)
{
    if (l == r)
        lca_tree[i] = dfs_list[l];
    else
    {
        int m = (l + r) >> 1;
        lca_tree_build(i+i, l, m);
        lca_tree_build(i+i+1, m+1, r);
        int lt = lca_tree[i+i], rt = lca_tree[i+i+1];
        lca_tree[i] = h[lt] < h[rt] ? lt : rt;
    }
}

void lca_prepare (int n)
{
    lca_tree.assign(dfs_list.size() * 8, -1);
    lca_tree_build(1, 0, (int)dfs_list.size()-1);

    first.assign(n, -1);
    for (int i=0; i < (int)dfs_list.size(); ++i)
    {
        int v = dfs_list[i];
        if (first[v] == -1) first[v] = i;
    }
}

int lca_tree_query (int i, int tl, int tr, int l, int r)
{
    if (tl == l && tr == r)
        return lca_tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)
        return lca_tree_query(i+i, tl, m, l, r);
    if (l > m)
        return lca_tree_query(i+i+1, m+1, tr, l, r);
    int lt = lca_tree_query(i+i, tl, m, l, m);
    int rt = lca_tree_query(i+i+1, m+1, tr, m+1, r);
    return h[lt] < h[rt] ? lt : rt;
}

int lca (int a, int b)
{
    if (first[a] > first[b]) swap(a, b);
    return lca_tree_query(1, 0, (int)dfs_list.size()-1, first[a], first[b]);
}

vector<int> first1, first2;
vector<char> rib_used;
vector<int> tree1, tree2;

void query_prepare (int n)
{
    first1.resize(n-1, -1);
    first2.resize(n-1, -1);
    for (int i = 0; i < (int) ribs_list.size(); ++i)
    {
        int j = ribs_list[i];
        if (first1[j] == -1)
            first1[j] = i;
        else
            first2[j] = i;
    }

    rib_used.resize(n-1);
    tree1.resize(ribs_list.size() * 8);
    tree2.resize(ribs_list.size() * 8);
}

```

```

void sum_tree_update (vector<int> & tree, int i, int l, int r, int j, int delta)
{
    tree[i] += delta;
    if (l < r)
    {
        int m = (l + r) >> 1;
        if (j <= m)
            sum_tree_update (tree, i+i, l, m, j, delta);
        else
            sum_tree_update (tree, i+i+1, m+1, r, j, delta);
    }
}

int sum_tree_query (const vector<int> & tree, int i, int tl, int tr, int l, int r)
{
    if (l > r || tl > tr)  return 0;
    if (tl == l && tr == r)
        return tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)
        return sum_tree_query (tree, i+i, tl, m, l, r);
    if (l > m)
        return sum_tree_query (tree, i+i+1, m+1, tr, l, r);
    return sum_tree_query (tree, i+i, tl, m, l, m)
        + sum_tree_query (tree, i+i+1, m+1, tr, m+1, r);
}

int query (int v1, int v2)
{
    return sum_tree_query (tree1, 1, 0, (int)ribs_list.size()-1, first[v1], first[v2]-1)
        - sum_tree_query (tree2, 1, 0, (int)ribs_list.size()-1, first[v1], first[v2]-1);
}

int main()
{
    // чтение графа
    int n;
    scanf ("%d", &n);
    graph g (n), rib_ids (n);
    for (int i=0; i<n-1; ++i)
    {
        int v1, v2;
        scanf ("%d%d", &v1, &v2);
        --v1, --v2;
        g[v1].push_back (v2);
        g[v2].push_back (v1);
        rib_ids[v1].push_back (i);
        rib_ids[v2].push_back (i);
    }

    h.assign (n, -1);
    dfs (0, g, rib_ids);
    lca_prepare (n);
    query_prepare (n);

    for (;;) {
        if () {
            // запрос о покраске ребра с номером x;
            // если start=true, то ребро красится, иначе покраска снимается
            rib_used[x] = start;
            sum_tree_update (tree1, 1, 0, (int)ribs_list.size()-1, first1[x], start?1:-1);
            sum_tree_update (tree2, 1, 0, (int)ribs_list.size()-1, first2[x], start?1:-1);
        }
        else {
            // запрос кол-ва покрашенных рёбер на пути между v1 и v2
            int l = lca (v1, v2);
            int result = query (l, v1) + query (l, v2);
            // result - ответ на запрос
        }
    }
}

```

Задача 2-SAT

Задача 2-SAT (2-satisfiability) - это задача распределения значений булевым переменным таким образом, чтобы они удовлетворяли всем наложенным ограничениям.

Задачу 2-SAT можно представить в виде конъюнктивной нормальной формы, где в каждом выражении в скобках стоит ровно по две переменной; такая форма называется 2-CNF (2-conjunctive normal form). Например:

```
(a || c) && (a || !d) && (b || !d) && (b || !e) && (c || d)
```

Приложения

Алгоритм для решения 2-SAT может быть применен во всех задачах, где есть набор величин, каждая из которых может принимать 2 возможных значения, и есть связи между этими величинами:

- **Расположение текстовых меток на карте или диаграмме.**

Имеется в виду нахождение такого расположения меток, при котором никакие две не пересекаются.

Стоит заметить, что в общем случае, когда каждая метка может занимать множество различных позиций, мы получаем задачу general satisfiability, которая является NP-полной. Однако, если ограничиться только двумя возможными позициями, то полученная задача будет задачей 2-SAT.

- **Расположение рёбер при рисовании графа.**

Аналогично предыдущему пункту, если ограничиться только двумя возможными способами провести ребро, то мы придём к 2-SAT.

- **Составление расписания игр.**

Имеется в виду такая система, когда каждая команда должна сыграть с каждой по одному разу, а требуется распределить игры по типу домашняя-выездная, с некоторыми наложенными ограничениями.

- и т.д.

Алгоритм

Сначала приведём задачу к другой форме - так называемой импликативной форме. Заметим, что выражение вида $a \parallel b$ эквивалентно $\neg a \Rightarrow b$ или $\neg b \Rightarrow a$. Это можно воспринимать следующим образом: если есть выражение $a \parallel b$, и нам необходимо добиться обращения его в true, то, если $a=false$, то необходимо $b=true$, и наоборот, если $b=false$, то необходимо $a=true$.

Построим теперь так называемый **граф импликаций**: для каждой переменной в графе будет по две вершины, обозначим их через x_i и $\neg x_i$. Рёбра в графе будут соответствовать импликативным связям.

Например, для 2-CNF формы:

```
(a || b) && (b || !c)
```

Граф импликаций будет содержать следующие рёбра (ориентированные):

```
!a => b  
!b => a  
!b => !c  
c => b
```

Стоит обратить внимание на такое свойство графа импликаций, что если есть ребро $a \Rightarrow b$, то есть и ребро $\neg b \Rightarrow \neg a$.

Теперь заметим, что если для какой-то переменной x выполняется, что из x достижимо $\neg x$, а из $\neg x$ достижимо x , то задача решения не имеет. Действительно, какое бы значение для переменной x мы бы ни выбрали, мы всегда придём к противоречию - что должно быть выбрано и обратное ему значение. Оказывается, что это условие является не только достаточным, но и необходимым (доказательством этого факта будет описанный ниже алгоритм). Переформулируем данный критерий в терминах теории графов. Напомним, что если из одной вершины достижима другая, а из той вершины достижима первая, то эти две вершины находятся в одной сильно связанной компоненте. Тогда мы можем сформулировать **критерий существования решения** следующим образом:

Для того, чтобы данная задача 2-SAT имела решение, необходимо и достаточно, чтобы для любой переменной x вершины x и $\neg x$ находились в разных компонентах сильной связности графа импликаций.

Этот критерий можно проверить за время $O(N + M)$ с помощью [алгоритма поиска сильно связных компонент](#).

Теперь построим собственно **алгоритм** нахождения решения задачи 2-SAT в предположении, что решение существует.

Заметим, что, несмотря на то, что решение существует, для некоторых переменных может выполняться, что из x достижимо $\neg x$, или (но не одновременно), из $\neg x$ достижимо x . В таком случае выбор одного из значений переменной x будет приводить к противоречию, в то время как выбор другого - не будет. Научимся выбирать из двух значений то, которое не приводит к возникновению противоречий. Сразу заметим, что, выбрав какое-либо значение, мы должны запустить из него обход в глубину/ширину и пометить все значения, которые следуют из него, т.е. достижимы в графе импликаций. Соответственно, для уже помеченных вершин никакого выбора между x и $\neg x$ делать не нужно, для них значение уже выбрано и зафиксировано. Нижеописанное правило применяется только к непомеченным ещё вершинам.

Утверждается следующее. Пусть $comp[v]$ обозначает номер компоненты сильной связности, которой принадлежит вершина v , причём номера упорядочены в порядке топологической сортировки компонент сильной связности в графе компонентов (т.е. более ранним в порядке топологической сортировки соответствуют большие номера: если есть путь из v в w , то $comp[v] \leq comp[w]$). Тогда, если $comp[x] < comp[\neg x]$, то выбираем значение $\neg x$, иначе, т.е. если $comp[x] > comp[\neg x]$, то выбираем x .

Докажем, что при таком выборе значений мы не придём к противоречию. Пусть, для определённости, выбрана вершина x (случай, когда выбрана вершина $\neg x$, доказывается симметрично).

Во-первых, докажем, что из x не достижимо $\neg x$. Действительно, так как номера компоненты сильной связности $comp[x]$ больше номера компоненты $comp[\neg x]$, то это означает, что компонента связности, содержащая x , расположена левее компоненты связности, содержащей $\neg x$, и из первой никак не может быть достижима последняя.

Во-вторых, докажем, что никакая вершина y , достижимая из x , не является "плохой", т.е. неверно, что из y достижимо $\neg y$. Докажем это от противного. Пусть из x достижимо y , а из y достижимо $\neg y$. Так как из x достижимо y , то, по свойству графа импликаций, из $\neg y$ будет достижимо $\neg x$. Но, по предположению, из y достижимо $\neg y$. Тогда мы получаем, что из x достижимо $\neg x$, что противоречит условию, что и требовалось доказать.

Итак, мы построили алгоритм, который находит искомые значения переменных в предположении, что для любой переменной x вершины x и $\neg x$ находятся в разных компонентах сильной связности. Выше показали корректность этого алгоритма. Следовательно, мы одновременно доказали указанный выше критерий существования решения.

Теперь мы можем собрать **весь алгоритм** воедино:

- Построим граф импликаций.
- Найдём в этом графе компоненты сильной связности за время $O(N + M)$, пусть $\text{comp}[v]$ - это номер компоненты сильной связности, которой принадлежит вершина v .
- Проверим, что для каждой переменной x вершины x и $\neg x$ лежат в разных компонентах, т.е. $\text{comp}[x] \neq \text{comp}[\neg x]$. Если это условие не выполняется, то вернуть "решение не существует".
- Если $\text{comp}[x] > \text{comp}[\neg x]$, то переменной x выбираем значение `true`, иначе - `false`.

Реализация

Ниже приведена реализация решения задачи 2-SAT для уже построенного графа импликаций g и обратного ему графа gt (т.е. в котором направление каждого ребра изменено на противоположное).

Программа выводит номера выбранных вершин, либо фразу "NO SOLUTION", если решения не существует.

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (!used[to])
            dfs1 (to);
    }
    order.push_back (v);
}

void dfs2 (int v, int cl) {
    comp[v] = cl;
    for (size_t i=0; i<gt[v].size(); ++i) {
        int to = gt[v][i];
        if (comp[to] == -1)
            dfs2 (to, cl);
    }
}

int main() {
    ... чтение n, графа g, построение графа gt ...

    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs1 (i);

    comp.assign (n, -1);
    for (int i=0, j=0; i<n; ++i) {
        int v = order[n-i-1];
        if (comp[v] == -1)
            dfs2 (v, j++);
    }

    for (int i=0; i<n; ++i)
        if (comp[i] == comp[i^1]) {
            puts ("NO SOLUTION");
            return 0;
        }
    for (int i=0; i<n; ++i) {
        int ans = comp[i] > comp[i^1] ? i : i^1;
        printf ("%d ", ans);
    }
}
```

Даны N отрезков на прямой, т.е. каждый отрезок задаётся парой координат (X_1, X_2) . Рассмотрим объединение этих отрезков и найдём его длину.

Алгоритм был предложен Кли (Klee) в 1977 году. Алгоритм работает за $O(N \log N)$. Было доказано, что этот алгоритм является быстрейшим (асимптотически).

Описание

Положим все координаты концов отрезков в массив X и отсортируем его по значению координаты. Дополнительное условие при сортировке - при равенстве координат первыми должны идти левые концы. Кроме того, для каждого элемента массива будем хранить, относится он к левому или к правому концу отрезка. Теперь пройдёмся по всему массиву, имея счётчик C перекрывающихся отрезков. Если C отлично от нуля, то к результату добавляем разницу $X_i - X_{i-1}$. Если текущий элемент относится к левому концу, то увеличиваем счётчик C , иначе уменьшаем его.

Реализация

```
unsigned segments_union_measure (const vector <pair <int,int> > & a)
{
    unsigned n = a.size();
    vector <pair <int,bool> > x (n*2);
    for (unsigned i=0; i<n; i++)
    {
        x[i*2] = make_pair (a[i].first, false);
        x[i*2+1] = make_pair (a[i].second, true);
    }

    sort (x.begin(), x.end());

    unsigned result = 0;
    unsigned c = 0;
    for (unsigned i=0; i<n*2; i++)
    {
        if (c && i)
            result += unsigned (x[i].first - x[i-1].first);
        if (x[i].second)
            ++c;
        else
            --c;
    }
    return result;
}
```

Ориентированная площадь треугольника и предикат "По часовой стрелке"

Определение

Пусть даны три точки p_1, p_2, p_3 . Найдём значение **ориентированной площади** треугольника $p_1p_2p_3$, т.е. площади этого треугольника, взятой со знаком плюс или минус в зависимости от типа поворота, образуемого точками p_1, p_2, p_3 : против часовой стрелки или по ней соответственно.

Понятно, что, если мы научимся вычислять такую ориентированную ("знаковую") площадь, то сможем и находить обычную площадь любого треугольника, и проверять, по часовой стрелке или против направлена какая-либо тройка точек.

Вычисление

Воспользуемся понятием **косого** (псевдоскалярного) произведения векторов. Оно как раз равно удвоенной ориентированной площади треугольника:

$$a \wedge b = |a||b| \sin \angle(a, b)$$

Где угол $\angle(a, b)$ берётся ориентированным, т.е. это угол вращения между этими векторами против часовой стрелки.

Косое произведение вычисляется как величина определителя, составленного из координат точек:

$$\begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$$

Раскрывая определитель, можно получить такую формулу:

$$x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

Можно сгруппировать третье слагаемое с первыми двумя, избавившись от одного умножения:

$$(x_1 - x_3)(y_2 - y_3) + (x_2 - x_3)(y_3 - y_1)$$

Реализация

Реализации функций, вычисляющих соответственно ориентированную, обычную площадь треугольника, а также две функции проверки, по/против часовой стрелке направлена тройка точек или нет.

```
int triangle_square_2 (int x1, int y1, int x2, int y2, int x3, int y3) {
    return x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2);
}

double triangle_square (int x1, int y1, int x2, int y2, int x3, int y3) {
    return abs (triangle_square_2 (x1, y1, x2, y2, x3, y3)) / 2.0;
}

bool clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
    return triangle_square_2 (x1, y1, x2, y2, x3, y3) < 0;
}

bool counter_clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
    return triangle_square_2 (x1, y1, x2, y2, x3, y3) > 0;
}
```

Проверка двух отрезков на пересечение

Даны два отрезка AB и CD , возможно, вырожденные в точку. Требуется проверить, пересекаются они или нет.

Первый способ: ориентированная площадь треугольника

Воспользуемся [Ориентированной площадью треугольника и предикат 'По часовой стрелке'](#). Действительно, чтобы отрезки AB и CD пересекались, необходимо и достаточно, чтобы точки A и B находились по разные стороны прямой CD , и, аналогично, точки C и D — по разные стороны прямой AB . Проверить это можно, вычисляя ориентированные площади соответствующих треугольников и сравнивая их знаки.

Единственное, на что следует обратить внимание — граничные случаи, когда какие-то точки попадают на саму прямую. При этом возникает единственный особый случай, когда вышеописанные проверки ничего не дадут — случай, когда оба отрезка лежат на [одной прямой](#). Этот случай надо рассмотреть отдельно. Для этого достаточно проверить, что проекции этих двух отрезков на оси X и Y пересекаются.

Реализация:

```
struct pt {
    int x, y;
};

int square (pt a, pt b, pt c) {
    return a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y);
}

bool intersect_1 (int a, int b, int c, int d) {
    return max (a, b) >= min (c, d) && max (c, d) >= min (a, b);
}

bool intersect (pt a, pt b, pt c, pt d) {
    int s11 = square (a, b, c);
    int s12 = square (a, b, d);
    int s21 = square (c, d, a);
    int s22 = square (c, d, b);
    if (s11 == 0 && s12 == 0 && s21 == 0 && s22 == 0)
        return intersect_1 (a.x, b.x, c.x, d.x)
            && intersect_1 (a.y, b.y, c.y, d.y);
    else
        return (s11 * s12 <= 0) && (s21 * s22 <= 0);
}
```

В целях оптимизации здесь можно вместо вызовов функции `square` подставить сами формулы, и вместо многократного вычисления

одинаковых слагаемых запоминать их во временных переменных.

Второй способ: пересечение двух прямых

Вместо пересечения отрезков выполним [пересечение двух прямых](#), в результате, если прямые не параллельны, получим какую-то точку, которую надо проверить на принадлежность обоим отрезкам; для этого достаточно проверить, что эта точка принадлежит обоим отрезкам в проекции на ось X и на ось Y .

Если же прямые оказались параллельными, то, если они не совпадают, то отрезки точно не пересекаются. Если же прямые совпали, то отрезки лежат на одной прямой, и для проверки их пересечения достаточно проверить, что пересекаются их проекции на ось X и Y .

Остается ещё особый случай, когда один или оба отрезка **вырождаются** в точки: в таком случае говорить о прямых некорректно. По коду получится, как будто прямые были совпадающими, и будет выполнена только проверка на пересечение в проекциях, однако этой проверки будет недостаточно. Поэтому этот случай тоже придётся разбирать особо (надо будет выполнить проверку принадлежности точки отрезку).

Этот способ оказывается проще первого способа, если заранее известно, что никакие два отрезка не лежат на одной прямой, и отрезки невырождены. Ну и, конечно, преимущество его в том, что он находит и саму точку пересечения. Недостаток — в использовании дробной арифметики.

Реализация (без учёта случая вырожденных отрезков):

```
struct pt {
    int x, y;
};

int det (int a, int b, int c, int d) {
    return a * d - b * c;
}

const double EPS = 1E-9;

bool in (int a, int b, double c) {
    return min(a,b) <= c + EPS && c - EPS <= max(a,b);
}

bool intersect_1 (int a, int b, int c, int d) {
    return max (a, b) >= min (c, d) && max (c, d) >= min (a, b);
}

bool intersect (pt a, pt b, pt c, pt d) {
    int A1 = a.y-b.y, B1 = b.x-a.x, C1 = -A1*a.x - B1*a.y;
    int A2 = c.y-d.y, B2 = d.x-c.x, C2 = -A2*c.x - B2*c.y;
    int zn = det (A1, B1, A2, B2);
    if (zn != 0) {
        double x = - det (C1, B1, C2, B2) * 1. / zn;
        double y = - det (A1, C1, A2, C2) * 1. / zn;
        return in (a.x, b.x, x) && in (a.y, b.y, y)
            && in (c.x, d.x, x) && in (c.y, d.y, y);
    }
    else
        return det (A1, C1, A2, C2) == 0 && det (B1, C1, B2, C2) == 0
        && intersect_1 (a.x, b.x, c.x, d.x)
        && intersect_1 (a.y, b.y, c.y, d.y);
}
```

Здесь сначала вычисляется коэффициент zn — знаменатель в формуле Крамера. Если $zn = 0$, то коэффициенты A и B прямых пропорциональны, и прямые параллельны или совпадают. В этом случае надо проверить, совпадают они или нет, для чего надо проверить, что коэффициенты C прямых пропорциональны с тем же коэффициентом, для чего достаточно вычислить два следующих определителя, если они оба равны нулю, то прямые совпадают:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

Если же $zn \neq 0$, то прямые пересекаются, и по формуле Крамера находим точку пересечения (x, y) и проверяем её принадлежность обоим отрезкам.

Нахождение уравнения прямой для отрезка

Уравнение прямой имеет вид $Ax + By + C = 0$. Найдём значения коэффициентов A, B, C для любого заданного отрезка.

Понятно, что, вообще говоря, таких наборов значений существует бесконечное множество.

Способ 1

Мы устраним неоднозначность таким образом: коэффициент В, если отличен от нуля, должен быть равен единице; если же коэффициент В равен нулю (вертикальная прямая), что тогда А должно быть равно единице.

Вот готовые формулы для не вертикальных прямых (их легко получить, если представить уравнение прямой в виде $y = Kx + B$):

```
A = - (Y1 - Y2) / (X1 - X2),
B = 1,
C = - (A * X1) - (B * Y1)
```

Для вертикальных прямых имеем:

```
A = 1,
B = 0,
C = - X1
```

Реализация:

```
struct segment {
    double x1, y1, x2, y2;
};

struct line {
    double a, b, c;
};

const double EPS = 1e-6;

bool eq (double a, double b)
{
    return fabs (a-b) < EPS;
}

void segment_to_line (const segment & s, line & l)
{
    if (eq (s.x1, s.x2))
    {
        l.a = 1;
        l.b = 0;
        l.c = - s.x1;
    }
    else
    {
        l.a = - (s.y1 - s.y2) / (s.x1 - s.x2);
        l.b = 1;
        l.c = - (l.a * s.x1 + l.b * s.y1);
    }
}
```

Способ 2

По-другому избежать неоднозначности можно следующим образом:

```
A = Y1 - Y2
B = X2 - X1
C = - (A * X1 + B * Y1)
```

В корректности этих формул можно убедиться непосредственной подстановкой.

Способ 2 обычно лучше способа 1 в том смысле, что если все координаты целочисленные, то и коэффициенты в уравнении прямой также получатся целочисленными.

Точка пересечения прямых

Пусть нам даны две прямые, заданные своими коэффициентами A_1, B_1, C_1 и A_2, B_2, C_2 . Требуется найти их точку пересечения, или выяснить, что прямые параллельны.

Решение

Если две прямые не параллельны, то они пересекаются. Чтобы найти точку пересечения, достаточно составить из двух уравнений прямых систему и решить её:

$$\begin{cases} A_1x + B_1y + C_1 = 0, \\ A_2x + B_2y + C_2 = 0. \end{cases}$$

Пользуясь формулой Крамера, сразу находим решение системы, которое и будет искомой **точкой пересечения**:

$$x = -\frac{\begin{vmatrix} C_1 & B_1 \\ C_2 & B_2 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{C_1B_2 - C_2B_1}{A_1B_2 - A_2B_1},$$
$$y = -\frac{\begin{vmatrix} A_2 & C_2 \\ A_1 & C_1 \end{vmatrix}}{\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}} = -\frac{A_1C_2 - A_2C_1}{A_1B_2 - A_2B_1}.$$

Если знаменатель нулевой, т.е.

$$\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} = A_1B_2 - A_2B_1 = 0$$

то система решений не имеет (прямые **параллельны** и не совпадают) или имеет бесконечно много (прямые **совпадают**). Если необходимо различить эти два случая, надо проверить, что коэффициенты C прямых пропорциональны с тем же коэффициентом пропорциональности, что и коэффициенты A и B , для чего достаточно посчитать два определителя, если они оба равны нулю, то прямые совпадают:

$$\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}$$

Реализация

```
struct pt {
    double x, y;
};

struct line {
    double a, b, c;
};

const double EPS = 1e-9;

double det (double a, double b, double c, double d) {
    return a * d - b * c;
}

bool intersect (line m, line n, pt & res) {
    double zn = det (m.a, m.b, n.a, n.b);
    if (abs (zn) < EPS)
        return false;
    res.x = - det (m.c, m.b, n.c, n.b) / zn;
    res.y = - det (m.a, m.c, n.a, n.c) / zn;
    return true;
}

bool parallel (line m, line n) {
    return abs (det (m.a, m.b, n.a, n.b)) < EPS;
}

bool equivalent (line m, line n) {
    return abs (det (m.a, m.b, n.a, n.b)) < EPS
    && abs (det (m.a, m.c, n.a, n.c)) < EPS
    && abs (det (m.b, m.c, n.b, n.c)) < EPS;
}
```

Точка пересечения отрезков

Пусть нам даны два отрезка, заданные координатами своих концов. Требуется алгоритм, который определял, пересекаются ли отрезки, и если да, то возвращал координаты точки пересечения (если она одна) или целого отрезка - их пересечения.

Решение довольно просто, если мы воспользуемся полученными результатами: Нахождение уравнения прямой для отрезка и Точка пересечения прямых.

Реализация

```
int segments_intersection (segment s1, segment s2, point & p1, point & p2)
{
    if (!intersect (s1, s2))
        return 0;
    if (s1 == s2)
    {
        p1.x = s1.x1;
        p1.y = s2.y1;
        return 1;
    }

    line l1, l2;
    segment_to_line (s1, l1);
    segment_to_line (s2, l2);
    if (!same_line (l1, l2))
    {
        lines_intersection (l1, l2, p1);
        return 1;
    }

    if (s1.x1 > s1.x2 || s1.x2 == s1.x2 && s1.y1 > s1.y2)
        swap (s1.x1, s1.x2), swap (s1.y1, s1.y2);
    if (s2.x1 > s2.x2 || s2.x1 == s2.x2 && s2.y1 > s2.y2)
        swap (s2.x1, s2.x2), swap (s2.y1, s2.y2);
    if (s1.x1 > s2.x1 || s1.x1 == s2.x1 && s1.y1 > s2.y1)
        p1.x = s1.x1, p1.y = s1.y1;
    else
        p1.x = s2.x1, p1.y = s2.y1;
    if (s1.x2 < s2.x2 || s1.x2 == s2.x2 && s1.y2 < s2.y2)
        p2.x = s1.x2, p2.y = s1.y2;
    else
        p2.x = s2.x2, p2.y = s2.y2;
    if (p1 == p2)
        return 1;
    return 2;
}
```

Нахождение площади простого многоугольника

Пусть дан простой многоугольник (т.е. без самопересечений, но не обязательно выпуклый), заданный координатами своих вершин в порядке обхода по или против часовой стрелки. Требуется найти его площадь.

Способ 1

Это легко сделать, если перебрать все рёбра и сложить площади трапеций, ограниченных каждым ребром. Площадь нужно брать с тем знаком, с каким она получится (именно благодаря знаку вся "лишняя" площадь сократится). Т.е. формула такова:

```
S += (X2 - X1) * (Y1 + Y2) / 2
```

Код:

```
double sq (const vector<point> & fig)
{
    double res = 0;
    for (unsigned i=0; i<fig.size(); i++)
    {
        point
        p1 = i ? fig[i-1] : fig.back(),
        p2 = fig[i];
        res += (p1.x - p2.x) * (p1.y + p2.y);
    }
    return fabs (res) / 2;
}
```

Способ 2

Можно поступить другим образом. Выберем произвольно точку O , переберём все рёбра, прибавляя к ответу ориентированную площадь треугольника, образованного ребром и точкой O (см. [Ориентированная площадь треугольника](#)). Опять же, благодаря знаку, вся лишняя площадь сократится, и останется только ответ.

Этот способ хорош тем, что его проще обобщить на более сложные случаи (например, когда некоторые стороны - не прямые, а дуги окружности).

Теорема Пика. Нахождение площади решётчатого многоугольника

Многоугольник без самопересечений называется решётчатым, если все его вершины находятся в точках с целочисленными координатами (в некоторой декартовой системе координат).

Теорема Пика

Пусть дан некоторый решётчатый многоугольник. Обозначим его площадь через S ; количество точек с целочисленными координатами, лежащих строго внутри многоугольника - через I ; количество точек с целочисленными координатами, лежащих на сторонах многоугольника - через B . Тогда справедливо соотношение:

$$S = I + B/2 - 1$$

В частности, если известны значения I и B для некоторого многоугольника, то его площадь можно посчитать за $O(1)$.

Это соотношение было открыто и доказано Пиком (Pick) в 1899 г.

Задача о покрытии отрезков точками

Дано N отрезков на прямой. Требуется покрыть их наименьшим числом точек, т.е. найти наименьшее множество точек такое, что каждому отрезку принадлежит хотя бы одна точка.

Также рассмотрим усложнённый вариант этой задачи - когда дополнительно указано "запрещённое" множество отрезков, т.е. никакая точка из ответа не должна принадлежать ни одному запрещённому отрезку.

Следует также заметить, что эту задачу можно рассматривать и как задачу в теории расписаний - требуется покрыть заданный набор мероприятий-отрезков наименьшим числом точек.

Ниже будет описан жадный алгоритм, решающий обе задачи за $O(N \log N)$.

Решение первой задачи

Заметим сначала, что можно рассматривать только те решения, в которых каждая из точек находится на правом конце какого-либо отрезка. Действительно, нетрудно понять, что любое решение, если оно не удовлетворяет этому свойству, можно привести к нему, сдвигая его точки вправо настолько, насколько это возможно.

Попытаемся теперь построить решение, удовлетворяющее указанному свойству. Возьмём точки-правые концы отрезков, отсортируем их, и будем двигаться по ним слева направо. Если текущая точка является правым концом уже покрытого отрезка, то мы пропускаем её. Пусть теперь текущая точка является правым концом текущего отрезка, который ещё не был покрыт до этого. Тогда мы должны добавить в ответ текущую точку, и отметить все отрезки, которым принадлежит эта точка, как покрытые. Действительно, если бы мы пропустили текущую точку и не стали бы добавлять её в ответ, то, так как она является правым концом текущего отрезка, то мы уже не смогли бы покрыть текущий отрезок.

Однако при наивной реализации этот метод будет работать за $O(N^2)$. Опишем **эффективную реализацию** этого метода.

Возьмём все точки-концы отрезков (как левые, так и правые) и отсортируем их. При этом для каждой точки сохраним вместе с ней номер отрезка, а также то, каким концом его она является (левым или правым). Кроме того, отсортируем точки таким образом, что, если есть несколько точек с одной координатой, то сначала будут идти левые концы, и только потом - правые. Заведём стек, в котором будут храниться номера отрезков, рассматриваемых в данный момент; изначально стек пуст. Будем двигаться по точкам в отсортированном порядке. Если текущая точка - левый конец, то просто добавляем номер её отрезка в стек. Если же она является правым концом, то

проверяем, не был ли покрыт этот отрезок (для этого можно просто завести массив булевых переменных). Если он уже был покрыт, то ничего не делаем и переходим к следующей точке (забегая вперёд, мы утверждаем, что в этом случае в стеке текущего отрезка уже нет). Если же он ещё не был покрыт, то мы добавляем текущую точку в ответ, и теперь мы хотим отметить для всех текущих отрезков, что они становятся покрытыми. Поскольку в стеке как раз хранятся номера непокрытых ещё отрезков, то будем доставать из стека по одному отрезку и отмечать, что он уже покрыт, пока стек полностью не опустеет. По окончании работы алгоритма все отрезки будут покрыты, и притом наименьшим числом точек (повторимся, здесь важно требование, что при равенстве координат сначала идут левые концы, и только затем правые).

Таким образом, весь алгоритм выполняется за $O(N)$, не считая сортировки точек, а итоговая сложность алгоритма как раз равна $O(N \log N)$.

Решение второй задачи

Здесь уже появляются запрещённые отрезки, поэтому, во-первых, решения вообще может не существовать, а во-вторых, уже нельзя утверждать, что ответ можно составить только из правых концов отрезков. Однако описанный выше алгоритм можно соответствующим образом модифицировать.

Снова возьмём все точки-концы отрезков (как целевых отрезков, так и запрещённых), отсортируем их, сохранив вместе с каждой точкой её тип и отрезок, концом которого она является. Оять же, отсортируем отрезки так, чтобы при равенстве координат левые концы шли перед правыми, а если и типы концов равны, то левые концы запрещённых должны идти перед левыми концами целевых, а правые концы запрещённых - после целевых (чтобы запрещённые отрезки учитывались как можно дальше при равенстве координат). Заведём счётчик запрещённых отрезков, который будет равен числу запрещённых отрезков, покрывающих текущую точку. Заведём очередь (queue), в которой будут храниться номера текущих целевых отрезков. Будем перебирать точки в отсортированном порядке. Если текущая точка - левый конец целевого отрезка, то просто добавим номер её отрезка в очередь. Если текущая точка - правый конец целевого отрезка, то, если счётчик запрещённых отрезков равен нулю, то мы поступаем аналогично предыдущей задаче - ставим точку в текущую точку, и выталкиваем все отрезки из очереди, отмечая, что они покрыты. Если же счётчик запрещённых отрезков больше нуля, то в текущую точку мы стрелять не можем, а потому мы должны найти самую последнюю точку, свободную от запрещённых отрезков; для этого надо поддерживать соответствующий указатель last_free, который будет обновляться при поступлении запрещённых отрезков. Тогда мы стреляем в last_free-EPS (потому что прямо в неё нельзя стрелять - эта точка принадлежит запрещённому отрезку), и выталкивать отрезки из очереди, пока точка last_free-EPS принадлежит им. А именно, если текущая точка - левый конец запрещённого отрезка, то мы увеличиваем счётчик, и если перед этим счётчик был равен нулю, то присваиваем last_free текущую координату. Если текущая точка - правый конец запрещённого отрезка, то просто уменьшаем счётчик.

Пересечение окружности и прямой

Дана окружность (координатами своего центра и радиусом) и прямая (своим уравнением). Требуется найти точки их пересечения (одна, две, либо ни одной).

Решение

Вместо формального решения системы двух уравнений подойдём к задаче **с геометрической стороны** (причём, за счёт этого мы получим более точное решение с точки зрения численной устойчивости).

Предположим, не теряя общности, что центр окружности находится в начале координат (если это не так, то перенесём его туда, исправив соответствующие константу C в уравнении прямой). Т.е. имеем окружность с центром в (0,0) радиуса r и прямую с уравнением $Ax + By + C = 0$.

Сначала найдём **ближайшую к центру точку** прямой - точку с некоторыми координатами (x_0, y_0) . Во-первых, эта точка должна находиться на таком расстоянии от начала координат:

$$\begin{aligned} |C| \\ \hline \hline \sqrt{A^2+B^2} \end{aligned}$$

Во-вторых, поскольку вектор (A,B) перпендикулярен прямой, то координаты этой точки должны быть пропорциональны координатам этого вектора. Учитывая, что расстояние от начала координат до искомой точки нам известно, нам нужно просто нормировать вектор (A,B) к этой длине, и мы получаем:

$$\begin{aligned} x_0 &= -\frac{A}{\sqrt{A^2+B^2}} \cdot C \\ y_0 &= -\frac{B}{\sqrt{A^2+B^2}} \cdot C \end{aligned}$$

(здесь неочевидны только знаки 'минус', но эти формулы легко проверить подстановкой в уравнение прямой - должен получиться ноль)

Зная ближайшую к центру окружности точку, мы уже можем определить, сколько точек будет содержать ответ, и даже дать ответ, если этих точек 0 или 1.

Действительно, если расстояние от (x_0, y_0) до начала координат (а его мы уже выразили формулой - см. выше) больше радиуса, то **ответ -**

ноль точек. Если это расстояние равно радиусу, то **ответом будет одна точка** - (x_0, y_0) . А вот в оставшемся случае точек будет две, и их координаты нам предстоит найти.

Итак, мы знаем, что точка (x_0, y_0) лежит внутри круга. Искомые точки (ax, ay) и (bx, by) , помимо того что должны принадлежать прямой, должны лежать на одном и том же расстоянии d от точки (x_0, y_0) , причём это расстояние легко найти:

$$d = \sqrt{r^2 - \frac{c^2}{A^2+B^2}}$$

Заметим, что вектор $(-B, A)$ коллинеарен прямой, а потому искомые точки (ax, ay) и (bx, by) можно получить, прибавив к точке (x_0, y_0) вектор $(-B, A)$, нормированный к длине d (мы получим одну искомую точку), и вычтя этот же вектор (получим вторую искомую точку).

Окончательное решение такое:

$$\begin{aligned} d^2 &= \text{sqrt} \left(\frac{c^2}{A^2+B^2} \right) \\ ax &= x_0 + B \cdot d \\ ay &= y_0 - A \cdot d \\ bx &= x_0 - B \cdot d \\ by &= y_0 + A \cdot d \end{aligned}$$

Если бы мы решали эту задачу чисто алгебраически, то скорее всего получили бы решение в другом виде, которое даёт большую погрешность. Поэтому "геометрический" метод, описанный здесь, помимо наглядности, ещё и более точен.

Реализация

Как и было указано в начале описания, предполагается, что окружность расположена в начале координат.

Поэтому входные параметры - это радиус окружности и коэффициенты A, B, C уравнения прямой.

```
double r, a, b, c; // входные данные

double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
if (c*c > r*r*(a*a+b*b)+EPS)
    puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < EPS) {
    puts ("1 point");
    cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b));
    double ax,ay,bx,by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx << ' ' << by << '\n';
}
```

Пересечение двух окружностей

Даны две окружности, каждая определена координатами своего центра и радиусом. Требуется найти все их точки пересечения (либо одна, либо две, либо ни одной точки, либо окружности совпадают).

Решение

Сведём нашу задачу к задаче о **Пересечении окружности и прямой**.

Предположим, не теряя общности, что центр первой окружности - в начале координат (если это не так, то перенесём центр в начало координат, а при выводе ответа будем обратно прибавлять координаты центра). Тогда мы имеем систему двух уравнений:

$$\begin{aligned} x^2 + y^2 &= r_1^2 \\ (x - x_2)^2 + (y - y_2)^2 &= r_2^2 \end{aligned}$$

Вычтем из второго уравнения первое, чтобы избавиться от квадратов переменных:

$$\begin{aligned}x^2 + y^2 &= r_1^2 \\x(-2x_2) + y(-2y_2) + (x_2^2 + y_2^2 + r_1^2 - r_2^2) &= 0\end{aligned}$$

Таким образом, мы свели задачу о пересечении двух окружностей к задаче о пересечении первой окружности и следующей прямой:

$$\begin{aligned}Ax + By + C &= 0, \\A &= -2x_2, \\B &= -2y_2, \\C &= x_2^2 + y_2^2 + r_1^2 - r_2^2.\end{aligned}$$

А решение последней задачи описано в [соответствующей статье](#).

Единственный **вырожденный случай**, который надо рассмотреть отдельно - когда центры окружностей совпадают. Действительно, в этом случае вместо уравнения прямой мы получим уравнение вида $0 = C$, где C - некоторое число, и этот случай будет обрабатываться некорректно. Поэтому этот случай нужно рассматривать отдельно: если радиусы окружностей совпадают, то ответ - бесконечность, иначе - точек пересечения нет.

Построение выпуклой оболочки обходом Грэхэма

Даны N точек на плоскости. Построить их выпуклую оболочку, т.е. наименьший выпуклый многоугольник, содержащий все эти точки.

Мы рассмотрим метод **Грэхэма** (Graham) (предложен в 1972 г.) с улучшениями Эндрю (Andrew) (1979 г.). С его помощью можно построить выпуклую оболочку за время $O(N \log N)$ с использованием только операций сравнения, сложения и умножения. Алгоритм является асимптотически оптимальным (доказано, что не существует алгоритма с лучшей асимптотикой), хотя в некоторых задачах он неприемлем (в случае параллельной обработки или при online-обработке).

Описание

Алгоритм. Найдём самую левую и самую правую точки A и B (если таких точек несколько, то возьмём самую нижнюю среди левых, и самую верхнюю среди правых). Понятно, что и A , и B обязательно попадут в выпуклую оболочку. Далее, проведём через них прямую AB , разделив множество всех точек на верхнее и нижнее подмножества $S1$ и $S2$ (точки, лежащие на прямой, можно отнести к любому множеству - они всё равно не войдут в оболочку). Точки A и B отнесём к обоим множествам. Теперь построим для $S1$ верхнюю оболочку, а для $S2$ - нижнюю оболочку, и объединим их, получив ответ. Чтобы получить, скажем, верхнюю оболочку, нужно отсортировать все точки по абсциссе, затем пройтись по всем точкам, рассматривая на каждом шаге кроме самой точки две предыдущие точки, вошедшие в оболочку. Если текущая тройка точек образует не правый поворот (что легко проверить с помощью [Ориентированной площади](#)), то ближайшего соседа нужно удалить из оболочки. В конце концов, останутся только точки, входящие в выпуклую оболочку.

Итак, алгоритм заключается в сортировке всех точек по абсциссе и двух (в худшем случае) обходах всех точек, т.е. требуемая асимптотика $O(N \log N)$ достигнута.

Реализация

```
struct pt {
    double x, y;
};

bool cmp (pt a, pt b) {
    return a.x < b.x || a.x == b.x && a.y < b.y;
}

bool cw (pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw (pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
}

void convex_hull (vector<pt> & a) {
    if (a.size() == 1) return;
    sort (a.begin(), a.end(), &cmp);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back (p1);
    down.push_back (p1);
    for (size_t i=1; i<a.size(); ++i) {
        if (i==a.size()-1 || cw (p1, a[i], p2)) {
```

```

        up.pop_back();
        up.push_back(a[i]);
    }
    if (i==a.size()-1 || ccw (p1, a[i], p2)) {
        while (down.size()>=2 && !ccw (down[down.size()-2], down[down.size()-1], a[i]))
            down.pop_back();
        down.push_back(a[i]);
    }
}
a.clear();
for (size_t i=0; i<up.size(); ++i)
    a.push_back(up[i]);
for (size_t i=down.size()-2; i>0; --i)
    a.push_back(down[i]);
}

```

Нахождение площади объединения треугольников. Метод вертикальной декомпозиции

Даны N треугольников. Требуется найти площадь их объединения.

Решение

Здесь мы рассмотрим метод **вертикальной декомпозиции**, который в задачах на геометрию часто оказывается очень важным.

Итак, у нас имеется N треугольников, которые могут как угодно пересекаться друг с другом. Избавимся от этих пересечений с помощью вертикальной декомпозиции: найдём все точки пересечения всех отрезков (образующих треугольники), и отсортируем найденные точки. Пусть мы получили некоторый массив B. Будем двигаться по этому массиву. На i-ом шаге рассматриваем элементы B[i] и B[i+1]. Мы имеем вертикальную полосу между прямыми $X = B[i]$ и $X = B[i+1]$, причём, согласно самому построению массива B, внутри этой полосы отрезки никак не пересекаются друг с другом. Следовательно, внутри этой полосы треугольники обрезаются до трапеций, причём стороны этих трапеций внутри полосы не пересекаются вообще. Будем двигаться по сторонам этих трапеций снизу вверх, и складывать площади трапеций, следя за тем, чтобы каждый кусок был учтён ровно один раз. Фактически, этот процесс очень напоминает обработку вложенных скобок. Сложив площади трапеций внутри каждой полосы, и сложив результаты для всех полос, мы и найдём ответ - площадь объединения треугольников.

Рассмотрим ещё раз процесс сложения площадей трапеций, уже с точки зрения реализации. Мы перебираем все стороны всех треугольников, и если какая-то сторона (не вертикальная, нам вертикальные стороны не нужны, и даже наоборот, будут сильно мешать) попадает в эту вертикальную полосу (полностью или частично), то мы кладём эту сторону в некоторый вектор, удобнее всего это делать в таком виде: координаты Y в точках пересечения стороны с границами вертикальной полосы, и номер треугольника. После того, как мы построили этот вектор, содержащий куски сторон, сортируем его по значению Y: сначала по левой Y, потом по правой Y. В результате первый в векторе элемент будет содержать нижнюю сторону самой нижней трапеции. Теперь мы просто идём по полученному вектору. Пусть i - текущий элемент; это означает, что i-ый кусок - это нижняя сторона некоторой трапеции, некоторого блока (который может содержать несколько трапеций), площадь которого мы хотим сразу прибавить к ответу. Поэтому мы устанавливаем некий счётчик треугольников равным 1, и поднимаемся по отрезкам вверх, и увеличиваем счётчик, если мы встречаем сторону какого-то треугольника в первый раз, и уменьшаем счётчик, если мы встречаем треугольник во второй раз. Если на каком-то отрезке j счётчик стал равным нулю, то мы нашли верхнюю границу блока - на этом мы останавливаемся, прибавляем площадь трапеции, ограниченной отрезками i и j, и i присваиваем j+1, и повторяем весь процесс заново.

Итак, благодаря методу вертикальной декомпозиции мы решили эту задачу, из геометрических примитивов использовав только пересечение двух отрезков.

Реализация

```

struct segment {
    int x1, y1, x2, y2;
};

struct point {
    double x, y;
};

struct item {
    double y1, y2;
    int triangle_id;
};

struct line {
    int a, b, c;
};

```

```

const double EPS = 1E-7;

void intersect (segment s1, segment s2, vector<point> & res) {
    line l1 = { s1.y1-s1.y2, s1.x2-s1.x1, l1.a*s1.x1+l1.b*s1.y1 },
    l2 = { s2.y1-s2.y2, s2.x2-s2.x1, l2.a*s2.x1+l2.b*s2.y1 };
    double det1 = l1.a * l2.b - l1.b * l2.a;
    if (abs (det1) < EPS) return;
    point p = { (l1.c * 1.0 * l2.b - l1.b * 1.0 * l2.c) / det1,
                (l1.a * 1.0 * l2.c - l1.c * 1.0 * l2.a) / det1 };
    if (p.x >= s1.x1-EPS && p.x <= s1.x2+EPS && p.x >= s2.x1-EPS && p.x <= s2.x2+EPS)
        res.push_back (p);
}

double segment_y (segment s, double x) {
    return s.y1 + (s.y2 - s.y1) * (x - s.x1) / (s.x2 - s.x1);
}

bool eq (double a, double b) {
    return abs (a-b) < EPS;
}

vector<item> c;

bool cmp_y1_y2 (int i, int j) {
    const item & a = c[i];
    const item & b = c[j];
    return a.y1 < b.y1-EPS || abs (a.y1-b.y1) < EPS && a.y2 < b.y2-EPS;
}

int main() {

    int n;
    cin >> n;
    vector<segment> a (n*3);
    for (int i=0; i<n; ++i) {
        int x1, y1, x2, y2, x3, y3;
        scanf ("%d%d%d%d%d", &x1,&y1,&x2,&y2,&x3,&y3);
        segment s1 = { x1,y1,x2,y2 };
        segment s2 = { x1,y1,x3,y3 };
        segment s3 = { x2,y2,x3,y3 };
        a[i*3] = s1;
        a[i*3+1] = s2;
        a[i*3+2] = s3;
    }

    for (size_t i=0; i<a.size(); ++i)
        if (a[i].x1 > a[i].x2)
            swap (a[i].x1, a[i].x2), swap (a[i].y1, a[i].y2);

    vector<point> b;
    b.reserve (n*n*3);
    for (size_t i=0; i<a.size(); ++i)
        for (size_t j=i+1; j<a.size(); ++j)
            intersect (a[i], a[j], b);

    vector<double> xs (b.size());
    for (size_t i=0; i<b.size(); ++i)
        xs[i] = b[i].x;
    sort (xs.begin(), xs.end());
    xs.erase (unique (xs.begin(), xs.end(), &eq), xs.end());

    double res = 0;
    vector<char> used (n);
    vector<int> cc (n*3);
    c.resize (n*3);
    for (size_t i=0; i+1<xs.size(); ++i) {
        double x1 = xs[i], x2 = xs[i+1];
        size_t csz = 0;
        for (size_t j=0; j<a.size(); ++j)
            if (a[j].x1 != a[j].x2)
                if (a[j].x1 <= x1+EPS && a[j].x2 >= x2-EPS) {
                    item it = { segment_y (a[j], x1), segment_y (a[j], x2), (int)j/3 };
                    cc[csz] = (int)csz;
                    c[csz++] = it;
                }
        sort (cc.begin(), cc.begin()+csz, &cmp_y1_y2);
        double add_res = 0;
        for (size_t j=0; j<csz; ) {
            item lower = c[cc[j++]];
            used[lower.triangle_id] = true;
            int cnt = 1;
            while (cnt && j<csz) {
                char & cur = used[c[cc[j++]].triangle_id];
                cur = !cur;

```

```

    if (cur) ++cnt; else --cnt;
}
item upper = c[cc[j-1]];
add_res += upper.y1 - lower.y1 + upper.y2 - lower.y2;
}
res += add_res * (x2 - x1) / 2;
}

cout.precision (8);
cout << fixed << res;
}

```

Проверка точки на принадлежность выпуклому многоугольнику

Дан выпуклый многоугольник с N вершинами, координаты всех вершин целочисленны (хотя это не меняет суть решения); вершины заданы в порядке обхода против часовой стрелки (в противном случае нужно просто отсортировать их). Поступают запросы - точки, и требуется для каждой точки определить, лежит она внутри этого многоугольника или нет (границы многоугольника включаются). На каждый запрос будем отвечать в режиме on-line за $O(\log N)$. Предварительная обработка многоугольника будет выполняться за $O(N)$.

Алгоритм

Решать будем бинарным поиском по углу.

Один из вариантов решения таков. Выберем точку с наименьшей координатой X (если таких несколько, то выбираем самую нижнюю, т.е. с наименьшим Y). Относительно этой точки, обозначим её $Zero$, все остальные вершины многоугольника лежат в правой полуплоскости. Далее, заметим, что все вершины многоугольника уже упорядочены по углу относительно точки $Zero$ (это вытекает из того, что многоугольник выпуклый, и уже упорядочен против часовой стрелки), причём все углы находятся в промежутке $(-\pi/2 ; \pi/2]$.

Пусть поступает очередной запрос - некоторая точка P . Рассмотрим её полярный угол относительно точки $Zero$. Найдём бинарным поиском две такие соседние вершины L и R многоугольника, что полярный угол P лежит между полярными углами L и R . Тем самым мы нашли тот сектор многоугольника, в котором лежит точка P , и нам остаётся только проверить, лежит ли точка P в треугольнике $(Zero, L, R)$. Это можно сделать, например, с помощью [Ориентированной площади треугольника](#) и [Предиката "По часовой стрелке"](#), достаточно посмотреть, по часовой стрелке или против находится тройка вершин (R, L, P) .

Таким образом, мы за $O(\log N)$ находим сектор многоугольника, а затем за $O(1)$ проверяем принадлежность точки треугольнику, и, следовательно, требуемая асимптотика достигнута. Предварительная обработка многоугольника заключается только в том, чтобы предпосчитать полярные углы для всех точек, хотя, эти вычисления тоже можно перенести на этап бинарного поиска.

Замечания по реализации

Чтобы определять полярный угол, можно воспользоваться стандартной функцией atan2 . Тем самым мы получим очень короткое и простое решение, однако взамен могут возникнуть проблемы с точностью.

Учитывая, что изначально все координаты являются целочисленными, можно получить решение, вообще не использующее дробной арифметики.

Заметим, что полярный угол точки (X, Y) относительно начала координат однозначно определяется дробью Y/X , при условии, что точка находится в правой полуплоскости. Более того, если у одной точки полярный угол меньше, чем у другой, то и дробь Y_1/X_1 будет меньше Y_2/X_2 , и обратно.

Таким образом, для сравнения полярных углов двух точек нам достаточно сравнить дроби Y_1/X_1 и Y_2/X_2 , что уже можно выполнить в целочисленной арифметике.

Реализация

Эта реализация предполагает, что в данном многоугольнике нет повторяющихся вершин, и площадь многоугольника ненулевая.

```

struct pt {
    int x, y;
};

struct ang {
    int a, b;
};

bool operator < (const ang & p, const ang & q) {
    if (p.b == 0 && q.b == 0)
        return p.a < q.a;
    return p.a * 111 * q.b < p.b * 111 * q.a;
}

long long sq (pt & a, pt & b, pt & c) {

```

```

    return a.x*111*(b.y-c.y) + b.x*111*(c.y-a.y) + c.x*111*(a.y-b.y);
}

int main() {

    int n;
    cin >> n;
    vector<pt> p (n);
    int zero_id = 0;
    for (int i=0; i<n; ++i) {
        scanf ("%d%d", &p[i].x, &p[i].y);
        if (p[i].x < p[zero_id].x || p[i].x == p[zero_id].x && p[i].y < p[zero_id].y)
            zero_id = i;
    }
    pt zero = p[zero_id];
    rotate (p.begin(), p.begin()+zero_id, p.end());
    p.erase (p.begin());
    --n;

    vector<ang> a (n);
    for (int i=0; i<n; ++i) {
        a[i].a = p[i].y - zero.y;
        a[i].b = p[i].x - zero.x;
        if (a[i].a == 0)
            a[i].b = a[i].b < 0 ? -1 : 1;
    }

    for (;;) {
        pt q; // очередной запрос
        bool in = false;
        if (q.x >= zero.x)
            if (q.x == zero.x && q.y == zero.y)
                in = true;
            else {
                ang my = { q.y-zero.y, q.x-zero.x };
                if (my.a == 0)
                    my.b = my.b < 0 ? -1 : 1;
                vector<ang>::iterator it = upper_bound (a.begin(), a.end(), my);
                if (it == a.end() && my.a == a[n-1].a && my.b == a[n-1].b)
                    it = a.end()-1;
                if (it != a.end() && it != a.begin()) {
                    int p1 = int (it - a.begin());
                    if (sq (p[p1], p[p1-1], q) <= 0)
                        in = true;
                }
            }
        puts (in ? "INSIDE" : "OUTSIDE");
    }
}

```

Нахождение вписанной окружности в выпуклом многоугольнике с помощью тернарного поиска

Дан выпуклый многоугольник с N вершинами. Требуется найти координаты центра и радиус наибольшей вписанной окружности.

Здесь описывается простой метод решения этой задачи с помощью двух тернарных поисков, работающий за $O(N \log^2 C)$, где C - коэффициент, определяемый величиной координат и требуемой точностью (см. ниже).

Алгоритм

Определим функцию **Radius (X, Y)**, возвращающую радиус вписанной в данный многоугольник окружности с центром в точке $(X;Y)$. Предполагается, что точки X и Y лежат внутри (или на границе) многоугольника. Очевидно, эту функцию легко реализовать с асимптотикой $O(N)$ - просто проходим по всем сторонам многоугольника, считаем для каждой расстояние до центра (причём расстояние можно брать как от прямой до точки, не обязательно рассматривать как отрезок), и возвращаем минимум из найденных расстояний - очевидно, он и будет наибольшим радиусом.

Итак, нам нужно максимизировать эту функцию. Заметим, что, поскольку многоугольник выпуклый, то эта функция будет пригодна для **тернарного поиска** по обоим аргументам: при фиксированном X_0 (разумеется, таком, что прямая $X=X_0$ пересекает многоугольник) функция $\text{Radius}(X_0, Y)$ как функция одного аргумента Y будет сначала возрастать, затем убывать (опять же, мы рассматриваем только такие

Y , что точка (X_0, Y) принадлежит многоугольнику). Более того, функция \max (по Y) { $\text{Radius}(X, Y)$ } как функция одного аргумента X будет сначала возрастать, затем убывать. Эти свойства ясны из геометрических соображений.

Таким образом, нам нужно сделать два тернарных поиска: по X и внутри него по Y , максимизируя значение функции Radius . Единственный особый момент - нужно правильно выбирать границы тернарных поисков, поскольку вычисление функции Radius за пределами многоугольника будет некорректным. Для поиска по X никаких сложностей нет, просто выбираем абсциссу самой левой и самой правой точки. Для поиска по Y находим те отрезки многоугольника, в которые попадает текущий X , и находим ординаты точек этих отрезков при абсциссе X (вертикальные отрезки не рассматриваем).

Осталось оценить **асимптотику**. Пусть максимальное значение, которое могут принимать координаты - это C_1 , а требуемая точность - порядка 10^{-C_2} , и пусть $C = C_1 + C_2$. Тогда количество шагов, которые должен будет совершить каждый тернарный поиск, есть величина $O(\log C)$, и итоговая асимптотика получается: $O(N \log^2 C)$.

Реализация

Константа steps определяет количество шагов обоих тернарных поисков.

В реализации стоит отметить, что для каждой стороны сразу предпосчитываются коэффициенты в уравнении прямой, и сразу же нормализуются (делятся на $\sqrt{A^2+B^2}$), чтобы избежать лишних операций внутри тернарного поиска.

```
const double EPS = 1E-9;
int steps = 60;

struct pt {
    double x, y;
};

struct line {
    double a, b, c;
};

double dist (double x, double y, line & l) {
    return abs (x * l.a + y * l.b + l.c);
}

double radius (double x, double y, vector<line> & l) {
    int n = (int) l.size();
    double res = INF;
    for (int i=0; i<n; ++i)
        res = min (res, dist (x, y, l[i]));
    return res;
}

double y_radius (double x, vector<pt> & a, vector<line> & l) {
    int n = (int) a.size();
    double ly = INF, ry = -INF;
    for (int i=0; i<n; ++i) {
        int x1 = a[i].x, x2 = a[(i+1)%n].x, y1 = a[i].y, y2 = a[(i+1)%n].y;
        if (x1 == x2) continue;
        if (x1 > x2) swap (x1, x2), swap (y1, y2);
        if (x1 <= x+EPS && x-EPS <= x2) {
            double y = y1 + (x - x1) * (y2 - y1) / (x2 - x1);
            ly = min (ly, y);
            ry = max (ry, y);
        }
    }
    for (int sy=0; sy<steps; ++sy) {
        double diff = (ry - ly) / 3;
        double y1 = ly + diff, y2 = ry - diff;
        double f1 = radius (x, y1, l), f2 = radius (x, y2, l);
        if (f1 < f2)
            ly = y1;
        else
            ry = y2;
    }
    return radius (x, ly, l);
}

int main() {

    int n;
    vector<pt> a (n);
    ... чтение a ...

    vector<line> l (n);
    for (int i=0; i<n; ++i) {
        l[i].a = a[i].y - a[(i+1)%n].y;
        l[i].b = a[(i+1)%n].x - a[i].x;
        double sq = sqrt (l[i].a*l[i].a + l[i].b*l[i].b);
        l[i].a /= sq, l[i].b /= sq;
        l[i].c = - (l[i].a * a[i].x + l[i].b * a[i].y);
    }

    double lx = INF, rx = -INF;
    for (int i=0; i<n; ++i) {
```

```

lx = min (lx, a[i].x);
rx = max (rx, a[i].x);
}

for (int sx=0; sx<stepsx; ++sx) {
    double diff = (rx - lx) / 3;
    double x1 = lx + diff, x2 = rx - diff;
    double f1 = y_radius (x1, a, 1), f2 = y_radius (x2, a, 1);
    if (f1 < f2)
        lx = x1;
    else
        rx = x2;
}

double ans = y_radius (lx, a, 1);
printf ("% .7lf", ans);
}

```

Нахождение вписанной окружности в выпуклом многоугольнике методом "сжатия сторон"

Дан выпуклый многоугольник с N вершинами. Требуется найти координаты центра и радиус наибольшей вписанной окружности.

В отличие от описанного [здесь](#) метода тернарного поиска, при данном методе решения время работы - $O(N \log N)$ - не зависит от ограничений на координаты и от точности, и поэтому этот метод проходит при значительно больших N.

Спасибо [mf](#) за описание этого красивого алгоритма.

Алгоритм

Итак, дан выпуклый многоугольник. Начнём одновременно и с одинаковой скоростью **сдвигать** все его стороны параллельно самим себе внутрь многоугольника:

Пусть, для удобства, это движение происходит со скоростью 1 координатная единица в секунду (т.е. время в данном случае - это расстояние от сторон до их новых положений).

Ясно, что в процессе этого движения стороны многоугольника будут постепенно исчезать (обращаться в точки). Наконец, заметим, что время, через которое весь многоугольник сожмётся в точку или отрезок, и будет являться ответом на задачу (искомым радиусом; центр искомой окружности будет лежать на этой точке (или отрезке)).

Научимся эффективно моделировать этот процесс. Для этого научимся для каждой стороны **определять время**, через которое она сожмётся в точку в результате движения её соседей.

Для этого рассмотрим внимательно процесс движения сторон. Заметим, что вершины многоугольника всегда двигаются по биссектрисам углов (это следует из равенства соответствующих треугольников). Но тогда вопрос о времени, через которое сторона сожмётся, сводится к вопросу об определении высоты треугольника, в котором известна одна сторона A и два прилежащих к ней угла α и β . Воспользовавшись, например, теоремой синусов, получаем формулу:

$$H = A \sin(\alpha) \sin(\beta) / \sin(\alpha+\beta)$$

Теперь мы умеем за $O(1)$ определять время, через которое сторона сожмётся в точку.

Занесём эти времена для каждой стороны в некую **структуру данных для извлечения минимума**, например, set (доступ к произвольному элементу нам также понадобится, см. ниже).

Будем **извлекать** по одной стороне с наименьшим временем. Каждую извлечённую сторону надо **удалить** из многоугольника, это выражается в том, что соседи этой стороны становятся соседями друг друга, т.е. для них надо пересчитать значение времени, также увеличиваются их длины (эти стороны надо продлить до их пересечения). Таким образом, для каждой стороны надо будет завести указатели на её соседей. Если в какой-то момент удаляемой стороны соседи параллельны, то на этом процесс удаления сторон надо остановить, и ответом будет время исчезновения текущей стороны (это ясно из смысла самого алгоритма - мы удаляем текущую вершину, т.е. её соседи сдвигаются друг к другу, но если они параллельны, то они совпадут, и больше сдвигать стороны мы не сможем). Также мы останавливаем процесс, если остаётся только две стороны, и ответом будет время исчезновения последней удалённой стороны (опять же, если остаётся только две стороны, то получается, что весь многоугольник сжался до отрезка, и ответом будет время, за которое мы достигли этого состояния, т.е. время сжатия последней удалённой стороны).

Очевидно, асимптотика этого метода $O(N \log N)$, поскольку алгоритм состоит из $O(N)$ шагов, на каждом из которых на операции со структурой данных затрачивается $O(\log N)$, и $O(1)$ на все остальные операции.

Из вычислительной геометрии нам потребуется только нахождение угла между двумя сторонами, пересечение двух прямых и проверка двух прямых на параллельность.

Реализация

Программа, которая выводит радиус вписанной окружности:

```
const double EPS = 1E-9;
const double INF = 1E+40;

struct pt {
    double x, y;
    pt() { }
    pt (double x, double y) : x(x), y(y) { }
};

double get_ang (pt & a, pt & b) {
    double angl = atan2 (a.y, a.x);
    double ang2 = atan2 (b.y, b.x);
    double ang = abs (angl - ang2);
    return min (ang, 2*M_PI-ang);
}

pt vec (pt & a, pt & b) {
    return pt (b.x-a.x, b.y-a.y);
}

double dist (pt & a, pt & b) {
    return sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

double get_h (double a, double alpha, double beta) {
    return a * sin(alpha) * sin(beta) / sin(alpha+beta);
}

double det (double a, double b, double c, double d) {
    return a * d - b * c;
}

pt intersect_line (pt & p1, pt & p2, pt & q1, pt & q2) {
    double a1 = p1.y - p2.y;
    double b1 = p2.x - p1.x;
    double c1 = - a1 * p1.x - b1 * p1.y;
    double a2 = q1.y - q2.y;
    double b2 = q2.x - q1.x;
    double c2 = - a2 * q1.x - b2 * q1.y;
    return pt (
        - det (c1, b1, c2, b2) / det (a1, b1, a2, b2),
        - det (a1, c1, a2, c2) / det (a1, b1, a2, b2)
    );
}

bool parallel (pt & p1, pt & p2, pt & q1, pt & q2) {
    double a1 = p1.y - p2.y;
    double b1 = p2.x - p1.x;
    double a2 = q1.y - q2.y;
    double b2 = q2.x - q1.x;
    return abs (det (a1, b1, a2, b2)) < EPS;
}

double calc_val (pt & p1, pt & p2, pt & q1, pt & q2, pt & r1, pt & r2) {
    pt l1 = intersect_line (p1, p2, q1, q2);
    pt l2 = intersect_line (p1, p2, r1, r2);
    return get_h (dist (l1, l2), get_ang(vec(q1,q2),vec(p1,p2))/2,
        get_ang(vec(r1,r2),vec(p2,p1))/2);
}

int main() {

    int n;
    vector<pt> a (n);
    ... чтение n и a ...

    set < pair<double,int> > q;
    vector<double> val (n);
    for (int i=0; i<n; ++i) {
        pt & p1 = a[i], & p2 = a[(i+1)%n], & q1 = a[(i-1+n)%n], & q2 = a[(i+2)%n];
        val[i] = calc_val (p1, p2, p1, q1, p2, q2);
        q.insert (make_pair (val[i], i));
    }

    vector<int> next (n), prev (n);
    for (int i=0; i<n; ++i) {
        next[i] = (i + 1) % n;
        prev[i] = (i - 1 + n) % n;
    }

    double last_time;
```

```

while (q.size() > 2) {
    last_time = q.begin()->first;
    int id = q.begin()->second;
    q.erase (q.begin());
    val[id] = -1;

    next[prev[id]] = next[id];
    prev[next[id]] = prev[id];
    int nxt = next[id], prv = prev[id];
    if (parallel (a[nxt], a[(nxt+1)%n], a[prv], a[(prv+1)%n]))
        break;
    q.erase (make_pair (val[nxt], nxt));
    q.erase (make_pair (val[prv], prv));
    val[nxt] = calc_val (a[nxt], a[(nxt+1)%n], a[(prv+1)%n],
    a[prv], a[next[nxt]], a[(next[nxt]+1)%n]);
    val[prv] = calc_val (a[prv], a[(prv+1)%n], a[(prev[prv]+1)%n],
    a[prev[prv]], a[nxt], a[(nxt+1)%n]);
    q.insert (make_pair (val[nxt], nxt));
    q.insert (make_pair (val[prv], prv));
}

printf ("% .9lf", last_time);

}

```

Диаграмма Вороного в 2D

Определение

Даны n точек $P_i(x_i, y_i)$ на плоскости. Рассмотрим разбиение плоскости на n областей V_i (называемых многоугольниками Вороного или ячейками Вороного, иногда — многоугольниками близости, ячейками Дирихле, разбиением Тиссена), где V_i — множество всех точек плоскости, которые находятся ближе к точке P_i , чем ко всем остальным точкам P_k :

$$V_i = \{(x, y) : \rho((x, y), P_i) = \min_{k=1 \dots N} \rho((x, y), P_k)\}$$

Само разбиение плоскости называется диаграммой Вороного данного набора точек P_k .

Здесь $\rho(p, q)$ — заданная метрика, обычно это стандартная Евклидова метрика: $\rho(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$, однако ниже будет рассмотрен случай так называемой манхэттенской метрики. Здесь и далее, если не оговорено иного, будет рассматриваться случай Евклидовой метрики.

Ячейки Вороного представляют собой выпуклые многоугольники, некоторые являются бесконечными. Точки, принадлежащие согласно определению сразу нескольким ячейкам Вороного, обычно так и относят сразу к нескольким ячейкам (в случае Евклидовой метрики множество таких точек имеет меру нуль; в случае манхэттенской метрики всё несколько сложнее).

Такие многоугольники впервые были глубоко изучены русским математиком Вороным (1868-1908 гг.).

Свойства

- Диаграмма Вороного является планарным графом, поэтому она имеет $O(n)$ вершин и рёбер.
- Зафиксируем любое $i = 1 \dots n$. Тогда для каждого $j = 1 \dots n, j \neq i$ проведём прямую — серединный перпендикуляр отрезка (P_i, P_j) ; рассмотрим ту полуплоскость, образуемую этой прямой, в которой лежит точка P_i . Тогда пересечение всех полуплоскостей для каждого j даст ячейку Вороного P_i .
- Каждая вершина диаграммы Вороного является центром окружности, проведённой через какие-либо три точки множества P . Эти окружности существенно используются во многих доказательствах, связанных с диаграммами Вороного.
- Ячейка Вороного V_i является бесконечной тогда и только тогда, когда точка P_i лежит на границе выпуклой оболочки множества P_k .
- Рассмотрим граф, двойственный к диаграмме Вороного, т.е. в этом графе вершинами будут точки P_i , а ребро проводится между точками P_i и P_j , если их ячейки Вороного V_i и V_j имеют общее ребро. Тогда, при условии, что никакие четыре точки не лежат на одной окружности, двойственный к диаграмме Вороного граф является триангulationей Делоне (обладающей множеством интересных свойств).

Применение

Диаграмма Вороного представляет собой компактную структуру данных, хранящую всю необходимую информацию для решения множества задач о близости.

В рассмотренных ниже задачах время, необходимое на построение самой диаграммы Вороного, в асимптотиках не учитывается.

- Нахождение ближайшей точки для каждой.

Отметим простой факт: если для точки P_i ближайшей является точка P_j , то эта точка P_j имеет "своё" ребро в ячейке V_i . Отсюда следует, что, чтобы найти для каждой точки ближайшую к ней, достаточно просмотреть рёбра её ячейки Вороного. Однако каждое ребро принадлежит ровно двум ячейкам, поэтому будет просмотрено ровно два раза, и вследствие линейности числа рёбер мы получаем решение данной задачи за $O(n)$.

- Нахождение выпуклой оболочки.

Вспомним, что вершина принадлежит выпуклой оболочке тогда и только тогда, когда её ячейка Вороного бесконечна. Тогда найдём в диаграмме Вороного любое бесконечное ребро, и начнём двигаться в каком-либо фиксированном направлении (например, против часовой стрелки) по ячейке, содержащей это ребро, пока не дойдём до следующего бесконечного ребра. Тогда перейдём через это ребро в соседнюю ячейку и продолжим обход. В результате все просмотренные рёбра (кроме бесконечных) будут являться сторонами искомой выпуклой оболочки. Очевидно, время работы алгоритма - $O(n)$.

- Нахождение Евклидова минимального оствового дерева.

Требуется найти минимальное оствовое дерево с вершинами в данных точках P , соединяющее все эти точки. Если применять стандартные методы теории графов, то, т.к. граф в данном случае имеет $O(n^2)$ рёбер, даже оптимальный алгоритм будет иметь не меньшую асимптотику.

Рассмотрим граф, двойственный диаграмме Вороного, т.е. триангуляцию Делоне. Можно показать, что нахождение Евклидова минимального оства эквивалентно построению оства триангуляции Делоне. Действительно, в [алгоритме Прима](#) каждый раз ищется кратчайшее ребро между двумя множествами точек; если мы зафиксируем точку одного множества, то ближайшая к ней точка имеет ребро в ячейке Вороного, поэтому в триангуляции Делоне будет присутствовать ребро к ближайшей точке, что и требовалось доказать.

Триангуляция является планарным графом, т.е. имеет линейное число рёбер, поэтому к ней можно применить [алгоритм Крускала](#) и получить алгоритм с временем работы $O(n \log n)$.

- Нахождение наибольшей пустой окружности.

Требуется найти окружность наибольшего радиуса, не содержащую внутри никакую из точек P_i (центр окружности должен лежать внутри выпуклой оболочки точек P_i). Заметим, что, т.к. функция наибольшего радиуса окружности в данной точке $f(x, y)$ является строго монотонной внутри каждой ячейки Вороного, то она достигает своего максимума в одной из вершин диаграммы Вороного, либо в точке пересечения рёбер диаграммы и выпуклой оболочки (а число таких точек не более чем в два раза больше числа рёбер диаграммы). Таким образом, остаётся только перебрать указанные точки и для каждой найти ближайшую, т.е. решение за $O(n)$.

Простой алгоритм построения диаграммы Вороного за $O(n^4)$

Диаграммы Вороного — достаточно хорошо изученный объект, и для них получено множество различных алгоритмов, работающих за оптимальную асимптотику $O(n \log n)$, а некоторые из этих алгоритмов даже работают в среднем за $O(n)$. Однако все эти алгоритмы весьма сложны.

Рассмотрим здесь самый простой алгоритм, основанный на приведённом выше свойстве, что каждая ячейка Вороного представляет собой пересечение полуплоскостей. Зафиксируем i . Проведём между точкой P_i и каждой точкой P_j прямую — серединный перпендикуляр, затем пересечём попарно все полученные прямые — получим $O(n^2)$ точек, и каждую проверим на принадлежность всем n полуплоскостям. В результате за $O(n^3)$ действий мы получим все вершины ячейки Вороного V_i (их уже будет не более n , поэтому мы можем без ухудшения асимптотики отсортировать их по полярному углу), а всего на построение диаграммы Вороного потребуется $O(n^4)$ действий.

Случай манхэттенской метрики

Манхэттенская метрика такова:

$$\rho(p, q) = \max(|x_p - x_q|, |y_p - y_q|)$$

Начать рассмотрение следует с разбора простейшего случая — случая двух точек A и B .

Если $A_x = B_x$ или $A_y = B_y$, то диаграммой Вороного для них будет соответственно вертикальная или горизонтальная прямая.

Иначе диаграмма Вороного будет иметь вид "уголка": отрезок под углом 45 градусов в прямоугольнике, образованном точками A и B , и горизонтальные/вертикальные лучи из его концов в зависимости от того, длиннее ли вертикальная сторона прямоугольника или горизонтальная.

Особый случай — когда этот прямоугольник имеет одинаковую длину и ширину, т.е. $|A_x - B_x| = |A_y - B_y|$. В этом случае будут иметься две бесконечные области ("уголки", образованные двумя лучами, параллельными осям), которые по определению должны принадлежать сразу обеим ячейкам. В таком случае дополнительно определяют в условии, как следует понимать эти области (иногда искусственно вводят правило, по которому каждый уголок относят к своей ячейке).

Таким образом, уже для двух точек диаграмма Вороного в манхэттенской метрике представляет собой нетривиальный объект, а в случае большего числа точек эти фигуры надо будет уметь быстро пересекать.

Нахождение всех граней, внешней грани планарного графа

Дан планарный, расположенный на плоскости граф G с n вершинами. Требуется найти все его грани. Гранью называется часть плоскости,

ограниченная рёбрами этого графа.

Одна из граней будет отличаться от остальных тем, что будет иметь бесконечную площадь, такая грань называется внешней гранью. В некоторых задачах требуется находить только внешнюю грань, алгоритм нахождения которой, как мы увидим, по сути ничем не отличается от алгоритма для всех граней.

Теорема Эйлера

Приведём здесь теорему Эйлера и несколько следствий из неё, из которых будет следовать, что число рёбер и граней планарного простого (без петель и кратных рёбер) графа являются величинами порядка $O(n)$.

Пусть планарный граф G является связным. Обозначим через n число вершин в графе, m — число рёбер, f — число граней. Тогда справедлива теорема Эйлера:

$$f + n - m = 2$$

Доказать эту формулу легко следующим образом. В случае дерева ($m = n - 1$) формула легко проверяется. Если граф — не дерево, то удалим любое ребро, принадлежащее какому-либо циклу; при этом величина $f + n - m$ не изменится. Будем повторять этот процесс, пока не придём к дереву, для которого тождество $f + n - m = 2$ уже установлено. Таким образом, теорема доказана.

Следствие. Для произвольного планарного графа пусть k — количество компонент связности. Тогда выполняется:

$$f + n - m = 1 + k$$

Следствие. Число рёбер m простого планарного графа является величиной $O(n)$.

Доказательство. Пусть граф G является связным и $n \geq 3$ (в случае $n < 3$ утверждение получаем автоматически). Тогда, с одной стороны, каждая грань ограничена как минимум тремя рёбрами. С другой стороны, каждое ребро ограничивает максимум две грани. Следовательно, $3f \leq 2m$, откуда, подставляя это в формулу Эйлера, получаем:

$$f + n - m = 2 \Leftrightarrow 3f = 6 - 3n + 3m \Leftrightarrow 6 - 3n + 3m \leq 2m \Leftrightarrow m \leq 3n - 6$$

Т.е. $m = O(n)$.

Если граф не является связным, то, суммируя полученные оценки по его компонентам связности, снова получаем $m = O(n)$, что и требовалось доказать.

Следствие. Число граней f простого планарного графа является величиной $O(n)$.

Это следствие вытекает из предыдущего следствия и связи $f = 2 - n + m$.

Обход всех граней

Всегда будем считать, что граф, если он не является связным, уложен на плоскости таким образом, что никакая компонента связности не лежит внутри другой (например, квадрат с лежащим строго внутри него отрезком — некорректный для нашего алгоритма тест).

Разумеется, считается, что граф корректно уложен на плоскости, т.е. никакие две вершины не совпадают, а рёбра не пересекаются в "несанкционированных" точках. Если во входном графе допускаются такие пересекающиеся рёбра, то предварительно надо избавиться от них, вводя в каждую точку пересечения дополнительную вершину (надо заметить, что в результате этого процесса вместо n точек мы можем получить порядка n^2 точек). Более подробно об этом процессе см. ниже в соответствующем разделе.

Пусть для каждой вершины все исходящие из неё рёбра упорядочены по полярному углу. Если это не так, то их следует упорядочить, произведя сортировку каждого списка смежности (т.к. $m = O(n)$, на это потребуется $O(n \log n)$ операций).

Теперь выберем произвольное ребро (a, b) и пусть следующий обход. Приходя в какую-то вершину v по некоторому ребру, выходить из этой вершины мы обязательно должны по следующему в порядке сортировки ребру.

Например, на первом шаге мы находимся в вершине b , и должны найти вершину a в списке смежности вершины b , тогда обозначим через c следующую вершину в списке смежности (если a была последней, то в качестве c возьмём первую вершину), и пройдём по ребру (b, c) .

Повторяя этот процесс много раз, мы рано или поздно придём обратно к стартовому ребру (a, b) , после чего надо остановиться. Нетрудно заметить, что при таком обходе мы обойдём ровно одну грань. Причём направление обхода будет против часовой стрелки для внешней грани, и по часовой стрелке — для внутренних граней. Иными словами, при таком обходе внутренность грани будет всегда по правую сторону от текущего ребра.

Итак, мы научились обходить одну грань, стартуя с любого ребра на её границе. Осталось научиться выбирать стартовые рёбра таким образом, чтобы получаемые грани не повторялись. Заметим, что у каждого ребра различаются два направления, в которых его можно обходить: при каждом из них будут получаться свои грани. С другой стороны, ясно, что одно такое ориентированное ребро принадлежит ровно одной грани. Таким образом, если мы будем помечать все рёбра каждой обнаруженной грани в некотором массиве `used`, и не запускать обход из уже помеченных рёбер, то мы обойдём все грани (включая внешнюю), притом ровно по одному разу.

Приведём сразу реализацию этого обхода. Будем считать, что в графе G списки смежности уже упорядочены по углу, а кратные рёбра и петли отсутствуют.

Первый вариант реализации упрощённый, следующую вершину в списке смежности он ищёт простым поиском. Такая реализация теоретически работает за $O(n^2)$, хотя на практике на многих тестах она работает весьма быстро (со скрытой константой, значительно меньшей единицы).

```
int n; // число вершин
vector<vector<int>> g; // граф

vector<vector<char>> used (n);
for (int i=0; i<n; ++i)
    used[i].resize (g[i].size ());
for (int i=0; i<n; ++i)
    for (size_t j=0; j<g[i].size (); ++j)
        if (!used[i][j])
            used[i][j] = true;
```

```

int v = g[i][j], pv = i;
vector<int> facet;
for (;;) {
    facet.push_back (v);
    vector<int>::iterator it = find (g[v].begin(), g[v].end(), pv);
    if (++it == g[v].end()) it = g[v].begin();
    if (used[v][it-g[v].begin()]) break;
    used[v][it-g[v].begin()] = true;
    pv = v, v = *it;
}
... вывод facet - текущей грани ...
}

```

Другой, более оптимизированный вариант реализации — пользуется тем, что вершине в списке смежности упорядочены по углу. Если реализовать функцию `cmp_ang` сравнения двух точек по полярному углу относительно третьей точки (например, оформив её в виде класса, как в примере ниже), то при поиске точки в списке смежности можно воспользоваться бинарным поиском. В результате получаем реализацию за $O(n \log n)$.

```

class cmp_ang {
    int center;
public:
    cmp_ang (int center) : center(center)
    {}
    bool operator() (int a, int b) {
        ... должна возвращать true, если точка a имеет
        меньший чем b полярный угол относительно center ...
    }
};

int n; // число вершин
vector < vector<int> > g; // граф

vector < vector<char> > used (n);
for (int i=0; i<n; ++i)
    used[i].resize (g[i].size());
for (int i=0; i<n; ++i)
    for (size_t j=0; j<g[i].size(); ++j)
        if (!used[i][j]) {
            used[i][j] = true;
            int v = g[i][j], pv = i;
            vector<int> facet;
            for (;;) {
                facet.push_back (v);
                vector<int>::iterator it = lower_bound (g[v].begin(), g[v].end(),
                    pv, cmp_ang(v));
                if (++it == g[v].end()) it = g[v].begin();
                if (used[v][it-g[v].begin()]) break;
                used[v][it-g[v].begin()] = true;
                pv = v, v = *it;
            }
            ... вывод facet - текущей грани ...
        }
}

```

Возможен и вариант, основанный на контейнере `map`, ведь нам нужно всего лишь быстро узнавать позиции чисел в массиве. Разумеется, такая реализация также будет работать $O(n \log n)$.

Следует отметить, что алгоритм не совсем корректно работает с **изолированными** вершинами — такие вершины он просто не обнаружит как отдельные грани, хотя, с математической точки зрения, они должны представлять собой отдельные компоненты связности и грани.

Кроме того, особой гранью является **внешняя грань**. Как её отличать от "обычных" граней, описано в следующем разделе. Следует заметить, что если граф является не связным, то внешняя грань будет состоять из нескольких контуров, и каждый из этих контуров будет найден алгоритмом отдельно.

Выделение внешней грани

Приведённый выше код выводит все грани, не делая различия между внешней гранью и внутренними гранями. На практике обычно, наоборот, требуется найти или только внешнюю грань, или только внутренние. Есть несколько приёмов выделения внешней грани.

Например, её можно определять по площади — внешняя грань должна иметь наибольшую площадь (следует только учесть, что внутренняя грань может иметь ту же площадь, что и внешняя). Этот способ не будет работать, если данный планарный граф G не является связным.

Другой, более надёжный критерий — по направлению обхода. Как уже отмечалось выше, все грани, кроме внешней, обходятся в направлении по часовой стрелки. Внешняя грань, даже если она состоит из нескольких контуров, обойдётся алгоритмом против часовой стрелки. Определить направление обхода можно, просто посчитав **знаковую площадь многоугольника**. Площадь можно считать прямо по ходу внутреннего цикла. Однако и у этого метода есть своя тонкость — обработка граней нулевой площади. Например, если граф состоит из единственного ребра, то алгоритм найдёт единственную грань, площадь которой будет нулевой. По-видимому, если грань имеет нулевую площадь, то она является внешней гранью.

В некоторых случаях бывает применим и такой критерий, как количество вершин. Например, если граф представляет собой выпуклый многоугольник с проведёнными в нём непересекающимися диагоналями, то его внешняя грань будет содержать все вершины. Но снова надо быть аккуратным со случаем, когда и внешняя, и внутренняя грани имеют одинаковое число вершин.

Наконец есть и следующий метод нахождения внешней грани: можно специально запуститься от такого ребра, что найденная в результате грань будет внешней. Например, можно взять самую левую вершину (если таких несколько, то подойдёт любая) и выбрать из неё ребро, идущее первым в порядке сортировки. В результате обход из этого ребра найдёт внешнюю грань. Этот способ можно распространить и на случай несвязного графа: нужно в каждой компоненте связности найти самую левую вершину и запускать обход из первого ребра из неё.

Приведём реализацию самого простого метода, основанного на знаке площади (сам обход я для примера взял за $O(n^2)$, здесь это неважно). Если граф не связный, то код "... грань является внешней ..." выполнится отдельно для каждого контура, составляющего внешнюю грань.

```

int n; // число вершин
vector < vector<int> > g; // граф
vector < pair<double,double> > p; // координаты всех точек
const double EPS = 1E-9; // константа сравнения вещественных чисел

vector < vector<char> > used (n);
for (int i=0; i<n; ++i)
    used[i].resize (g[i].size());
for (int i=0; i<n; ++i)
    for (size_t j=0; j<g[i].size(); ++j)
        if (!used[i][j]) {
            used[i][j] = true;
            int v = g[i][j], pv = i;
            vector<int> facet;
            for (;;) {
                facet.push_back (v);
                vector<int>::iterator it = find (g[v].begin(), g[v].end(), pv);
                if (++it == g[v].end()) it = g[v].begin();
                if (used[v][it-g[v].begin()]) break;
                used[v][it-g[v].begin()] = true;
                pv = v, v = *it;
            }
            // считаем площадь
            double area = 0;
            // добавляем фиктивную точку для простоты подсчёта площади
            facet.push_back (facet[0]);
            for (size_t k=0; k+1<facet.size(); ++k)
                area += (p[facet[k]].first + p[facet[k+1]].first)
                    * (p[facet[k]].second - p[facet[k+1]].second);
            if (area < EPS)
                ... грань является внешней ...
        }
    }
}

```

Построение планарного графа

Для вышеописанных алгоритмов существенно то, что входной граф является корректно уложенным планарным графом. Однако на практике часто на вход программе подаётся набор отрезков, возможно, пересекающихся между собой в "несанкционированных" точках, и нужно по этим отрезкам построить планарный граф.

Реализовать построение планарного графа можно следующим образом. Зафиксируем какой-либо входной отрезок. Теперь пересечём этот отрезок со всеми остальными отрезками. Найденные точки пересечения, а также концы самого отрезка положим в вектор, и его отсортируем стандартным образом (т.е. сначала по одной координате, при равенстве — по другой). Потом пройдёмся по этому вектору и будем добавлять рёбра между соседними в этом векторе точками (разумеется, следя, чтобы мы не добавили петли). Выполнив этот процесс для всех отрезков, т.е. за $O(n^2 \log n)$, мы построим соответствующий планарный граф (в котором будет $O(n^2)$ точек).

Реализация:

```

const double EPS = 1E-9;

struct point {
    double x, y;
    bool operator< (const point & p) const {
        return x < p.x - EPS || abs (x - p.x) < EPS && y < p.y - EPS;
    }
};

map<point,int> ids;
vector<point> p;
vector < vector<int> > g;

int get_point_id (point pt) {
    if (!ids.count(pt)) {
        ids[pt] = (int)p.size();
        p.push_back (pt);
        g.resize (g.size() + 1);
    }
    return ids[p];
}

bool intersect (pair<point,point> a, pair<point,point> b, point & res) {
    ... стандартная процедура, пересекает два отрезка a и b ...
}

int main() {
    // входные данные
    int m;
    vector < pair<point,point> > a (m);
    ... чтение ...

    // построение графа
}
```

```

for (int i=0; i<m; ++i) {
    vector<point> cur;
    cur.push_back (a[i].first);
    cur.push_back (a[i].second);
    for (int j=0; j<m; ++j) {
        point t;
        if (j != i && intersect (a[i], a[j], t))
            cur.push_back (t);
    }
    sort (cur.begin(), cur.end());
    for (size_t j=0; j+1<cur.size(); ++j) {
        int x = get_id (cur[j]), y = get_id (cur[j+1]);
        if (x != y) {
            g[x].push_back (y);
            g[y].push_back (x);
        }
    }
}
int n = (int) g.size();
// сортировка по углу и удаление кратных рёбер
for (int i=0; i<n; ++i) {
    sort (g[i].begin(), g[i].end(), cmp_ang (i));
    g[i].erase (unique (g[i].begin(), g[i].end()), g[i].end());
}
}

```

Нахождение пары ближайших точек

Постановка задачи

Даны n точек p_i на плоскости, заданные своими координатами (x_i, y_i) . Требуется найти среди них такие две точки, расстояние между которыми минимально:

$$\min_{\substack{i,j=0\dots n-1, \\ i \neq j}} \rho(p_i, p_j)$$

Расстояния мы берём обычные евклиды:

$$\rho(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Тривиальный алгоритм — перебор всех пар и вычисление расстояния для каждой — работает за $O(n^2)$. Ниже описывается алгоритм, работающий за время $O(n \log n)$. Этот алгоритм был предложен Препаратой (Preparata) в 1975 г. Препарата и Шамос также показали, что в модели дерева решений этот алгоритм асимптотически оптимален.

Алгоритм

Построим алгоритм по общей схеме алгоритмов "разделяй-и-властвуй": алгоритм оформляем в виде рекурсивной функции, которой передаётся множество точек; эта рекурсивная функция разбивает это множество пополам, вызывает себя рекурсивно от каждой половины, а затем выполняет какие-то операции по объединению ответов. Операция объединения заключается в обнаружении случаев, когда одна точка оптимального решения попала в одну половину, а другая точка — в другую (в этом случае рекурсивные вызовы от каждой из половинок отдельно обнаружить эту пару, конечно, не смогут). Основная сложность, как всегда, заключается в эффективной реализации этой стадии объединения. Если рекурсивной функции передаётся множество из n точек, то стадия объединения должна работать не более, чем $O(n)$, тогда асимптотика всего алгоритма $T(n)$ будет находиться из уравнения:

$$T(n) = 2T(n/2) + O(n)$$

Решением этого уравнения, как известно, является $T(n) = O(n \log n)$.

Итак, перейдём к построению алгоритма. Чтобы в будущем прийти к эффективной реализации стадии объединения, разбивать множество точек на два будем согласно их x -координатам: фактически мы проводим некоторую вертикальную прямую, разбивающую множество точек на два подмножества примерно одинаковых размеров. Такое разбиение удобно произвести следующим образом: отсортируем точки стандартно как пары чисел, т.е.:

$$p_i < p_j \iff (x_i < x_j) \vee ((x_i = x_j) \wedge (y_i < y_j))$$

Тогда возьмём среднюю после сортировки точку p_m ($m = \lfloor n/2 \rfloor$) и все точки до неё и саму p_m отнесём к первой половине, а все точки после неё — ко второй половине:

$$A_1 = \{p_i \mid i = 0 \dots m\}$$

$$A_2 = \{p_i \mid i = m + 1 \dots n - 1\}$$

Теперь, вызвавшись рекурсивно от каждого из множеств A_1 и A_2 , мы найдём ответы h_1 и h_2 для каждой из половинок. Возьмём лучший из них: $h = \min(h_1, h_2)$.

Теперь нам надо произвести **стадию объединения**, т.е. попытаться обнаружить такие пары точек, расстояние между которыми меньше h , причём одна точка лежит в A_1 , а другая — в A_2 . Очевидно, что для этого достаточно рассматривать только те точки, которые отстоят от вертикальной прямой разделя не расстояние, меньшее h , т.е. множество B рассматриваемых на этой стадии точек равно:

$$B = \{p_i \mid |x_i - x_m| < h\}$$

Для каждой точки из множества B надо попытаться найти точки, находящиеся к ней ближе, чем h . Например, достаточно рассматривать только те точки, координата y которых отличается не более чем на h . Более того, не имеет смысла рассматривать те точки, у которых y -координата больше y -координаты текущей точки. Таким образом, для каждой точки p_i определим множество рассматриваемых точек $C(p_i)$ следующим образом:

$$C(p_i) = \{p_j \mid p_j \in B, \quad y_i - h < y_j \leq y_i\}$$

Если мы отсортируем точки множества B по y -координате, то находить $C(p_i)$ будет очень легко: это несколько точек подряд до точки p_i .

Итак, в новых обозначениях **стадия объединения** выглядит следующим образом: построить множество B , отсортировать в нём точки по y -координате, затем для каждой точки $p_i \in B$ рассмотреть все точки $p_j \in C(p_i)$, и каждой пары (p_i, p_j) посчитать расстояние и сравнить с текущим наилучшим расстоянием.

На первый взгляд, это по-прежнему неоптимальный алгоритм: кажется, что размеры множеств $C(p_i)$ будут порядка n , и требуемая асимптотика никак не получится. Однако, как это ни удивительно, можно доказать, что размер каждого из множеств $C(p_i)$ есть величина $O(1)$, т.е. не превосходит некоторой малой константы вне зависимости от самих точек. Доказательство этого факта приведено в следующем разделе.

Наконец, обратим внимание на сортировки, которых вышеописанный алгоритм содержит сразу две: сначала сортировка по парам (x, y) , а затем сортировка элементов множества B по y . На самом деле, от обеих этих сортировок внутри рекурсивной функции можно избавиться (иначе бы мы не достигли оценки $O(n)$ для стадии объединения, и общая асимптотика алгоритма получилась бы $O(n \log^2 n)$). От первой сортировки избавиться легко — достаточно предварительно, до запуска рекурсии, выполнить эту сортировку: ведь внутри рекурсии сами элементы не меняются, поэтому нет никакой необходимости выполнять сортировку заново. Со второй сортировкой чуть сложнее, выполнить её предварительно не получится. Зато, вспомнив **сортировку слиянием** (merge sort), которая тоже работает по принципу разделей-и-властвуй, можно просто встроить эту сортировку в нашу рекурсию. Пусть рекурсия, принимая какое-то множество точек (как мы помним, упорядоченное по парам (x, y)) возвращает это же множество, но отсортированное уже по координате y . Для этого достаточно просто выполнить слияние (за $O(n)$) двух результатов, возвращённых рекурсивными вызовами. Тем самым получится отсортированное по y множество.

Оценка асимптотики

Чтобы показать, что вышеописанный алгоритм действительно выполняется за $O(n \log n)$, нам осталось доказать следующий факт: $|C(p_i)| = O(1)$.

Итак, пусть мы рассматриваем какую-то точку p_i ; напомним, что множество $C(p_i)$ — это множество точек, y -координата которых не больше y_i , но и не меньше $y_i - h$, а, кроме того, по координате x и сама точка p_i , и все точки множества $C(p_i)$ лежат в полосе шириной $2h$. Иными словами, рассматриваемые нами точки p_i и $C(p_i)$ лежат в прямоугольнике размера $2h \times h$.

Наша задача — оценить максимальное количество точек, которое может лежать в этом прямоугольнике $2h \times h$; тем самым мы оценим и максимальный размер множества $C(p_i)$ (он будет на единицу меньше, т.к. в этом прямоугольнике лежит ещё и точка p_i). При этом надо не забывать, что в общем случае могут встречаться и повторяющиеся точки.

Вспомним, что h получалось как минимум из двух результатов рекурсивных вызовов — от множеств A_1 и A_2 , причём A_1 содержит точки слева от линии разделя и частично на ней, A_2 — оставшиеся точки линии разделя и точки справа от неё. Для любой пары точек из A_1 , равно как и из A_2 , расстояние не может оказаться меньше h — иначе бы это означало некорректность работы рекурсивной функции.

Для оценки максимального количества точек в прямоугольнике $2h \times h$ разобьём его на два квадрата $h \times h$, к первому квадрату отнесём все точки $C(p_i) \cap A_1$, а ко второму — все остальные, т.е. $C(p_i) \cap A_2$. Из приведённых выше соображений следует, что в каждом из этих квадратов расстояние между любыми двумя точками не превосходит h . Но тогда это означает, что в каждом квадрате не более четырёх точек!

Действительно, пусть есть квадрат $h \times h$, и расстояние между любыми двумя точками не превосходит той же h . Докажем, что в квадрате не может быть больше 4 точек. Например, это можно сделать следующим образом: разобьём этот квадрат на 4 квадрата со сторонами $h/2$. Тогда в каждом из этих маленьких квадратов не может быть больше одной точки (т.к. даже диагональ равна $h/\sqrt{2}$, что меньше h).

Следовательно, во всём квадрате никак не может быть более 4 точек.

Итак, мы доказали, что в прямоугольнике $2h \times h$ не может быть больше $4 \cdot 2 = 8$ точек, а, следовательно, размер множества $C(p_i)$ не может превосходить 7, что и требовалось доказать.

Реализация

Введём структуру данных для хранения точки (её координаты и некий номер) и операторы сравнения, необходимые для двух видов сортировки:

```
struct pt {
    int x, y, id;
};

inline bool cmp_x (const pt & a, const pt & b) {
    return a.x < b.x || a.x == b.x && a.y < b.y;
}
```

```

inline bool cmp_y (const pt & a, const pt & b) {
    return a.y < b.y;
}

pt a[MAXN];

```

Для удобной реализации рекурсии введём вспомогательную функцию upd_ans, которая будет вычислять расстояние между двумя точками и проверять, не лучше ли это текущего ответа:

```

double mindist;
int ansa, ansb;

inline void upd_ans (const pt & a, const pt & b) {
    double dist = sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + .0);
    if (dist < mindist)
        mindist = dist, ansa = a.id, ansb = b.id;
}

```

Наконец, реализация самой рекурсии. Предполагается, что перед её вызовом массив *a* уже отсортирован по компаратору cmp_x. Рекурсии передаётся просто два указателя *l*, *r*, которые указывают, что она должна искать ответ для *a[l...r]*. Если расстояние между *r* и *l* слишком мало, то рекурсию надо остановить, и выполнить тривиальный алгоритм поиска ближайшей пары и затем отсортировать подмассив по компаратору cmp_y.

Внутри рекурсии для выполнения слияния по компаратору cmp_y используется стандартная функция STL inplace_merge.

Наконец, множество *B* хранится в массиве *t*, длина которого равна tsz. Этот массив статический, т.е. общий для всех уровней рекурсии, а не выделяется заново при каждом вызове (примерно то же самое, что сделать его глобальным).

```

void rec (int l, int r) {
    if (r - l <= 3) {
        for (int i=l; i<=r; ++i)
            for (int j=i+1; j<=r; ++j)
                upd_ans (a[i], a[j]);
        sort (a+l, a+r+1, &cmp_y);
        return;
    }

    int m = (l + r) >> 1;
    int midx = a[m].x;
    rec (l, m), rec (m+1, r);
    inplace_merge (a+l, a+m+1, a+r+1, &cmp_y);

    static pt t[MAXN];
    int tsz = 0;
    for (int i=l; i<=r; ++i)
        if (abs (a[i].x - midx) < mindist) {
            for (int j=tsz-1; j>=0 && a[i].y - t[j].y < mindist; --j)
                upd_ans (a[i], t[j]);
            t[tsz++] = a[i];
        }
    }
}

```

Кстати говоря, если все координаты целые, то на время работы рекурсии можно вообще не переходить к дробным величинам, и хранить в mindist квадрат минимального расстояния.

В основной программе вызывать рекурсию следует так:

```

sort (a, a+n, &cmp_x);
mindist = 1E20;
rec (0, n-1);

```

Нахождение треугольника наименьшей площади за $O(N^2 \log N)$

Пусть даны *N* точек на плоскости. Требуется найти невырожденный треугольник наименьшей площади, вершины которого находятся в этих точках.

Алгоритм будет следующим. Последовательно переберём все точки, и для каждой точки P_i выполним следующие действия.

Рассмотрим все остальные точки, и для каждой точки будем рассматривать только её относительные координаты (относительно $P[i]$). Очевидно, что в этом случае площадь треугольника $P_i P_j P_k$ будет равна $0.5 * |X_1 * Y_2 - X_2 * Y_1|$, где X_1, Y_1, X_2, Y_2 - координаты точек P_j и P_k соответственно (см. "ориентированная площадь треугольника"). Следовательно, тот треугольник будет обладать наименьшей площадью, у которого модуль разности $X_1 * Y_2 - X_2 * Y_1$ будет наименьшим.

Если мы будем тривиально перебирать всевозможные пары вершин j и k , то общая асимптотика решения достигнет $O(N^3)$, что нас не устраивает. Однако можно пойти другим путём: зафиксировав некоторое i , отсортировать все остальные вершины, используя для сравнения предикат $X_1 * Y_2 < X_2 * Y_1$. Теперь, если мы пройдёмся по отсортированному таким образом массиву вершин и рассмотрим каждую пару соседних вершин, то среди них мы обязательно рассмотрим треугольник минимальной площади. Теперь, очевидно, асимптотика решения составит $O(N^2 \log N)$.

Реализация

```

struct point {
    double x, y;
};

bool my_cmp (point p1, point p2)
{
    return p1.x * p2.y < p2.x * p1.y;
}

const double EPS = 1E-8;
const double INF = 1E+20;

int main()
{
    int n;
    vector<point> a;
    ... чтение ...

    double result = INF;
    for (int i=0; i<n; i++)
    {
        vector<point> b (a);
        for (int j=0; j<n; j++)
            b[j].x -= a[i].x, b[j].y -= a[i].y;
        sort (b.begin(), b.end(), my_cmp);
        for (int j=1; j<n; j++)
        {
            double s = 0.5 * (b[j].x * b[j-1].y - b[j].y * b[j-1].x);
            if (s > EPS)
                result = min (result, s);
        }
    }

    cout << result;
}

```

Z-функция строки и её вычисление за $O(N)$

Z-функция ("зет-функция") от строки S - это массив Z , каждый элемент которого $Z[i]$ равен наилдлиннейшему префиксу подстроки, начинающейся с позиции i в строке S , который одновременно является и префиксом всей строки S .

Значение Z-функции в позиции 0 обычно считается не определенным (или, например, равным нулю).

Задача - посчитать Z-функцию для некоторой строки длины N за время $O(N)$.

Алгоритм

Смысл алгоритма заключается в том, чтобы при вычислении текущего значения Z-функции стараться максимально использовать уже вычисленные значения. Для этого мы будем хранить переменные-индексы L и R , которые будут указывать на начало и конец текущего префикса. Теперь, если текущая позиция I находится между L и R включительно, то мы постараемся использовать предыдущие значения. Поскольку мы находимся внутри подстроки, являющейся префиксом всей строки, то мы можем посмотреть значение Z-функции в начале строки в позиции J , соответствующей текущей позиции I , т.е. $J = I - L$. Если значение $Z[J]$ "помещается" внутри текущего префикса, т.е. $Z[J] < R - I + 1$, то выполняется $Z[I] = Z[J]$. Если же "не помещается", то придётся выполнить дополнительную работу: пройтись по символам, стоящим после окончания текущего префикса и найти последнее совпадение; теперь нам известно значение Z-функции, а также мы можем подправить значения L и R , увеличив их. Последний оставшийся случай - когда I находится вне текущего префикса, т.е. $I > R$, решается тривиальным циклом.

Реализация

```
void calc_z (const char * s, vector<unsigned> & z)
{
    unsigned len = strlen (s);
    z.resize (len);
    z[0] = 0;

    unsigned l = 0, r = 0;
    for (unsigned i = 1; i < len; i++)
        if (i > r)
        {
            unsigned j;
            for (j = 0; i+j < len && s[i+j] == s[j]; j++) ;
            z[i] = j;
            l = i;
            r = i + j - 1;
        }
        else
            if (z[i-1] < r - i + 1)
                z[i] = z[i-1];
            else
            {
                unsigned j;
                for (j = 1; r+j < len && s[r+j] == s[r-i+j]; j++) ;
                z[i] = r - i + j;
                l = i;
                r = r + j - 1;
            }
    }
}
```

Можно значительно уменьшить размер кода, правда, в ущерб понятности:

```
void calc_z (const char * s, vector<int> & z)
{
    int len = (int) strlen (s);
    z.resize (len);

    int l = 0, r = 0;
    for (int i=1; i<len; ++i)
        if (z[i-1]+i <= r)
            z[i] = z[i-1];
        else
        {
            l = i;
            if (i > r) r = i;
            for (z[i] = r-i; r<len; ++r, ++z[i])
                if (s[r] != s[z[i]])
                    break;
            --r;
        }
    }
```

Префикс-функция. Алгоритм Кнута-Морриса-Пратта

Префикс-функция. Определение

Дана строка $s[0 \dots n - 1]$. Требуется вычислить для неё префикс-функцию, т.е. массив чисел $\pi[0 \dots n - 1]$, где $\pi[i]$ определяется следующим образом: это длина наибольшего собственного суффикса подстроки $s[0..i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ всегда равно нулю.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0..i} s[0 \dots k - 1] = s[i - k + 1 \dots i]$$

Например, для строки "abcabcd" префикс-функция равна: [0, 0, 0, 1, 2, 3, 0]. Для строки "aabaaab" она равна: [0, 1, 0, 1, 2, 2, 3].

Тривиальный алгоритм

Непосредственно следуя определению, можно написать такой алгоритм вычисления префикс-функции (как нетрудно заметить, работать он будет за $O(n^3)$):

```
vector<int> prefix_function (string s) {
    int n = (int) s.length();
    vector<int> pi (n);
    for (int i=0; i<n; ++i)
        for (int k=0; k<=i; ++k)
            if (s.substr(0,k) == s.substr(i-k+1,k))
                pi[i] = k;
    return pi;
}
```

Эффективный алгоритм

Этот алгоритм был разработан Кнутом (Knuth) и Праттом (Pratt) и независимо от них Моррисом (Morris) в 1977 г. (как основной элемент для алгоритма поиска подстроки в строке).

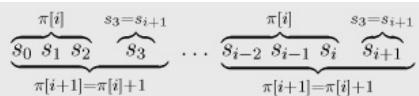
Первое важное замечание — что значение $\pi[i+1]$ не более чем на единицу превосходит значение $\pi[i]$ для любого i . Действительно, в противном случае, если бы $\pi[i+1] > \pi[i] + 1$, то рассмотрим этот суффикс, оканчивающийся в позиции $i+1$ и имеющий длину $\pi[i+1]$ — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину $\pi[i+1] - 1$, что лучше $\pi[i]$, т.е. пришли к противоречию. Иллюстрация этого противоречия (в этом примере $\pi[i-1]$ должно быть равно 3):



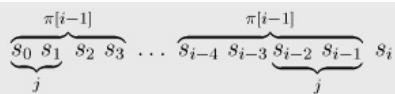
Таким образом, при переходе к следующей позиции очередной элемент префикс-функции мог либо увеличиться на единицу, либо не изменяться, либо уменьшаться на какую-либо величину. Уже этот факт позволяет нам снизить асимптотику до $O(n^2)$ — поскольку за один шаг значение могло вырасти максимум на единицу, то суммарно для всей строки могло произойти максимум n увеличений на единицу, и, как следствие (т.к. значение никогда не могло стать меньше нуля), максимум n уменьшений. В итоге получится $O(n)$ сравнений строк, т.е. мы уже достигли асимптотики $O(n^2)$.

Но пойдём дальше, и избавимся от явных сравнений подстрок. Для этого постараемся максимально использовать информацию, вычисленную на предыдущих шагах.

Итак, пусть мы вычислили значение префикс-функции $\pi[i]$ для некоторого i . Теперь, если $s[i+1] == s[\pi[i]]$, то мы можем с уверенностью сказать, что $\pi[i+1] = \pi[i] + 1$, это иллюстрирует схему:



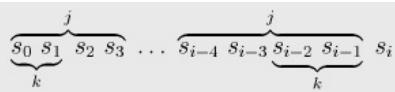
Пусть теперь, наоборот, оказалось, что $s[i+1] \neq s[\pi[i]]$. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине $j < \pi[i]$, что по-прежнему выполняется префикс-свойство в позиции $i-1$, т.е. $s[0\dots j-1] = s[i-1-j+1\dots i-1]$:



Действительно, когда мы найдём такую длину j , то нам будет снова достаточно сравнить символы $s[i+1]$ и $s[j]$ — если они совпадут, то $\pi[i+1] = j$. Иначе нам надо будет снова найти меньшее (следующее по величине) значение j , для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся, это происходит, когда $j = 0$. В этом случае, если $s[i+1] = s[0]$, то $\pi[i+1] = 1$, иначе $\pi[i+1] = 0$.

Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин j . Поставим этот вопрос формально: по текущей длине j и позиции $i-1$ (для которых выполняется префикс-свойство, т.е.

$s[0\dots j-1] = s[i-1-j+1\dots i-1]$) требуется найти наибольшее $k < j$, для которого по-прежнему выполняется префикс-свойство:



После столь подробного описания, да и из картинки, уже весьма отчётливо видно, что это значение k есть не что иное, как $\pi[j-1]$ (" -1 " появляется из-за 0-индексации, впрочем, это ясно видно на рисунке), которое уже было вычислено нами ранее. Таким образом, находить это k мы можем за $O(1)$.

Итак, мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет $O(n)$ действий. Реализация этого удивительно короткого алгоритма:

```
vector<int> prefix_function (string s) {
    int n = (int) s.length();
    vector<int> pi (n);
    for (int i=1; i<n; ++i) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j]) ++j;
        pi[i] = j;
    }
}
```

```
    }
    return pi;
}
```

Как нетрудно заметить, этот алгоритм является **онлайновым** алгоритмом, т.е. он обрабатывает данные по ходу поступления — можно, например, считывать строку по одному символу и сразу обрабатывать этот символ, находя ответ для очередной позиции. Алгоритм требует хранения самой строки и предыдущих вычисленных значений префикс-функции, однако, как нетрудно заметить, если нам заранее известно максимальное значение, которое может принимать префикс-функция на всей строке, то достаточно будет хранить лишь на единицу большее количество первых символов строки и значений префикс-функции.

Применения

Поиск подстроки в строке. Алгоритм Кнута-Морриса-Пратта

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Дан текст t и строка s , требуется найти и вывести позиции всех вхождений строки s в текст t .

Образуем строку $s + \# + t$, где символ $\#$ — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых $s.length() + 1$ (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наибольшую длину подстроки, оканчивающейся в позиции i и совпадающую с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i . Больше, чем $s.length()$ эта длина быть не может — за счёт разделителя. А вот равенство $\pi[i] = s.length()$ (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке $s + \# + t$).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = s.length()$, то в позиции

$i - s.length() + 1 - (s.length() + 1) = i - 2s.length()$ строки t начинается очередное вхождение строки s в строку t .

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку $s + \#$ и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Итак, если длины строк s и t равны n и m соответственно, то алгоритм Кнута-Морриса-Пратта решает эту задачу за $O(n + m)$ времени и $O(n)$ памяти.

Подсчёт числа вхождений каждого префикса

Здесь мы рассмотрим сразу две задачи. Дана строка s длины n . В первом варианте требуется для каждого префикса $s[0 \dots i]$ посчитать, сколько раз он встречается в самой же строке s . Во втором варианте задачи дана другая строка t , и требуется для каждого префикса $s[0 \dots i]$ посчитать, сколько раз он встречается в t .

Решим сначала первую задачу. Рассмотрим в какой-либо позиции i значение префикс-функции в ней $\pi[i]$. По определению, оно означает, что в позиции i оканчивается вхождение префикса строки s длины $\pi[i]$, и никакой больший префикс оканчиваться в позиции i не может. В то же время, в позиции i могло оканчиваться и вхождение префиксов меньших длин (и, очевидно, совсем не обязательно длины $\pi[i] - 1$). Однако, как нетрудно заметить, мы пришли к тому же вопросу, на который мы уже отвечали при рассмотрении алгоритма вычисления префикс-функции: по данной длине j надо сказать, какой наилднейший её собственный суффикс совпадает с её префиксом. Мы уже выяснили, что ответом на этот вопрос будет $\pi[j - 1]$. Но тогда и в этой задаче, если в позиции i оканчивается вхождение подстроки длины $\pi[i]$, совпадающей с префиксом, то в i также оканчивается вхождение подстроки длины $\pi[\pi[i] - 1]$, совпадающей с префиксом, а для неё применимы те же рассуждения, поэтому в i также оканчивается и вхождение длины $\pi[\pi[\pi[i] - 1] - 1]$ и так далее (пока индекс не станет нулевым). Таким образом, для вычисления ответа мы должны выполнить такой цикл:

```
vector<int> ans (n+1);
for (int i=0; i<n; ++i)
    +=ans[pi[i]];
for (int i=n-1; i>0; --i)
    ans[pi[i-1]] += ans[i];
```

Здесь мы для каждого значения префикс-функции сначала посчитали, сколько раз он встречался в массиве $\pi[]$, а затем посчитали такую в некотором роде динамику: если мы знаем, что префикс длины i встречался ровно $ans[i]$ раз, то именно такое количество надо прибавить к числу вхождений его длиннейшего собственного суффикса, совпадающего с его префиксом; затем уже из этого суффикса (конечно, меньшей чем i длины) выполнится "пробрасывание" этого количества к своему суффиксу, и т.д.

Теперь рассмотрим вторую задачу. Применим стандартный приём: припишем к строке s строку t через разделитель, т.е. получим строку $s + \# + t$, и посчитаем для неё префикс-функцию. Единственное отличие от первой задачи будет в том, что учитывать надо только те значения префикс-функции, которые относятся к строке t , т.е. все $\pi[i]$ для $i \geq s.length() + 1$.

Количество различных подстрок в строке

Дана строка s длины n . Требуется посчитать количество её различных подстрок.

Будем решать эту задачу итеративно. А именно, научимся, зная текущее количество различных подстрок, пересчитывать это количество при добавлении в конец одного символа.

Итак, пусть k — текущее количество различных подстрок строки s , и мы добавляем в конец символ c . Очевидно, в результате могли появиться некоторые новые подстроки, оканчивавшиеся на этом новом символе c . А именно, добавляются в качестве новых тех подстроки, оканчивающиеся на символе c и не встречавшиеся ранее.

Возьмём строку $t = s + c$ и инвертируем её (запишем символы в обратном порядке). Наша задача — посчитать, сколько у строки t таких префиксов, которые не встречаются в ней более нигде. Но если мы посчитаем для строки t префикс-функцию и найдём её максимальное значение π_{\max} , то, очевидно, в строке t встречается (не в начале) её префикс длины π_{\max} , но не большей длины. Понятно, префиксы меньшей длины уж точно встречаются в ней.

Итак, мы получили, что число новых подстрок, появляющихся при дописывании символа c , равно $s.length() + 1 - \pi_{\max}$.

Таким образом, для каждого дописываемого символа мы за $O(n)$ можем пересчитать количество различных подстрок строки.

Следовательно, за $O(n^2)$ мы можем найти количество различных подстрок для любой заданной строки.

Стоит заметить, что совершенно аналогично можно пересчитывать количество различных подстрок и при дописывании символа в начало, а также при удалении символа с конца или с начала.

Сжатие строки

Дана строка s длины n . Требуется найти самое короткое её "сжатое" представление, т.е. найти такую строку t наименьшей длины, что s можно представить в виде конкатенации одной или нескольких копий t .

Понятно, что проблема является в нахождении длины искомой строки t . Зная длину, ответом на задачу будет, например, префикс строки s этой длины.

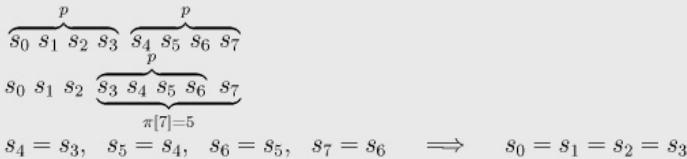
Посчитаем по строке s префикс-функцию. Рассмотрим её последнее значение, т.е. $\pi[n - 1]$, и введём обозначение $k = n - \pi[n - 1]$.

Покажем, что если n делится на k , то это k и будет длиной ответа, иначе эффективного сжатия не существует, и ответ равен n .

Действительно, пусть n делится на k . Тогда строку можно представить в виде нескольких блоков длины k , причём, по определению префикс-функции, префикс длины $n - k$ будет совпадать с её суффиксом. Но тогда последний блок должен будет совпадать с предпоследним, предпоследний - с предпредпоследним, и т.д. В итоге получится, что все блоки блоки совпадают, и такое k действительно подходит под ответ.

Покажем, что этот ответ оптимальен. Действительно, в противном случае, если бы нашлось меньшее k , то и префикс-функция на конце была бы больше, чем $n - k$, т.е. пришли к противоречию.

Пусть теперь n не делится на k . Покажем, что отсюда следует, что длина ответа равна n . Докажем от противного — предположим, что ответ существует, и имеет длину p (p делитель n). Заметим, что префикс-функция необходимо должна быть больше $n - p$, т.е. этот суффикс должен частично накрывать первый блок. Теперь рассмотрим второй блок строки; т.к. префикс совпадает с суффиксом, и и префикс, и суффикс покрывают этот блок, и их смещение друг относительно друга k не делит длину блока p (а иначе бы k делило n), то все символы блока совпадают. Но тогда строка состоит из одного и того же символа, отсюда $k = 1$, и ответ должен существовать, т.е. так мы придём к противоречию.



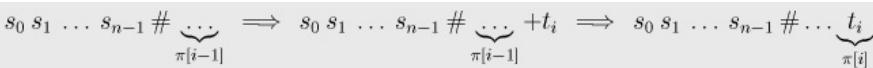
Построение автомата по префикс-функции

Вернёмся к уже неоднократно использованному приёму конкатенации двух строк через разделитель, т.е. для данных строк s и t вычисление префикс-функции для строки $s + \# + t$. Очевидно, что т.к. символ $\#$ является разделителем, то значение префикс-функции никогда не превысит $s.length()$. Отсюда следует, что, как упоминалось при описании алгоритма вычисления префикс-функции, достаточно хранить только строку $s + \#$ и значения префикс-функции для неё, а для всех последующих символов префикс-функцию вычислять на лету:



Действительно, в такой ситуации, зная очередной символ $c \in t$ и значение префикс-функции в предыдущей позиции, можно будет вычислить новое значение префикс-функции, никак при этом не используя все предыдущие символы строки t и значения префикс-функции в них.

Другими словами, мы можем построить **автомат**: состоянием в нём будет текущее значение префикс-функции, переходы из одного состояния в другое будут осуществляться под действием символа:



Таким образом, даже не имея строки t , мы можем предварительно построить такую таблицу переходов $(old_pi, c) \rightarrow new_pi$ с помощью того же алгоритма вычисления префикс-функции:

```
string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function(s);
vector < vector<int> > aut (n, vector<int> (alphabet));
for (int i=0; i<n; ++i)
    for (char c=0; c<alphabet; ++c) {
        int j = i;
        while (j > 0 && c != s[j])
            j = pi[j-1];
        if (c == s[j])  ++j;
        aut[i][c] = j;
    }
```

Правда, в таком виде алгоритм будет работать за $O(n^2k)$ (k — мощность алфавита). Но заметим, что вместо внутреннего цикла `while`, который постепенно укорачивает ответ, мы можем воспользоваться уже вычисленной частью таблицы: переходя от значения j к значению $\pi[j - 1]$, мы фактически говорим, что переход из состояния (j, c) приведёт в то же состояние, что и переход $(\pi[j - 1], c)$, а для него ответ уже точно посчитан (т.к. $\pi[j - 1] < j$):

```

string s; // входная строка
const int alphabet = 256; // мощность алфавита символов, обычно меньше

s += '#';
int n = (int) s.length();
vector<int> pi = prefix_function(s);
vector<vector<int>> aut(n, vector<int>(alphabet));
for (int i=0; i<n; ++i)
    for (char c=0; c<alphabet; ++c)
        if (i > 0 && c != s[i])
            aut[i][c] = aut[pi[i-1]][c];
        else
            aut[i][c] = i + (c == s[i]);

```

В итоге получилась крайне простая реализация построения автомата, работающая за $O(nk)$.

Когда может быть полезен такой автомат? Для начала вспомним, что мы считаем префикс-функцию для строки $s + \# + t$, и её значения обычно используют с единственной целью: найти все вхождения строки s в строку t .

Поэтому самая очевидная польза от построения такого автомата — **ускорение вычисления префикс-функции** для строки $s + \# + t$. Построив по строке $s + \#$ автомат, нам уже больше не нужна ни строка s , ни значения префикс-функции в ней, не нужны и никакие вычисления — все переходы (т.е. то, как будет меняться префикс-функция) уже предпосчитаны в таблице.

Но есть и второе, менее очевидное применение. Это случай, когда строка t **является гигантской строкой, построенной по какому-либо правилу**. Это может быть, например, строка Грея или строка, образованная рекурсивной комбинацией нескольких коротких строк, поданных на вход.

Пусть для определённости мы решаем **такую задачу**: дан номер $k \leq 10^5$ строки Грея, и дана строка s длины $n \leq 10^5$. Требуется посчитать количество вхождений строки s в k -ю строку Грея. Напомним, строки Грея определяются таким образом:

```

g1 = "a"
g2 = "aba"
g3 = "abacaba"
g4 = "abacabadabacaba"
...

```

В таких случаях даже просто построение строки t будет невозможным из-за её астрономической длины (например, k -ая строка Грея имеет длину $2^k - 1$). Тем не менее, мы сможем посчитать значение префикс-функции на конце этой строки, зная значение префикс-функции, которое было перед началом этой строки.

Итак, помимо самого автомата также посчитаем такие величины: $G[i][j]$ — значение автомата, достигаемое после "скармливания" ему строки g_i , если до этого автомат находился в состоянии j . Вторая величина — $K[i][j]$ — количество вхождений строки s в строку g_i , если до "скармливания" этой строки g_i автомат находился в состоянии j . Фактически, $K[i][j]$ — это количество раз, которое автомат принимал значение $s.length()$ за время "скармливания" строки g_i . Понятно, что ответом на задачу будет величина $K[k][0]$.

Как считать эти величины? Во-первых, базовыми значениями являются $G[0][j] = j$, $K[0][j] = 0$. А все последующие значения можно вычислять по предыдущим значениям и используя автомат. Итак, для вычисления этих значений для некоторого i мы вспоминаем, что строка g_i состоит из g_{i-1} плюс i -ый символ алфавита плюс снова g_{i-1} . Тогда после "скармливания" первого куска (g_{i-1}) автомат перейдёт в состояние $G[i-1][j]$, затем после "скармливания" символа char_i он перейдёт в состояние:

```
mid = aut[ G[i-1][j] ][char_i]
```

После этого автомату "скармливается" последний кусок, т.е. g_{i-1} :

```
G[i][j] = G[i-1][mid]
```

Количества $K[i][j]$ легко считаются как сумма количеств по трём кускам g_i : строка g_{i-1} , символ char_i и снова строка g_{i-1} :

```
K[i][j] = K[i-1][j] + (mid == s.length()) + K[i-1][mid]
```

Итак, мы решили задачу для строк Грея, аналогично можно решить целый класс таких задач. Например, точно таким же методом решается **следующая задача**: дана строка s , и образцы t_i , каждый из которых задаётся следующим образом: это строка из обычных символов, среди которых могут встречаться рекурсивные вставки других строк в форме $t_k[\text{cnt}]$, которая означает, что в это место должно быть вставлено cnt экземпляров строки t_k . Пример такой схемы:

```

t1 = "abdeca"
t2 = "abc" + t1[30] + "abd"
t3 = t2[50] + t1[100]
t4 = t2[10] + t3[100]

```

Гарантируется, что это описание не содержит в себе циклических зависимостей. Ограничения таковы, что если явным образом раскрывать рекурсию и находить строки t_i , то их длины могут достигать порядка 100^{100} .

Требуется найти количество вхождений строки s в каждую из строк t_i .

Задача решается так же, построением автомата префикс-функции, затем надо вычислять и добавлять в него переходы по целым строкам t_i . В общем-то, это просто более общий случай по сравнению с задачей о строках Грея.

Алгоритмы хэширования в задачах на строки

Алгоритмы хэширования строк помогают решить очень много задач. Но у них есть большой недостаток: что чаще всего они не 100%-ны, поскольку есть множество строк, хэши которых совпадают. Другое дело, что в большинстве задач на это можно не обращать внимания, поскольку вероятность совпадения хэшей всё-таки очень мала.

Определение хэша и его вычисление

Один из лучших способов определить хэш-функцию от строки S следующий:

$$h(S) = S[0] + S[1] * P + S[2] * P^2 + S[3] * P^3 + \dots + S[N] * P^N$$

где P - некоторое число.

Разумно выбирать для P простое число, примерно равное количеству символов во входном алфавите. Например, если строки предполагаются состоящими только из маленьких латинских букв, то хорошим выбором будет P = 31. Если буквы могут быть и заглавными, и маленькими, то, например, можно P = 57.

Во всех кусках кода в этой статье будет использоваться P = 31.

Само значение хэша желательно хранить в самом большом числовом типе - int64, он же long long. Очевидно, что при длине строки порядка 20 символов уже будет происходить переполнение значение. Ключевой момент - что мы не обращаем внимание на эти переполнения, как бы беря хэш по модулю 2^{64} .

Пример вычисления хэша, если допустимы только маленькие латинские буквы:

```
const int p = 31;
long long hash = 0, p_pow = 1;
for (size_t i=0; i<s.length(); ++i)
{
    // желательно отнимать 'a' от кода буквы
    // единицу прибавляем, чтобы у строки вида 'AAAAAA' хэш был ненулевой
    hash += (s[i] - 'a' + 1) * p_pow;
    p_pow *= p;
}
```

В большинстве задач имеет смысл сначала вычислить все нужные степени P в каком-либо массиве.

Пример задачи. Поиск одинаковых строк

Уже теперь мы в состоянии эффективно решить такую задачу. Дан список строк S[1..N], каждая длиной не более M символов. Допустим, требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.

Обычной сортировкой строк мы бы получили алгоритм со сложностью O(NM log N), в то время как используя хэши, мы получим O(NM + N log N).

Алгоритм. Посчитаем хэш от каждой строки, и отсортируем строки по этому хэшу.

```
vector<string> s (n);
// ... считывание строк ...

// считаем все степени p, допустим, до 10000 - максимальной длины строк
const int p = 31;
vector<long long> p_pow (10000);
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
    p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех строк
// в массиве храним значение хэша и номер строки в массиве s
vector < pair<long long, int> > hashes (n);
for (int i=0; i<n; ++i)
{
    long long hash = 0;
    for (size_t j=0; j<s[i].length(); ++j)
        hash += (s[i] - 'a' + 1) * p_pow[j];
    hashes[i] = make_pair (hash, i);
}

// сортируем по хэшам
sort (hashes.begin(), hashes.end());

// выводим ответ
for (int i=0, group=0; i<n; ++i)
{
    if (i == 0 || hashes[i].first != hashes[i-1].first)
        cout << "\nGroup " << ++group << ":";
    cout << ' ' << hashes[i].second;
```

Хэш подстроки и его быстрое вычисление

Предположим, нам дана строка S , и даны индексы I и J . Требуется найти хэш от подстроки $S[I..J]$.

По определению имеем:

$$H[I..J] = S[I] + S[I+1] * P + S[I+2] * P^2 + \dots + S[J] * P^{(J-I)}$$

откуда:

$$\begin{aligned} H[I..J] * P[I] &= S[I] * P[I] + \dots + S[J] * P[J], \\ H[I..J] * P[I] &= H[0..J] - H[0..I-1] \end{aligned}$$

Полученное свойство является очень важным.

Действительно, получается, что, зная только хэши от всех префиксов строки S , мы можем за $O(1)$ получить хэш любой подстроки.

Единственная возникающая проблема - это то, что нужно уметь делить на $P[I]$. На самом деле, это не так просто. Поскольку мы вычисляем хэш по модулю 2^{64} , то для деления на $P[I]$ мы должны найти к нему обратный элемент в поле (например, с помощью [Расширенного алгоритма Евклида](#)), и выполнить умножение на этот обратный элемент.

Впрочем, есть и более простой путь. В большинстве случаев, вместо того чтобы делить хэши на степени P , можно, наоборот, умножать их на эти степени.

Допустим, даны два хэша: один умноженный на $P[I]$, а другой - на $P[J]$. Если $I < J$, то умножим первый хэш на $P[J-I]$, иначе же умножим второй хэш на $P[J-I]$. Теперь мы привели хэши к одной степени, и можем их спокойно сравнивать.

Например, код, который вычисляет хэши всех префиксов, а затем за $O(1)$ сравнивает две подстроки:

```
string s; int i1, i2, len; // входные данные

// считаем все степени p
const int p = 31;
vector<long long> p_pow (s.length());
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
    p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех префиксов
vector<long long> h (s.length());
for (size_t i=0; i<s.length(); ++i)
{
    h[i] = (s[i] - 'a' + 1) * p_pow[i];
    if (i) h[i] += h[i-1];
}

// получаем хэши двух подстрок
long long h1 = h[i1+len-1];
if (i1) h1 -= h[i1-1];
long long h2 = h[i2+len-1];
if (i2) h2 -= h[i2-1];

// сравниваем их
if (i1 < i2 && h1 * p_pow[i2-i1] == h2 ||
   i1 > i2 && h1 == h2 * p_pow[i1-i2])
    cout << "equal";
else
    cout << "different";
```

Применение хэширования

Вот некоторые типичные применения хэширования:

- Алгоритм Рабина-Карпа поиска подстроки в строке за $O(N)$
- Определение количества различных подстрок за $O(N^2 \log N)$
- Определение количества палиндромов внутри строки

Алгоритм Рабина-Карпа поиска подстроки в строке за $O(N)$

Этот алгоритм базируется на хэшировании строк, и тех, кто не знаком с темой, отсылаю к "[Алгоритмам хэширования в задачах на строки](#)".

Авторы алгоритма - Рабин (Rabin) и Карп (Karp), 1987 год.

Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T за время O(|S| + |T|).

Алгоритм. Посчитаем хэш для строки S. Посчитаем значения хэшей для всех префиксов строки T. Теперь переберём все подстроки T длины |S| и каждую сравним с |S| за время O(1).

Реализация

```
string s, t; // входные данные

// считаем все степени р
const int p = 31;
vector<long long> p_pow (max (s.length(), t.length()));
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
    p_pow[i] = p_pow[i-1] * p;

// считаем хэши от всех префиксов строки T
vector<long long> h (t.length());
for (size_t i=0; i<t.length(); ++i)
{
    h[i] = (t[i] - 'a' + 1) * p_pow[i];
    if (i) h[i] += h[i-1];
}

// считаем хэш от строки S
long long h_s = 0;
for (size_t i=0; i<s.length(); ++i)
    h_s += (s[i] - 'a' + 1) * p_pow[i];

// перебираем все подстроки T длины |S| и сравниваем их
for (size_t i = 0; i + s.length() - 1 < t.length(); ++i)
{
    long long cur_h = h[i+s.length()-1];
    if (i) cur_h -= h[i-1];
    // приводим хэши к одной степени и сравниваем
    if (cur_h == h_s * p_pow[i])
        cout << i << ' ';
}
```

Определение количества различных подстрок за O(N^2 log N)

Этот алгоритм базируется на хэшировании строк, и тех, кто не знаком с темой, отсылаю к "[Алгоритмам хэширования в задачах на строки](#)".

Пусть дана строка S длиной N, состоящая только из маленьких латинских букв. Требуется найти количество различных подстрок в этой строке.

Алгоритм

Переберём по очереди длину подстроки: L = 1 .. N.

Для каждого L мы построим массив хэшей подстрок длины L, причём приведём хэши к одной степени, и отсортируем этот массив. Количество различных элементов в этом массиве прибавляем к ответу.

Реализация

```
string s; // входная строка
int n = (int) s.length();

// считаем все степени р
const int p = 31;
vector<long long> p_pow (s.length());
p_pow[0] = 1;
for (size_t i=1; i<p_pow.size(); ++i)
    p_pow[i] = p_pow[i-1] * p;
```

```

// считаем хэши от всех префиксов
vector<long long> h (s.length());
for (size_t i=0; i<s.length(); ++i)
{
    h[i] = (s[i] - 'a' + 1) * p_pow[i];
    if (i) h[i] += h[i-1];
}

int result = 0;

// перебираем длину подстроки
for (int l=1; l<n; ++l)
{
    // ищем ответ для текущей длины

    // получаем хэши для всех подстрок длины l
    vector<long long> hs (n-l+1);
    for (int i=0; i<n-l+1; ++i)
    {
        long long cur_h = h[i+l-1];
        if (i) cur_h -= h[i-1];
        // приводим все хэши к одной степени
        cur_h *= p_pow[n-i-1];
        hs[i] = cur_h;
    }

    // считаем количество различных хэшей
    sort (hs.begin(), hs.end());
    hs.erase (unique (hs.begin(), hs.end()), hs.end());
    result += (int) hs.size();
}

cout << result;

```

Разбор выражений. Обратная польская нотация

Дана строка, представляющая собой математическое выражение, содержащее числа, переменные, различные операции. Требуется вычислить его значение за $O(N)$, где N - длина строки.

Здесь описан алгоритм, который переводит это выражение в так называемую **обратную польскую нотацию** (явным или неявным образом), и уже в ней вычисляет выражение.

Обратная польская нотация

Обратная польская нотация - это форма записи математических выражений, в которой операторы расположены после своих operandов.

Например, следующее выражение:

```
a + b * c * d + (e - f) * (g * h + i)
```

в обратной польской нотации записывается следующим образом:

```
a b c * d * + e f - g h * i + * +
```

Обратная польская нотация была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в середине 1950-х на основе польской нотации, которая была предложена в 1920 г. польским математиком Яном Лукасевичем.

Удобство обратной польской нотации заключается в том, что выражения, представленные в такой форме, очень **легко вычислять**, причём за линейное время. Заведём стек, изначально он пуст. Будем двигаться слева направо по выражению в обратной польской нотации; если текущий элемент - число или переменная, то кладём на вершину стека её значение; если же текущий элемент - операция, то достаём из стека два верхних элемента (или один, если операция унарная), применяем к ним операцию, и результат кладём обратно в стек. В конце концов в стеке останется ровно один элемент - значение выражения.

Очевидно, этот простой алгоритм выполняется за $O(N)$, т.е. порядка длины выражения.

Разбор простейших выражений

Пока мы рассматриваем только простейший случай: все операции **бинарны** (т.е. от двух аргументов), и все **левоассоциативны** (т.е. при равенстве приоритетов выполняются слева направо). Скобки разрешены.

Заведём два стека: один для чисел, другой для операций и скобок (т.е. стек символов). Изначально оба стека пусты. Для второго стека будем поддерживать предусловие, что все операции упорядочены в нём по строгому убыванию приоритета, если двигаться от вершины стека. Если в стеке есть открывающие скобки, то упорядочен каждый блок операций, находящийся между скобками, а весь стек в таком случае не обязательно упорядочен.

Будем идти по строке слева направо. Если текущий элемент - цифра или переменная, то положим в стек значение этого числа/переменной. Если текущий элемент - открывающая скобка, то положим её в стек. Если текущий элемент - закрывающая скобка, то будем выталкивать из стека и выполнять все операции до тех пор, пока мы не извлечём открывающую скобку (т.е., иначе говоря, встречаю закрывающую скобку, мы выполняем все операции, находящиеся внутри этой скобки). Наконец, если текущий элемент - операция, то, пока на вершине стека находится операция с таким же или большим приоритетом, будем выталкивать и выполнять её.

После того, как мы обработаем всю строку, в стеке операций ещё могут остаться некоторые операции, которые ещё не были вычислены, и нужно выполнить их все (т.е. действуем аналогично случаю, когда встречаем закрывающую скобку).

Вот реализация данного метода на примере обычных операций $+ * \% / -$:

```
bool delim (char c) {
    return c == ' ';
}

bool is_op (char c) {
    return c=='+' || c=='-' || c=='*' || c=='/' || c=='%';
}

int priority (char op) {
    return
        op == '+' || op == '-' ? 1 :
        op == '*' || op == '/' || op == '%' ? 2 :
        -1;
}

void process_op (vector<int> & st, char op) {
    int r = st.back(); st.pop_back();
    int l = st.back(); st.pop_back();
    switch (op) {
        case '+': st.push_back (l + r); break;
        case '-': st.push_back (l - r); break;
        case '*': st.push_back (l * r); break;
        case '/': st.push_back (l / r); break;
        case '%': st.push_back (l % r); break;
    }
}

int calc (string & s) {
    vector<int> st;
    vector<char> op;
    for (size_t i=0; i<s.length(); ++i)
        if (!delim (s[i]))
            if (s[i] == '(')
                op.push_back ('(');
            else if (s[i] == ')') {
                while (op.back() != '(')
                    process_op (st, op.back()), op.pop_back();
                op.pop_back();
            }
            else if (is_op (s[i])) {
                char curop = s[i];
                while (!op.empty() && priority(op.back()) >= priority(s[i]))
                    process_op (st, op.back()), op.pop_back();
                op.push_back (curop);
            }
            else {
                string operand;
                while (s[i] >= 'a' && s[i] <= 'z' || isdigit (s[i]))
                    operand += s[i++];
                --i;
                if (isdigit (operand[0]))
                    st.push_back (atoi (operand.c_str()));
                else
                    st.push_back (get_variable_val (operand));
            }
        while (!op.empty())
            process_op (st, op.back()), op.pop_back();
    return st.back();
}
```

Таким образом, мы научились вычислять значение выражения за $O(N)$, и при этом мы неявно воспользовались обратнойпольской нотацией: мы расположили операции в таком порядке, когда к моменту вычисления очередной операции оба её операнда уже вычислены. Слегка модифицировав вышеописанный алгоритм, можно получить выражение в обратнойпольскойнотации и в явном виде.

Унарные операции

Теперь предположим, что выражение содержит унарные операции (т.е. от одного аргумента). Например, особенно часто встречаются унарный плюс и минус.

Одно из отличий этого случая заключается в необходимости определения того, является ли текущая операция унарной или бинарной.

Можно заметить, что перед унарной операцией всегда стоит либо другая операция, либо открывающая скобка, либо вообще ничего (если она стоит в самом начале строки). Перед бинарной операцией, напротив, всегда стоит либо операнд (число/переменная), либо закрывающая скобка. Таким образом, достаточно завести какой-нибудь флаг для указания того, может ли следующая операция быть унарной или нет.

Ещё чисто реализационная тонкость - как различать унарные и бинарные операции при извлечении из стека и вычислении. Здесь можно, например, для унарных операций вместо символа $s[i]$ кладь в стек $-s[i]$.

Приоритет для унарных операций нужно выбирать таким, чтобы он был больше приоритетов всех бинарных операций.

Кроме того, надо заметить, что унарные операции фактически являются правоассоциативными - если подряд идут несколько унарных операций, то они должны обрабатываться справа налево (для описания этого случая см. ниже; приведённый здесь код уже учитывает правоассоциативность).

Реализация для бинарных операций $+*/$ и унарных операций $+-$:

```
bool delim (char c) {
    return c == ' ';
}

bool is_op (char c) {
    return c=='+' || c=='-' || c=='*' || c=='/' || c=='%';
}

int priority (char op) {
    if (op < 0)
        return op == '-+' || op == '--' ? 4;
    return
        op == '+' || op == '-' ? 1 :
        op == '*' || op == '/' || op == '%' ? 2 :
        -1;
}

void process_op (vector<int> & st, char op) {
    if (op < 0) {
        int l = st.back(); st.pop_back();
        switch (-op) {
            case '+': st.push_back (l); break;
            case '-': st.push_back (-l); break;
        }
    }
    else {
        int r = st.back(); st.pop_back();
        int l = st.back(); st.pop_back();
        switch (op) {
            case '+': st.push_back (l + r); break;
            case '-': st.push_back (l - r); break;
            case '*': st.push_back (l * r); break;
            case '/': st.push_back (l / r); break;
            case '%': st.push_back (l % r); break;
        }
    }
}

int calc (string & s) {
    bool may_unary = true;
    vector<int> st;
    vector<char> op;
    for (size_t i=0; i<s.length(); ++i)
        if (!delim (s[i]))
            if (s[i] == '(') {
                op.push_back ('(');
                may_unary = true;
            }
            else if (s[i] == ')') {
                while (op.back() != '(')
                    process_op (st, op.back()), op.pop_back();
                op.pop_back();
                may_unary = false;
            }
            else if (is_op (s[i])) {
                char curop = s[i];
                if (may_unary && isunary (curop)) curop = -curop;
                while (!op.empty() && (
                    curop >= 0 && priority(op.back()) >= priority(curop)
                    || curop < 0 && priority(op.back()) > priority(curop)))
                )
                    process_op (st, op.back()), op.pop_back();
                op.push_back (curop);
                may_unary = true;
            }
            else {
                string operand;
                while (s[i] >= 'a' && s[i] <= 'z' || isdigit (s[i]))
                    operand += s[i++];
                --i;
                st.push_back (get_val (operand));
                may_unary = false;
            }
        }
}
```

```

    }
    while (!op.empty())
        process_op (st, op.back()), op.pop_back();
    return st.back();
}

```

Стоит заметить, что в простейших случаях, например, когда из унарных операций разрешены только + и -, правоассоциативность не играет никакой роли, поэтому в таких ситуациях никаких усложнений в схему можно не вводить. Т.е. цикл:

```

while (!op.empty() && (
    curop >= 0 && priority(op.back()) >= priority(curop)
    || curop < 0 && priority(op.back()) > priority(curop))
)
process_op (st, op.back()), op.pop_back();

```

Можно заменить на:

```

while (!op.empty() && priority(op.back()) >= priority(curop))
    process_op (st, op.back()), op.pop_back();

```

Правоассоциативность

Правоассоциативность оператора означает, что при равенстве приоритетов операторы вычисляются справа налево (соответственно, левоассоциативность - когда слева направо).

Как уже было отмечено выше, унарные операторы обычно являются правоассоциативными. Другой пример - обычно операция возведения в степень считается правоассоциативной (действительно, a^b^c обычно воспринимается как $a^{(b^c)}$, а не $(a^b)^c$).

Какие отличия нужно внести в алгоритм, чтобы корректно обрабатывать правоассоциативность? На самом деле, изменения нужны самые минимальные. Единственное отличие будет проявляться только при равенстве приоритетов, и заключается оно в том, что операции с равным приоритетом, находящиеся на вершине стека, не должны выполнять раньше текущей операции.

Таким образом, единственные отличия нужно внести в функцию calc:

```

int calc (string & s) {
...
    while (!op.empty() && (
        left_assoc(curop) && priority(op.back()) >= priority(curop)
        || !left_assoc(curop) && priority(op.back()) > priority(curop)))
...
}

```

Суффиксный массив

Дана строка $s[0 \dots n - 1]$ длины n .

i -ым суффиксом строки называется подстрока $s[i \dots n - 1]$, $i = 0 \dots n - 1$.

Тогда суффиксным массивом строки s называется перестановка индексов суффиксов $p[0 \dots n - 1]$, $p[i] \in [0; n - 1]$, которая задаёт порядок суффиксов в порядке лексикографической сортировки. Иными словами, нужно выполнить сортировку всех суффиксов заданной строки.

Например, для строки $s = abaab$ суффиксный массив будет равен:

(2, 3, 0, 4, 1)

Построение за $O(n \log n)$

Строго говоря, описываемый ниже алгоритм будет выполнять сортировку не суффиксов, а циклических сдвигов строки. Однако из этого алгоритма легко получить и алгоритм сортировки суффиксов: достаточно приписать в конец строки произвольный символ, который заведомо меньше любого символа, из которого может состоять строка (например, это может быть доллар или юань; в языке С в этих целях можно использовать уже имеющийся нулевой символ).

Сразу заметим, что поскольку мы сортируем циклические сдвиги, то и подстроки мы будем рассматривать циклическими: под подстрокой $s[i \dots j]$, когда $i > j$, понимается подстрока $s[i \dots n - 1] + s[0 \dots j]$. Кроме того, предварительно все индексы берутся по модулю длины строки (в целях упрощения формула буду опускать явные взятия индексов по модулю).

Рассматриваемый нами алгоритм состоит из примерно $\log n$ фаз. На k -ой фазе ($k = 0 \dots \lceil \log n \rceil$) сортируются циклические подстроки длины 2^k . На последней, $\lceil \log n \rceil$ -ой фазе, будут сортироваться подстроки длины $2^{\lceil \log n \rceil} > n$, что эквивалентно сортировке циклических сдвигов.

На каждой фазе алгоритм помимо перестановки $p[0 \dots n - 1]$ индексов циклических подстрок будет поддерживать для каждой циклической подстроки, начинающейся в позиции i с длиной 2^k , **номер $c[i]$ класса эквивалентности**, которому эта подстрока принадлежит. В самом деле, среди подстрок могут быть одинаковые, и алгоритму понадобится информация об этом. Кроме того, номера $c[i]$ классов эквивалентности будем давать таким образом, чтобы они сохраняли и информацию о порядке: если один суффикс меньше другого, то и номер класса он должен получить меньший. Классы будем для удобства нумеровать с нуля. Количество классов эквивалентности будем хранить в переменной `classes`.

Приведём **пример**. Рассмотрим строку $s = aaba$. Значения массивов $p[]$ и $c[]$ на каждой стадии с нулевой по вторую таковы:

```
0 : p = (0, 1, 3, 2)  c = (0, 0, 1, 0)
1 : p = (0, 3, 1, 2)  c = (0, 1, 2, 0)
2 : p = (3, 0, 1, 2)  c = (1, 2, 3, 0)
```

Стоит отметить, что в массиве $p[]$ возможны неоднозначности. Например, на нулевой фазе массив мог равняться: $p = (3, 1, 0, 2)$. То, какой именно вариант получится, зависит от конкретной реализации алгоритма, но все варианты одинаково правильны. В то же время, в массиве $c[]$ никаких неоднозначностей быть не могло.

Перейдём теперь к построению **алгоритма**. Входные данные:

```
char *s; // входная строка
int n; // длина строки

// константы
const int maxlen = ...; // максимальная длина строки
const int alphabet = 256; // размер алфавита, <= maxlen
```

На **нулевой фазе** мы должны отсортировать циклические подстроки длины 1 , т.е. отдельные символы строки, и разделить их на классы эквивалентности (просто одинаковые символы должны быть отнесены к одному классу эквивалентности). Это можно сделать trivialно, например, сортировкой подсчётом. Для каждого символа посчитаем, сколько раз он встретился. Потом по этой информации восстановим массив $p[]$. После этого, проходом по массиву $p[]$ и сравнением символов, строится массив $c[]$.

```
int p[maxlen], cnt[maxlen], c[maxlen];
memset (cnt, 0, alphabet * sizeof(int));
for (int i=0; i<n; ++i)
    ++cnt[s[i]];
for (int i=1; i<alphabet; ++i)
    cnt[i] += cnt[i-1];
for (int i=0; i<n; ++i)
    p[--cnt[s[i]]] = i;
c[p[0]] = 0;
int classes = 1;
for (int i=1; i<n; ++i) {
    if (s[p[i]] != s[p[i-1]])  ++classes;
    c[p[i]] = classes-1;
}
```

Далее, пусть мы выполнили $k - 1$ -ю фазу (т.е. вычислили значения массивов $p[]$ и $c[]$ для неё), теперь научимся за $O(n)$ выполнять **следующую, k -ю, фазу**. Поскольку фаз всего $O(\log n)$, это даст нам требуемый алгоритм с временем $O(n \log n)$.

Для этого заметим, что циклическая подстрока длины 2^k состоит из двух подстрок длины 2^{k-1} , которые мы можем сравнивать между собой за $O(1)$, используя информацию с предыдущей фазы — номера $c[]$ классов эквивалентности. Таким образом, для подстроки длины 2^k , начинающейся в позиции i , вся необходимая информация содержится в паре чисел $(c[i], c[i + 2^k])$ (повторимся, мы используем массив $c[]$ с предыдущей фазы).

length = 2^k

$\underbrace{\dots s_i \dots s_{i+2^{k-1}-1}}_{\text{length} = 2^{k-1}, \text{class} = c[i]} \quad \underbrace{s_{i+2^{k-1}} \dots s_{i+2^k-1}}_{\text{length} = 2^{k-1}, \text{class} = c[i+2^k]}$... $\underbrace{\dots s_j \dots s_{j+2^{k-1}-1}}_{\text{length} = 2^{k-1}, \text{class} = c[j]} \quad \underbrace{s_{j+2^{k-1}} \dots s_{j+2^k-1}}_{\text{length} = 2^{k-1}, \text{class} = c[j+2^k-1]} \dots$

Это даёт нам весьма простое решение: **отсортировать** подстроки длины 2^k просто **по этим парам чисел**, это и даст нам требуемый порядок, т.е. массив $p[]$. Однако обычная сортировка, выполняющаяся за время $O(n \log n)$, нас не устроит — это даст алгоритм построения суффиксного массива с временем $O(n \log^2 n)$ (зато этот алгоритм несколько проще в написании, чем описываемый ниже).

Как быстро выполнить такую сортировку пар? Поскольку элементы пар не превосходят n , то можно выполнить сортировку подсчётом. Однако для достижения лучшей скрытой в асимптотике константы вместо сортировки пар придёт к сортировке просто чисел.

Воспользуемся здесь приёмом, на котором основана так называемая **цифровая сортировка**: чтобы отсортировать пары, отсортируем их сначала по вторым элементам, а затем — по первым элементам (но уже обязательно стабильной сортировкой, т.е. не нарушающей относительного порядка элементов при равенстве). Однако отдельно вторые элементы уже упорядочены — этот порядок задан в массиве $p[]$ от предыдущей фазы. Тогда, чтобы упорядочить пары по вторым элементам, надо просто от каждого элемента массива $p[]$ отнять 2^{k-1} — это даст нам порядок сортировки пар по вторым элементам (ведь $p[]$ даёт упорядочение подстрок длины 2^{k-1} , и при переходе к строке вдвое большей длины эти подстроки становятся их вторыми половинками, поэтому от позиции второй половинки отнимается длина первой половинки).

Таким образом, с помощью всего лишь вычитаний от элементов массива $p[]$ мы производим сортировку по вторым элементам пар. Теперь надо произвести стабильную сортировку по первым элементам пар, её уже можно выполнить за $O(n)$ с помощью сортировки подсчётом.

Осталось только пересчитать номера $c[]$ классов эквивалентности, но их уже легко получить, просто пройдя по полученной новой перестановке $p[]$ и сравнивая соседние элементы (опять же, сравнивая как пары двух чисел).

Приведём **реализацию** выполнения всех фаз алгоритма, кроме нулевой. Вводятся дополнительно временные массивы pn и cn (pn — содержит перестановку в порядке сортировки по вторым элементам пар, cn — новые номера классов эквивалентности).

```

int pn[maxlen], cn[maxlen];
for (int h=0; (1<<h)<n; ++h) {
    for (int i=0; i<n; ++i) {
        pn[i] = p[i] - (1<<h);
        if (pn[i] < 0) pn[i] += n;
    }
    memset (cnt, 0, classes * sizeof(int));
    for (int i=0; i<n; ++i)
        ++cnt[c[pn[i]]];
    for (int i=1; i<classes; ++i)
        cnt[i] += cnt[i-1];
    for (int i=n-1; i>=0; --i)
        p[--cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i=1; i<n; ++i) {
        int mid1 = (p[i] + (1<<h)) % n, mid2 = (p[i-1] + (1<<h)) % n;
        if (c[p[i]] != c[p[i-1]] || c[mid1] != c[mid2])
            ++classes;
        cn[p[i]] = classes-1;
    }
    memcpy (c, cn, n * sizeof(int));
}

```

Этот алгоритм требует $O(n \log n)$ времени и $O(n)$ памяти. Впрочем, если учитывать ещё размер k алфавита, то время работы становится $O((n+k) \log n)$, а размер памяти — $O(n+k)$.

Применения

Нахождение наименьшего циклического сдвига строки

Вышеописанный алгоритм производит сортировку циклических сдвигов (если к строке не приписывать доллар), а потому $p[0]$ даст искомую позицию наименьшего циклического сдвига. Время работы — $O(n \log n)$.

Поиск подстроки в строке

Пусть требуется в тексте t искать строку s в режиме онлайн (т.е. заранее строку s нужно считать неизвестной). Построим суффиксный массив для текста t за $O(|t| \log |t|)$. Теперь подстроку s будем искать следующим образом: заметим, что искомое вхождение должно быть префиксом какого-либо суффикса t . Поскольку суффиксы у нас упорядочены (это даёт нам суффиксный массив), то подстроку s можно искать бинарным поиском по суффиксам строки. Сравнение текущего суффикса и подстроки s внутри бинарного поиска можно производить тривиально, за $O(|p|)$. Тогда асимптотика поиска подстроки в тексте становится $O(|p| \log |t|)$.

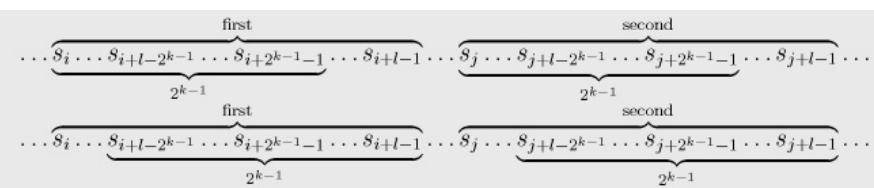
Сравнение двух подстрок строк

Требуется по заданной строке s , произведя некоторый её препроцессинг, научиться за $O(1)$ отвечать на запросы сравнения двух произвольных подстрок (т.е. проверка, что первая подстрока равна/меньше/больше второй).

Построим суффиксный массив за $O(|s| \log |s|)$, при этом сохраним промежуточные результаты: нам понадобятся массивы $c[]$ от каждой фазы. Поэтому памяти потребуется тоже $O(|s| \log |s|)$.

Используя эту информацию, мы можем за $O(1)$ сравнивать любые две подстроки длины, равной степени двойки: для этого достаточно сравнить номера классов эквивалентности из соответствующей фазы. Теперь надо обобщить этот способ на подстроки произвольной длины.

Пусть теперь поступил очередной запрос сравнения двух подстрок длины l с началами в индексах i и j . Найдём наибольшую длину блока, помещающегося внутри подстроки такой длины, т.е. наибольшее k такое, что $2^k \leq l$. Тогда сравнение двух подстрок можно заменить сравнением двух пар перекрывающихся блоков длины 2^k : сначала надо сравнить два блока, начинающихся в позициях i и j , а при равенстве — сравнить два блока, заканчивающихся в позициях $i+l-1$ и $j+l-1$:



Таким образом, реализация получается примерно такой (здесь считается, что вызывающая процедура сама вычисляет k , поскольку сделать это за константное время не так легко (по-видимому, быстрее всего — предпосчётом), но в любом случае это не имеет отношения к применению суффиксного массива):

```

int compare (int i, int j, int l, int k) {
    pair<int,int> a = make_pair (c[k][i], c[k][i+l-(1<<(k-1))]);
    pair<int,int> b = make_pair (c[k][j], c[k][j+l-(1<<(k-1))]);
    return a == b ? 0 : a < b ? -1 : 1;
}

```

Наибольший общий префикс двух подстрок: способ с дополнительной памятью

Требуется по заданной строке s , произведя некоторый её препроцессинг, научиться за $O(1)$ отвечать на запросы наибольшего общего префикса (longest common prefix, lcp) для двух произвольных суффиксов с позициями i и j .

Способ, описываемый здесь, требует $O(|s| \log |s|)$ дополнительной памяти; другой способ, использующий линейный объём памяти, но неконстантное время ответа на запрос, описан в следующем разделе.

Построим суффиксный массив за $O(|s| \log |s|)$, при этом сохраним промежуточные результаты: нам понадобятся массивы $c[]$ от каждой фазы. Поэтому памяти потребуется тоже $O(|s| \log |s|)$.

Пусть теперь поступил очередной запрос: пара индексов i и j . Воспользуемся тем, что мы можем за $O(1)$ сравнивать любые две подстроки длины, являющейся степенью двойки. Для этого будем перебирать степень двойки (от большей к меньшей), и для текущей степени проверять: если подстроки такой длины совпадают, то к ответу прибавить эту степень двойки, а наибольший общий префикс продолжим искать справа от одинаковой части, т.е. к i и j надо прибавить текущую степень двойки.

Реализация:

```
int lcp (int i, int j) {
    int ans = 0;
    for (int k=log_n; k>=0; --k)
        if (c[k][i] == c[k][j]) {
            ans += 1<<k;
            i += 1<<k;
            j += 1<<k;
        }
    return ans;
}
```

Здесь через \log_n обозначена константа, равная логарифму n по основанию 2, округлённому вниз.

Наибольший общий префикс двух подстрок: способ без дополнительной памяти. Наибольший общий префикс двух соседних суффиксов

Требуется по заданной строке s , произведя некоторый её препроцессинг, научиться отвечать на запросы наибольшего общего префикса (longest common prefix, lcp) для двух произвольных суффиксов с позициями i и j .

В отличие от предыдущего метода, описываемый здесь будет выполнять препроцессинг строки за $O(n \log n)$ времени с $O(n)$ памяти.

Результатом этого препроцессинга будет являться массив (который сам по себе является важным источником информации о строке, и потому использоваться для решения других задач). Ответы же на запрос будут производиться как результат выполнения запроса RMQ (минимум на отрезке, range minimum query) в этом массиве, поэтому при разных реализациях можно получить как логарифмическое, так и константное времена работы.

Базой для этого алгоритма является следующая идея: найдём каким-нибудь образом наибольшие общие префиксы для каждой **соседней в порядке сортировки пары суффиксов**. Иными словами, построим массив $lcp[0 \dots n - 2]$, где $lcp[i]$ равен наибольшему общему префиксу суффиксов $p[i]$ и $p[i + 1]$. Этот массив даст нам ответ для любых двух соседних суффиксов строки. Тогда ответ для любых двух суффиксов, не обязательно соседних, можно получить по этому массиву. В самом деле, пусть поступил запрос с некоторыми номерами суффиксов i и j . Найдём эти индексы в суффиксном массиве, т.е. пусть k_1 и k_2 — их позиции в массиве $p[]$ (упорядочим их, т.е. пусть $k_1 < k_2$). Тогда ответом на данный запрос будет минимум в массиве lcp , взятый на отрезке $[k_1; k_2 - 1]$. В самом деле, переход от суффикса i к суффиксу j можно заменить целой цепочкой переходов, начинающейся с суффикса i и заканчивающейся в суффиксе j , но включающей в себя все промежуточные суффиксы, находящиеся в порядке сортировки между ними.

Таким образом, если мы имеем такой массив lcp , то ответ на любой запрос наибольшего общего префикса сводится к запросу **минимума на отрезке** массива lcp . Эта классическая задача минимума на отрезке (range minimum query, RMQ) имеет множество решений с различными асимптотиками, описанные [здесь](#).

Итак, основная наша задача — **построение** этого массива lcp . Строить его мы будем по ходу алгоритма построения суффиксного массива: на каждой текущей итерации будем строить массив lcp для циклических подстрок текущей длины.

После нулевой итерации массив lcp , очевидно, должен быть нулевым.

Пусть теперь мы выполнили $k - 1$ -ю итерацию, получили от неё массив lcp' , и должны на текущей k -й итерации пересчитать этот массив, получив новое его значение lcp . Как мы помним, в алгоритме построения суффиксного массива циклические подстроки длины 2^k разбивались пополам на две подстроки длины 2^{k-1} ; воспользуемся этим же приёмом и для построения массива lcp .

Итак, пусть на текущей итерации алгоритм вычисления суффиксного массива выполнил свою работу, нашёл новое значение перестановки $p[]$ подстрок. Будем теперь идти по этому массиву и смотреть пары соседних подстрок: $p[i]$ и $p[i + 1]$, $i = 0 \dots n - 2$. Разбивая каждую подстроку пополам, мы получаем две различных ситуации: 1) первые половинки подстрок в позициях $p[i]$ и $p[i + 1]$ различаются, и 2) первые половинки совпадают (напомним, такое сравнение можно легко производить, просто сравнивая номера классов $c[]$ с предыдущей итерацией). Рассмотрим каждый из этих случаев отдельно.

1) Первые половинки подстрок различались. Заметим, что тогда на предыдущем шаге эти первые половинки необходимо были соседними. В самом деле, классы эквивалентности не могли исчезать (а могут только появляться), поэтому все различные подстроки длины 2^{k-1} дадут (в качестве первых половинок) на текущей итерации различные подстроки длины 2^k , и в том же порядке. Таким образом, для определения $lcp[i]$ в этом случае надо просто взять соответствующее значение из массива lcp' .

2) Первые половинки совпадали. Тогда вторые половинки могли как совпадать, так и различаться; при этом, если они различаются, то они совсем не обязательно должны были быть соседними на предыдущей итерации. Поэтому в этом случае нет простого способа определить $lcp[i]$. Для его определения надо поступить так же, как мы собираемся потом вычислять наибольший общий префикс для любых двух суффиксов: надо выполнить запрос минимума (RMQ) на соответствующем отрезке массива lcp' .

Оценим **асимптотику** такого алгоритма. Как мы видели при разборе этих двух случаев, только второй случай даёт увеличение числа классов эквивалентности. Иными словами, можно говорить о том, что каждый новый класс эквивалентности появляется вместе с одним запросом RMQ. Поскольку всего классов эквивалентности может быть до n , то и искать минимум мы должны за асимптотику $O(\log n)$. А для этого надо использовать уже какую-то структуру данных для минимума на отрезке; эту структуру данных надо будет строить заново на каждой итерации (которых всего $O(\log n)$). Хорошим вариантом структуры данных будет **Дерево отрезков**: его можно построить за $O(n)$, а потом выполнять запросы за $O(\log n)$, что как раз и даёт нам итоговую асимптотику $O(n \log n)$.

Реализация:

```

int lcp[maxlen], lcpn[maxlen], lpos[maxlen], rpos[maxlen];
memset(lcp, 0, sizeof lcp);
for (int h=0; (1<<h)<n; ++h) {
    for (int i=0; i<n; ++i)
        rpos[p[i]] = i;
    for (int i=n-1; i>=0; --i)
        lpos[p[i]] = i;
    ...
    ... все действия по построению суфф. массива, кроме последней строки (memcpy) ...
}

rmq_build (lcp, n-1);
for (int i=0; i<n-1; ++i) {
    int a = p[i], b = p[i+1];
    if (c[a] != c[b])
        lcpn[i] = lcp[rpos[c[a]]];
    else {
        int aa = (a + (1<<h)) % n, bb = (b + (1<<h)) % n;
        lcpn[i] = (1<<h) + rmq (lpos[c[aa]], rpos[c[bb]]-1);
        lcpn[i] = min (n, lcpn[i]);
    }
}
memcpy (lcp, lcpn, (n-1) * sizeof(int));
memcpy (c, cn, n * sizeof(int));
}

```

Здесь помимо массива `lcp` вводится временный массив `lcpn` с его новым значением. Также поддерживается массив `pos`, который для каждой подстроки хранит её позицию в перестановке `p`. Функция `rmq_build` — некоторая функция, строящая структуру данных для минимума по массиву-первому аргументу, размер его передаётся вторым аргументом. Функция `rmq` возвращает минимум на отрезке: с первого аргумента по второй включительно.

Из самого алгоритма построения суффиксного массива пришлось только вынести копирование массива `c`, поскольку во время вычисления `lcp` нам понадобятся старые значения этого массива.

Стоит отметить, что наша реализация находит длину общего префикса для **циклических подстрок**, в то время как на практике чаще бывает нужной длина общего префикса для суффиксов в их обычном понимании. В этом случае надо просто ограничить значения `lcp` по окончании работы алгоритма:

```

for (int i=0; i<n-1; ++i)
    lcp[i] = min (lcp[i], min (n-p[i], n-p[i+1]));

```

Для любых двух суффиксов длину их наибольшего общего префикса теперь можно найти как минимум на соответствующем отрезке массива `lcp`:

```

for (int i=0; i<n; ++i)
    pos[p[i]] = i;
rmq_build (lcp, n-1);

... поступил запрос (i,j) на нахождение LCP ...
int result = rmq (min(i,j), max(i,j)-1);

```

Количество различных подстрок

Выполним **препроцессинг**, описанный в предыдущем разделе: за $O(n \log n)$ времени и $O(n)$ памяти мы для каждой пары соседних в порядке сортировки суффиксов найдём длину их наибольшего общего префикса. Найдём теперь по этой информации количество различных подстрок в строке.

Для этого будем рассматривать, какие новые подстроки начинаются в позиции `p[0]`, затем в позиции `p[1]`, и т.д. Фактически, мы берём очередной в порядке сортировки суффикс и смотрим, какие его префиксы дают новые подстроки. Тем самым мы, очевидно, не упустим из виду никакие из подстрок.

Пользуясь тем, что суффиксы у нас уже отсортированы, нетрудно понять, что текущий суффикс `p[i]` даст в качестве новых подстрок все свои префиксы, кроме совпадающих с префиксами суффикса `p[i - 1]`. Т.е. все его префиксы, кроме `lcp[i - 1]` первых, дадут новые подстроки. Поскольку длина текущего суффикса равна $n - p[i]$, то окончательно получаем, что текущий суффикс `p[i]` даёт $n - p[i] - lcp[i - 1]$ новых подстрок. Суммируя это по всем суффиксам (для самого первого, `p[0]`, отнимать нечего — прибавится просто $n - p[0]$), получаем **ответ** на задачу:

$$\sum_{i=0}^n (n - p[i]) - \sum_{i=0}^{n-1} lcp[i]$$

Суффиксный автомат

Формальное определение. **Суффиксным автоматом** (suffix automaton) строки S называется минимальный детерминированный автомат, который распознаёт все суффиксы строки S , и только их. (суффиксами также считаются вся строка, а также пустая строка). В англоязычной литературе суффиксный автомат называется suffix automaton (во множественном числе *automata*), а также DAWG (directed acyclic word graph).

Расшифруем это определение. Суффиксный автомат M имеет некоторое конечное множество состояний, среди которых выделено начальное состояние $Init$, а также для всех состояний указано, являются ли они терминальными. Между некоторыми состояниями установлены переходы по символам из заданного алфавита (то, что автомат детерминированный, означает, что из каждого состояния по каждому символу есть не более одного перехода): если мы находимся в некотором состоянии и на вход поступает какой-то символ, то мы переходим по соответствующему переходу (т.е. по этому самому символу) в новое состояние. Говорят, что автомат "распознаёт строку", если, стартуя из начального состояния $Init$, выполнив последовательно переходы по символам этой строки, мы придём в терминальное состояние (если в какой-то момент мы попытались пройти по несуществующему переходу, то мы остановливаемся, и автомат такую строку не распознаёт). Соответственно, суффиксным автоматом будет называться только тот автомат, который распознаёт все суффиксы строки, а любые другие строки распознавать не будет.

Условимся говорить, что некоторой строке T **соответствует** состояние r , если, стартовав в состоянии $Init$ и выполнив все переходы по символам этой строки T , мы придём в состояние r . Например, можно говорить, что любому суффиксу строки S (по которой построен автомат) соответствует терминальное состояние, а любой другой строке - соответствует нетерминальное состояние, или вовсе никакое состояние не соответствует (если в какой-то момент мы попытались совершить несуществующий переход).

Итак, пусть дана строка S длины N в некотором алфавите размера K . Требуется построить суффиксный автомат для неё.

Примеры

Для развития интуиции приведём несколько примеров суффиксных автоматов.

Состояния будем обозначать числами (за исключением начального состояния $Init$), терминальные состояния - отмечать звёздочками, переходы - стрелочками с подписанными поверх них буквами.

```
S = ""
Init*  
  
S = "a"
Init* -a- 1*  
  
S = "aa"
Init* -a- 1* -a- 2*  
  
S = "ab"
Init* -a- 1 -b- 2*
 \           /
 \-----b----/  
  
S = "aaa"
Init* -a- 1* -a- 2* -a- 3*  
  
S = "aba"
Init* -a- 1* -b- 2 -a- 3*
 \           /
 \-----b----/  
  
S = "abb"
Init* -a- 1 -b- 2 -b- 3*
 \           /
 \--b-- 4* -----b----/  
  
S = "abbb"
Init* -a- 1 -b- 2 -b- 3*
 \           /
 \--b-- 4* -b- 5* -b-/
```

Анализ суффиксного автомата

Проведём сначала **анализ** суффиксного автомата.

Во-первых, в нём **не может быть циклов**, потому что в противном случае автомат бы распознавал слова бесконечной длины, которые никак не могут быть суффиксами строки S .

Во-вторых, **число состояний в нём не более $2N-1$** (за исключением $N=1$ и $N=2$, для них будет 1 и 3 состояния соответственно); этот факт мы пока примем без доказательства, он будет вытекать из корректности описанного ниже алгоритма построения суффиксного автомата.

В-третьих, **число переходов - не более $3N-4$** (за исключением, опять же, $N=1$ и $N=2$); доказательство этого факта мы также приведём после описания самого алгоритма.

Таким образом, мы обнаружили удивительный факт: **суффиксный автомат имеет размер $O(N)$** , несмотря на то, что суммарная длина всех суффиксов строки есть величина $O(N^2)$. Более того, ниже будет описан алгоритм, строящий суффиксный автомат также за линейное время (правда, с потреблением памяти $O(NK)$; если же необходимо экономно использовать память - $O(N)$, то время работы алгоритма увеличивается до $O(N \log K)$; более подробно см. в следующем разделе).

Рассмотрим теперь **наихудшие тесты** для суффиксного автомата. Несложно заметить, что для строки вида "abbb..." будет достигаться наибольшее число состояний. Так же непосредственной проверкой можно убедиться в том, что строка вида "abbb...bbc" является наихудшей с точки зрения количества переходов.

Построение суффиксного автомата

Для описания алгоритма нам потребуется ввести ещё две дополнительных характеристики для каждого состояния: `length` и `link`.

Для некоторого состояния `p` величина `length` равна длине наилдлиннейшего пути из `Init` в `p`. Иными словами, `length` - это длина наилдлиннейшего слова, под воздействием которого автомат из состояния `Init` переходит в состояние `p`.

Суффиксная ссылка `link` для состояния `p` определяется следующим образом. Рассмотрим наилдлиннейший путь из состояния `Init` в `p`, ему соответствует некоторая строка. Суффиксная ссылка не определена только для начального состояния `Init`. Найдём такой наибольший суффикс этой строки, что ему (этому суффиксу) соответствует некоторое состояние `q`, отличное от `p`. Из самого определения следует, что суффикс может быть только собственным, а `length[p] > length[q]`.

Сразу же определим понятие **суффиксного пути** для некоторого состояния `p`. Суффиксный путь - это последовательность $(p, link[p], link[link[p]], \dots)$, конечная (поскольку длины `length` вдоль пути строго убывают, и при этом они всегда неотрицательны). Легко понять, что суффиксный путь всегда завершается состоянием `Init`. Если для некоторого состояния `p` мы будем двигаться по его суффиксному пути, то мы будем фактически двигаться по суффиксам строки, соответствующей состоянию `p`.

Теперь опишем сам **алгоритм**. Он будет итеративным (в режиме on-line), т.е. сначала создаст автомат для пустой строки, а потом будет последовательно добавлять символы строки `S`, перестраивая автомат. Единственное исключение - что метки "терминальный" будут расставляться за $O(N)$ уже после построения автомата для всей строки.

Введём сразу тип данных "состояние автомата", где таблицу переходов представим для каждого состояния в виде массива величиной `K`. Состояния будем хранить для удобства в виде статического массива размером `MAXLEN*2-1` (где `MAXLEN` - заранее известная константа - ограничение на длину строки), хотя, понятно, легко будет исправить весь код на использование динамического `vector`.

```
const int MAXLEN = ...; // максимальная длина строки
const int K = ...; // размер алфавита
struct state {
    int length, link;
    int next[K];
    bool terminal;
};
state st[MAXLEN*2-1];
int sz = 0;
int last; // указывает на "последнее" состояние, т.е. то, в которое ведёт самый длинный путь из Init
```

Итак, сначала создадим суффиксный автомат для пустой строки, он состоит только из начального состояния `Init` (у него будет номер 0 в массиве `st`):

```
st[0].length = 0;
st[0].link = -1;
memset (st[0].next, -1, sizeof st[0].next); // делаем все next == -1
++sz;
last = 0;
```

Итак, пусть теперь автомат уже построен для некоторой строки `W`, и мы хотим перестроить его для строки `S = W+C`, где `C` - некоторый символ. Реализуем эту операцию в виде функции `sa_extend (char c)`.

Создадим сразу новое состояние `nlast`, которому по окончании работы `sa_extend` будет соответствовать строка `S`. Пока что у этого состояния мы знаем только длину `length` и то, что переходов из него не будет:

```
void sa_extend (char c) {
    int nlast = sz++;
    st[nlast].length = st[last].length + 1;
    memset (st[nlast].next, -1, sizeof st[nlast].next);
```

После этого мы хотим рассмотреть строку `W` и все её суффиксы, и дописать к ним символ `C`, т.е. добавить переходы по символу `C` в состояние `nlast`. Вспоминая понятие суффиксного пути, мы можем это сформулировать кратко: пройдёмся по суффиксному пути состояния `last` и добавим из каждого посещённого состояния переход по символу `C` в состояние `nlast`. Однако, если в какой-то момент мы столкнёмся с тем, что такой переход уже существует - мы вынуждены будем остановиться; этот случай мы рассмотрим ниже.

```
int p = last;
for ( ; p!=-1 && st[p].next[c]==-1; p=st[p].link)
    st[p].next[c] = nlast;
```

Теперь у нас есть два случая: 1) если мы ни разу не столкнулись с уже существующим переходом, тогда `p== -1`, и 2) если остановились на таком переходе.

Случай 1) - самый простой. Он означает, что символ `C` ещё ни разу не встретился в строке `W`. Мы добавили переходы из каждого суффикса строки `W` по символу `C` в строку `S`, больше нам никаких переходов добавлять не нужно. Осталось только установить суффиксную ссылку для состояния `nlast`, но она будет указывать, очевидно, на `Init` (это как раз следует из того, что символ `C` встретился нам впервые). Итак, мы можем реализовать этот случай:

```
if (p == -1)
    st[nlast].link = 0;
else {
```

Случай 2) распадается также на два случая 2а и 2б. Итак, мы остановились на состоянии `p`, из которого уже есть переход по символу `C`, обозначим через `q` состояние, в которое ведёт этот переход. Если `length[p] + 1 == length[q]`, то это случай 2а, иначе случай 2б (сейчас мы поймём смысл этого равенства).

Случай 2а), т.е. `length[p] + 1 == length[q]` (в этом случае говорят, что переход из состояния `p` в состояние `q` "сплошной" (*solid*)). Обозначим через `U` длинейшую строку, соответствующую состоянию `p` (т.е. `U.length() == length[p]`). Стока `U+C` является наилдлиннейшим суффиксом строки `S = W+C`, встречающимся в `W` (иное и невозможно, поскольку тогда бы мы раньше обнаружили существующий переход по символу `C`), а потому мы должны провести суффиксную ссылку в то состояние, которому соответствует строка `U+A` и при этом является наилдлиннейшей для него. Вспоминая, что `length[q] == length[p] + 1 == (U+C).length()`, мы как раз и получаем, что строка `U+C` является наилдлиннейшей для состояния `q`, и мы можем провести суффиксную ссылку из состояния `nlast` в состояние `q`. Наконец, можно утверждать, что больше никаких переходов добавлять не надо - строка `U+C` вместе со всеми своими суффиксами уже была обработана ранее (ведь она уже

присутствовала в W).

```
int q = st[p].next[c];
if (st[p].length + 1 == st[q].length)
    st[nlast].link = q;
else {
```

Случай 2б), т.е. $\text{length}[p] + 1 \neq \text{length}[q]$ (из вышеописанных свойств можно утверждать, что $\text{length}[p] + 1 < \text{length}[q]$). Снова, как и в случае 2а), обозначим через U длиннейшую строку для состояния p. Снова мы утверждаем, что строка U+C является наидлиннейшим суффиксом строки S, встречающимся в W, а потому суффиксную ссылку из состояния nlast нам хотелось бы провести в состояние, где эта строка U+C является длиннейшей. Однако на этот раз такого состояния не существует: $\text{length}[q] > \text{length}[p] + 1 = (\text{U}+\text{C}).\text{length}()$, и при этом самой строке U+C соответствует состояние q. Тут нам приходится выполнить более хитрое преобразование автомата - мы вынуждены расщепить состояние q на два состояния, для одного из которых строка U+C будет длиннейшей. Создадим новое состояние clone, и перенаправим в него часть переходов в q. Нам нужно перенаправить переходы, соответствующие строке U+C и всем её суффиксам (точнее говоря, всем суффиксам строки U и плюс символ C). Вспоминая, что строке U как раз соответствует состояние p, и вспоминая понятие суффиксного пути, мы получаем такую формулировку: продолжим двигаться по суффиксному пути состояния p, и, пока есть переход из него в состояние q по символу C, будем перенаправлять этот переход на состояние clone. Теперь разберёмся с новым состоянием clone, нам же нужно заполнить его параметры. Его длина length равна длине строки U+C, т.е. $\text{length}[p] + 1$. Его суффиксная ссылка ведёт туда, куда вела суффиксная ссылка состояния q. Наконец, переходы из clone надо скопировать из состояния q (отсюда и название состояния - "clone"). Осталось только заметить, что суффиксная ссылка для состояния q теперь изменится - она будет указывать, разумеется, на состояния clone (ведь мы отдалили строки U+C и короче неё, вот на них теперь и будет указывать суффиксная ссылка). Ну и, разумеется, суффиксная ссылка для состояния nlast будет также указывать на clone (для чего собственно и выполнялось это расщепление).

```
int clone = sz++;
st[clone].length = st[p].length + 1;
memcpy (st[clone].next, st[q].next, sizeof st[clone].next);
st[clone].link = st[q].link;
for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
    st[p].next[c] = clone;
st[q].link = st[nlast].link = clone;
}
```

Наконец, завершаем функцию sa_extend, не забывая обновить значение last:

```
last = nlast;
}
```

Теперь осталось только научиться **проставлять значения terminal** (правда, при решении большинства задач эти значения просто не нужны). Это очень легко сделать: рассмотрим состояние last и его суффиксный путь. Этим состояниям как раз и соответствуют строки S и все её суффиксы, причём только они (это следует непосредственно из определения суффиксной ссылки):

```
for (int p=last; p!=-1; p=st[p].link)
    st[p].terminal = true;
```

Анализ алгоритма

Приведём сначала обещанные **доказательства** того, что состояний в суффиксном автомате не более $2N-1$, а переходов - $3N-4$ (для $N \geq 3$).

Оценка на число состояний непосредственно вытекает из корректности алгоритма - мы добавляем одно состояние Init при создании, и добавляем на каждом шаге по одному или двум состояниям; при этом сразу два состояния могут быть добавлены только начиная с третьего шага; итого и получается $1 + N + N-2 = 2N-1$.

Оценим теперь максимальное число переходов. Рассмотрим оставное дерево из длиннейших путей в автоматае, начинающихся в вершине Init. Оно будет содержать только сплошные рёбра (причём всех их), и их будет на единицу меньше, чем число состояний. Теперь оценим число несплошных рёбер. Поставим каждому несплошному ребру из p в q по символу C такую строку: U+C+V, где U - строка, соответствующая длиннейшему пути из Init в p, V - строка, соответствующая длиннейшему пути из q в некоторое терминальное состояние. Заметим, что разным несплошным рёбрам будут соответствовать разные строки U+C+V. С другой стороны, в силу самого построения этой строки, эта строка U+C+V будет суффиксом строки S. Поскольку число различных суффиксов есть $N+1$, а суффиксы S и "" не могли быть учтены (потому что они входят в оставное дерево), то в итоге несплошных рёбер не более $N-1$. Складывая количества сплошных и несплошных рёбер, получаем оценку $3N-4$.

Оценим теперь **асимптотику** алгоритма. Нам нужно оценить суммарное время работы двух циклов for внутри функции sa_extend. С первым циклом всё понятно - на каждой своей итерации он добавляет новый переход, а т.к. число переходов есть $O(N)$, а переходы никогда не удаляются, то суммарное время работы первого цикла есть $O(N)$. Второй цикл так оценить не получится, поскольку он новых переходов не добавляет. Обозначим через V наимдлиннейшую строку, соответствующую состоянию p. Перед выполнением итерации этого цикла V является K-ым ($K \geq 2$) элементом в суффиксном пути W, а после него V+C является 2-ым элементом в суффиксном пути W+C, и потому позиция V как суффикса W строго увеличивается на каждой итерации этого цикла, причём не только в пределах одного вызова функции sa_extend, а в пределах всех вызовов. Потому число выполнений этого цикла также равно $O(N)$.

Итак, если переходы пех хранить в виде массива, то получаем асимптотику алгоритма **O(NK)** (хотя и с очень малой константой, т.к. все действия, за исключением копирования массивов memset, выполняются за $O(N)$), но при использовании памяти **O(NK)**.

Впрочем, на массивах можно достичь и настоящей линейной асимптотики, если вдобавок к массивам хранить списки переходов (в виде $\text{vector<} \text{pair<} \text{char}, \text{int} \text{>} \text{}$). Тогда копирование переходов будет осуществляться за $O()$ от количества переходов из текущего состояния, а не за $O(K)$.

Однако, если хранить переходы скжато, например, в виде $\text{map<} \text{char}, \text{int} \text{>} \text{}$, то время работы алгоритма увеличится до **O(N log K)**, но зато уменьшится используемая память до **O(N)**.

Реализация

Поскольку весь приведённый выше код равномерно "размазан" по тексту, приведём здесь полную реализацию (работающую за **O(NK)**) (как уже говорилось, на практике - близко к $O(N)$, поскольку $O(NK)$ операций связаны с копированием массивов вызовами memset, которые выполняются очень быстро) при потреблении памяти **O(NK)**:

```

const int MAXLEN = ...; // максимальная длина строки
const int K = ...; // размер алфавита
struct state {
    int length, link;
    int next[K];
    bool terminal;
};
state st[MAXLEN*2-1];
int sz, last;

void init() {
/*
// эти действия нужны, если автомат строится несколько раз для разных строк:
sz = last = 0;
for (int i=0; i<MAXLEN*2-1; ++i)
    st[i].terminal = false;
st[0].length = 0;
*/
st[0].link = -1;
memset (st[0].next, -1, sizeof st[0].next);
++sz;
}

void sa_extend (char c) {
    int nlast = sz++;
    st[nlast].length = st[last].length + 1;
    memset (st[nlast].next, -1, sizeof st[nlast].next);
    int p;
    for (p=last; p!=-1 && st[p].next[c]==-1; p=st[p].link)
        st[p].next[c] = nlast;
    if (p == -1)
        st[nlast].link = 0;
    else {
        int q = st[p].next[c];
        if (st[p].length + 1 == st[q].length)
            st[nlast].link = q;
        else {
            int clone = sz++;
            st[clone].length = st[p].length + 1;
            memcpy (st[clone].next, st[q].next, sizeof st[clone].next);
            st[clone].link = st[q].link;
            for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
                st[p].next[c] = clone;
            st[q].link = st[nlast].link = clone;
        }
    }
    last = nlast;
}

void mark_terminal() {
    for (int p=last; p!=-1; p=st[p].link)
        st[p].terminal = true;
}

```

Чтобы уменьшить потребление памяти до $O(N)$, но ценой увеличения времени работы до $O(N \log K)$, достаточно небольших изменений в этом коде. Массив `int next[K]` надо заменить на `map<char,int> next`, и соответствующим образом изменить весь код: убрать `memset`'ы, выражение `st[p].next[c]==-1` заменить на `!st[p].next.count(c)`, а выражение `st[p].next[c]==q` заменить на `st[p].next.count(c) && st[p].next[c]==q`.

Применения

Пусть дана строка S , для которой построен суффиксный автомат за ту или иную асимптотику.

Тогда мы можем решить следующие задачи (в их асимптотики построение автомата, конечно, не включено):

- Данна строка P , требуется за $O(|P|)$ проверить, **входит ли** она в текст S .
Это сделать очень легко: стартуем из начального состояния `Init`, и пройдём по строке P , выполняя соответствующие переходы по автоматау. Если в какой-то момент мы не смогли сделать переход (по той причине, что его не существовало), то строка P в тексте не входит; иначе же P содержится в S .
- Данна строка P , требуется за $O(|P|)$ вывести позицию её **первого вхождения** в текст S .
Чтобы решить эту задачу, понадобится добавить ещё одно информационное поле в суффиксный автомат: `endpos`. Для каждого состояния r `endpos[r]` будет равно позиции, в которой оканчиваются первые вхождения всех строк, соответствующих r . Здесь имеется в виду, что одному состоянию может соответствовать несколько строк, однако, можно доказать, что среди них найдётся одна такая, что все остальные будут её суффиксами. Тогда ясно, почему для каждого состояния мы не можем посчитать позицию начала первого вхождения - её просто невозможно определить, она будет различной для разных строк. А вот позицию окончания мы можем определить (как раз на основе того, что все строки являются суффиксами одной строки).
Теперь, как же вычислять `endpos` при построении автомата? Для добавляемого состояния `nlast` `endpos` будет равен длине текущей строки - т.е. `endpos[nlast] = length[nlast] - 1` (если в 0-индексации). При расщеплении для `clone` значение `endpos` будет равно `endpos[q]` (опять же, оправдывая название "clone").
Теперь вернёмся к нашей задаче. Снова, как и в предыдущей задаче, стартуем из состояния `Init`, выполним все переходы, придя к какому-то состоянию r . Тогда ответом будет `endpos[r] - P.length() + 1`.
- Данна строка P , требуется за $O(|P| + AnsSize)$ вывести позиции **всех вхождений** в текст S .
Посчитаем величины `endpos`, как описано в предыдущей задаче, также добавим специальный флаг `clonned[r]`, который будет равен `true` для состояний, полученных расщеплением (т.е. для `clone`), и `false` для остальных. Пусть мы выполним все переходы по строке P , пришли в некоторое состояние r . Первое вхождение в текст S оканчивается в позиции `endpos[r]`. Как нам найти список состояний, для которых `endpos` будет также указывать на конец некоторого вхождения? Если немного подумать, то понятно, что для этого достаточно запустить обход в ширину/глубину по инвертированным суффиксным ссылкам `link`. Действительно, если на некоторое состояние r

указывают суффиксные ссылки из других состояний q_1, q_2, \dots , то строка, соответствующая p , является суффиксом строк, соответствующих им; это и означает то, что $\text{endpos}[q_1], \text{endpos}[q_2], \dots$ дадут нам позиции окончания вхождений, равно как и $\text{endpos}[p]$. Пустив обход в ширину/глубину по инвертированным суффиксным ссылкам, мы найдём все позиции, в которых оканчиваются вхождения (так как циклов в графе из суффиксных ссылок нет, то для обхода в ширину/глубину даже не понадобится массив `used`). Однако при таком способе в выводе появятся повторы; но нетрудно заметить, что все повторы вызваны только лишь `clonned`-вершинами. Действительно, с одной стороны, если не добавлять `endpos` от `clonned`-вершин в ответ, то повторы никак не могут возникнуть - величины `endpos` различны для всех оставшихся вершин. С другой стороны, заметим, что при расщеплении выполняется: $\text{link}[q] = \text{clone}$, т.е. при обходе в ширину из вершины `clone` мы рано или поздно дойдём до q , и выведем её `endpos`. Следовательно, мы ничего не потеряем в ответе, если просто не будет выводить `endpos` для `clonned`-вершин.

Осталось только понять, что асимптотика этого обхода действительно будет $O(\text{AnsSize})$. Но это очевидный факт, поскольку `ne-clonned`-вершины добавляют в ответ по одному числу, а из каждой `clonned`-вершины достижима хотя бы одна `ne-clonned`.

Наконец, число реализационный момент - что здесь для экономии памяти вместо введения флага `clonned` можно просто присваивать `endpos[clone] = -1`.

Для большей ясности приведём здесь часть кода:

выполняющую построение графа инвертированных суффиксных ссылок:

```
struct state {
    int length, link, endpos;
    map<char, int> next;
    vector<int> ilink;
};

int main() {
    ... построение автомата ...
    for (int i=1; i<sz; ++i)
        st[st[i].link].ilink.push_back (i);
    ...
}
```

и выполняющую вывод ответа:

```
void dfs (int p, int len) {
    if (st[p].endpos != -1)
        printf ("%d ", st[p].endpos-len+2); // при выводе делаем 1-индексацию
    for (size_t i=0; i<st[p].ilink.size(); ++i)
        dfs (st[p].ilink[i], len);
}

int main() {
    ...
    dfs (p, (int)P.length()); // p - текущее состояние, P - искомая подстрока
    ...
}
```

Осталось только заметить, что выводиться позиции будут в неизвестно каком порядке, совсем необязательно, что отсортированными.

- Данна строка P , требуется найти за $O(|P|)$ количество вхождений её в текст S .

Посчитаем для каждого состояния такую динамику: $\text{cnt}[p]$ - количество вхождений строк, которым соответствует состояние p . Тогда ответом к задаче будет просто величина $\text{cnt}[p]$, где p - состояние, соответствующее строке P . Пронициализируем $\text{cnt}[p] = 1$, если p - `ne-clonned`-вершина, и 0 - если `clonned`. Потом посчитаем динамику таким образом: $\text{cnt}[\text{link}[p]] += \text{cnt}[p]$. Разумеется, вычислять её нужно в таком порядке, чтобы к этому моменту $\text{cnt}[p]$ уже было вычислено. Например, это можно сделать, отсортировав состояния по длине `length` в порядке уменьшения.

- По данной строке S найти за $O(|S|)$ наикратчайшую подстроку, не входящую в S в качестве подстроки.

Это решается динамикой по автомату. Пусть $d[p]$ - это ответ для состояния p - длина кратчайшей подстроки. Если из p нет перехода хотя бы по одной букве, то $d[p] = 1$ (мы берём эту букву; короче, очевидно, строки найти не может), иначе же мы одной буквой обойтись не может, и тогда $d[p]$ будет равно минимуму из $1 + d[\text{st}[p].\text{next}[c]]$ для всех c (т.е. мы добавляем одну букву и берём ответ для этого состояния). Восстановить саму строку - ответ, - также будет несложно.

- Найти наименьший циклический сдвиг строки S за $O(|S|)$.

Здесь построим автомат не для строки S , а для строки $S+S$. Тогда наша задача сводится к нахождению пути длины N (имеется в виду N рёбер) с наименьшей меткой (метка пути - строка, соответствующая ему). Но это абсолютно элементарная задача - начнём из состояния `Init` и будем идти, на каждом шаге жадно выполняя переход с наименьшей буквой (нетрудно понять, почему мы никогда не упрёмся в "тупик", из которого нет ни одного перехода).

- Количество различных подстрок за $O(|S|)$.

Эту задачу решим динамикой по состояниям автомата. Пусть $d[p]$ - ответ для состояния p , тогда $d[p] = \text{Сумма } (d[q] + 1)$ по всем переходам из p в состояния q . Действительно, мы как бы фиксируем первый символ подстроки, берём ответ для нового состояния, и затем прибавляем к нему единицу - это под строка длины 1, состоящая только из этого фиксированного символа.

- Суммарная длина всех различных подстрок за $O(|S|)$.

Эта задача очень похожа на предыдущую, и здесь нам также понадобится та же динамика $d[p]$ - количество различных подстрок. Сам же ответ `SumLen` будет считаться как $\text{SumLen}[p] = \text{СУММА } (d[q] + \text{SumLen}[q] + 1)$ - мы как бы дописываем первый символ к каждой из строк, учтённых в `SumLen[q]`.

- Данна строка T , и требуется найти все вхождения в ней строки S за $O(|T|)$ при том, что автомат по-прежнему построен для строки S (это обратная постановка задачи поиска строк).

Стартуем из состояния `Start`, и пусть p - это текущее состояние, а l - текущая длина совпадающего куска (изначально она, очевидно, равна нулю), и будем по очереди перебирать символы строки T ; пусть текущий символ - символ C . Если из текущего состояния p есть переход по символу C , то просто совершаем этот переход, и увеличиваем l на единицу; если l стало равно длине строки S , то мы нашли вхождение - оно как раз оканчивается в текущей позиции в строке T . Если же из текущего состояния p перехода нет, то это означает, что совпадающий кусок мы никак не можем увеличить; значит, нам надо пытаться найти наибольший суффикс совпадающего куска такой, что к нему можно дописать символ C . Вспоминая понятие суффиксной ссылки, это можно сформулировать так: будем двигаться по суффиксному пути состояния p , пока не встретим состояние, из которого есть переход по символу C . Если мы найдём такое состояние, то надо установить значение $l: l = \text{length}[p] + 1$ (поскольку символ C дописываем). Если же такого состояния мы так и не найдём, то $l = 0$, $p = \text{Init}$ - т.е. совпадающая часть вообще пуста.

Нетрудно понять, почему этот код работает за $O(|T|)$ - мы на каждом шаге либо увеличиваем длину совпадающей части на единицу, либо же уменьшаем на сколько-либо (на неотрицательную величину).

Для большей ясности приведём фрагмент реализации (в предположении, что `next` задано массивом, но сути это никак не меняет):

```

int l = 0, p = 0;
for (size_t i=0; i<t.length(); ++i) {
    char c = t[i];
    if (st[p].next[c] != -1)
        ++l, p = st[p].next[c];
    else {
        for (; p!=-1 && st[p].next[c]==-1; p=st[p].link) ;
        if (p == -1)
            l = 0, p = 0;
        else
            l = st[p].length + 1, p = st[p].next[c];
    }
    if (l == (int)s.length())
        cout << i-l+1 << ' ';
}

```

Кстати говоря, можно ещё ускорить этот алгоритм (уменьшить скрытую константу), если предпосчитать заранее результат выполнения цикла for. Действительно, результат этого цикла зависит только от текущего состояния p и текущего символа C , и результат выполнения этого цикла (так называемые "оптимизированные суффиксные ссылки") - новое значение p - можно предпосчитать за $O(|S|K)$.

- Даны строки S и T , требуется найти их **наи длиннейшую общую подстроку** (LCS) за $O(|S|+|T|)$.

Построим по строке S суффиксный автомат. Теперь будем идти по строке T и для каждой позиции в ней находить длину наи длиннейшего суффикса, оканчивающегося в этой позиции и встречающегося в S . Ясно, что искомая наи длиннейшая общая подстрока будет оканчиваться в той позиции, в которой эта величина максимальна.

Будем поддерживать переменную p - текущее состояние в автоматае, и l - текущая длина наи длиннейшего суффикса. Изначально $p = \text{Init}$, $l = 0$. Пусть мы нашли эти значения для $T[i-1]$, покажем, как найти их для $T[i]$. Если из состояния p суффиксного автомата (для строки S) есть переход по букве $T[i]$, то просто выполняем этот переход (т.е. $p = st[p].next[T[i]]$) и увеличиваем длину суффикса на единицу ($l++$). Если же из состояния p такого перехода нет, то мы вынуждены попытаться укоротить текущий суффикс, т.е. перейти по его суффиксной ссылке (т.е. $p = st[p].link$) и выбрать в качестве текущего суффикса длиннейшую строку, соответствующую этому состоянию (т.е. $l = st[p].length$); причём это мы должны повторять до тех пор, пока не найдём состояние p с имеющимся переходом по $T[i]$ (тогда мы приходим к предыдущему пункту - т.е. выполняем переход и увеличиваем l), или пока у нас не кончатся состояния (т.е. p не станет равно -1).

После построения автомата за $O(|S|)$ выполняется только $O(|T|)$ действий, поскольку величина l неотрицательна, и при обработке каждого символа может увеличиться максимум на единицу (а при переходе по суффиксной ссылке она строго уменьшается).

Итого имеем такой код:

```

string lcs (string a, string b) {
    init();
    for (size_t i=0; i<a.length(); ++i)
        sa_extend (a[i]);

    int p = 0, l = 0, best = 0, bestpos = 0;
    for (size_t i=0; i<b.length(); ++i) {
        if (st[p].next[b[i]-'A'] == -1) {
            for (; p!=-1 && st[p].next[b[i]-'A'] == -1; p=st[p].link) ;
            if (p == -1) {
                p = l = 0;
                continue;
            }
            l = st[p].length;
        }
        p = st[p].next[b[i]-'A'];
        ++l;
        if (l > best)
            best = l, bestpos = (int)i;
    }
    return b.substr (bestpos-best+1, best);
}

```

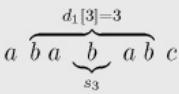
Нахождение всех подпалиндромов

Постановка задачи

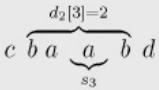
Дана строка s длины n . Требуется найти все такие пары (i, j) , где $i < j$, что подстрока $S[i \dots j]$ является палиндромом (т.е. читается одинаково слева направо и справа налево).

Понятно, что в худшем случае таких подстрок-палиндромов может быть $O(n^2)$, однако информацию можно возвращать более компактно: для каждой позиции $i = 0 \dots n - 1$ найдём значения $d_1[i]$ и $d_2[i]$, обозначающие количество палиндромов соответственно нечётной и чётной длины с центром в позиции i .

Например, для строки $s = abababc$ значение $d_1[3] = 3$:



А для строки $s = cbaabd$ значение $d_2[3] = 2$:



Идея в том, что если есть подпалиндром длины l с центром в какой-то позиции i , то есть также подпалиндромы длины $l - 2, l - 4$, и т.д. с центрами в i , поэтому двух таких массивов $d_1[i]$ и $d_2[i]$ достаточно для хранения информации обо всех подпалиндромах этой строки.

Более удивительным является то, что существует довольно простой алгоритм, который вычисляет эти массивы за линейное время, этот алгоритм описывается ниже.

Алгоритм решения

Эта задача имеет несколько известных решений: с помощью хэширования её можно решить за $O(n \log n)$, а с помощью суффиксных деревьев и быстрого алгоритма LCA эту задачу можно решить за $O(n)$.

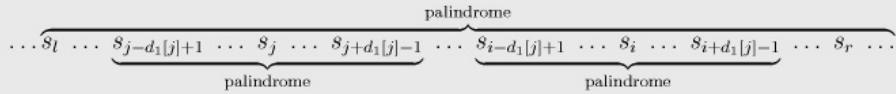
Однако описанный ниже метод значительно проще, и обладает меньшими скрытыми константами в асимптотике времени и памяти. Спасибо **Назарову Сергею** за описание этого красивого алгоритма (напоминающего [алгоритм вычисления Z-функции](#)).

Научимся сначала находить все подпалиндромы нечётной длины, т.е. вычислять массив $d_1[]$; решение для палиндромов чётной длины получится небольшой модификацией этого.

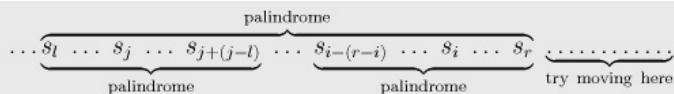
Для быстрого вычисления будем поддерживать **границы** (l, r) самого правого обнаруженного подпалиндрома (т.е. с наибольшим значением r). Изначально можно положить $l = 0, r = -1$.

Итак, пусть мы хотим вычислить значение $d_1[i]$ для очередного i , при этом все предыдущие значения $d_1[]$ уже подсчитаны. Если i не находится в пределах текущего подпалиндрома, т.е. $i > r$, то выполним тривиальный алгоритм (т.е. будем последовательно увеличивать значение $d_1[i]$, пока не найдём границу подпалиндрома); после этого мы должны не забыть обновить значения (l, r) .

Рассмотрим теперь случай, когда $i \leq r$. Попробуем извлечь часть информации из уже подсчитанных значений $d_1[]$, а именно, отразим позицию i внутри подпалиндрома (l, r) , т.е. получим позицию $j = l + (r - i)$, и рассмотрим значение $d_1[j]$. Поскольку j — позиция, симметричная позиции i , то мы можем взять ответ $d_1[j]$ в качестве ответа $d_1[i]$, но за одним исключением. Иллюстрация этого отражения в простом случае (палиндром вокруг j фактически "копируется" в палиндром вокруг i):



Особым случаем является случай, когда "внутренний палиндром" достигает границы внешнего или вылезает за неё, т.е. $j + d_1[j] - 1 \leq l$. Поскольку за границами внешнего палиндрома никакой симметрии нет, то при переносе подпалиндрома из позиции j в позицию i его нужно "обрезать", т.е. присвоить $d_1[i] = r - i$. После этого следует пустить тривиальный алгоритм, который будет пытаться увеличить значение $d_1[i]$ (он уже будет выходить за пределы "внешнего" палиндрома). Иллюстрация этого случая (на ней палиндром с центром в j уже "обрезан" до такой длины, что он впритык помещается во внешний):



Итак, мы описали поведение алгоритма в двух принципиально разных ситуациях. Осталось только заметить, что надо не забывать обновлять значения (l, r) после вычисления очередного значения $d_1[i]$.

Оценка асимптотики

Заметим, что на каждой, j -ой, стадии алгоритма сначала выполняются некоторые вычисления за $O(1)$, а затем запускается процесс тривиального обнаружения подпалиндромов большей длины. Более того, каждая итерация этого тривиального цикла (кроме последней) приводит в дальнейшем к продвижению указателя r вправо. Влево этот указатель у нас тоже никогда не двигается. Поскольку этот указатель не мог выйти за пределы всей строки, т.е. $r < n$, то мы получаем, что суммарно произойдёт не более n успешных итераций тривиального алгоритма. Т.к. все остальные части алгоритма работают также за $O(n)$, мы получаем итоговую асимптотику алгоритма: $O(n)$.

Реализация

Для случая палиндромов нечётной длины, т.е. для вычисления $d_1[]$ получаем такой код:

```
vector<int> d1 (n);
int l=0, r=-1;
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;
    while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;
    d1[i] = --k;
    if (i+k > r)
        l = i-k, r = i+k;
}
```

Для подпалиндромов чётной длины рассуждения те же самые, просто немного поменяются арифметические выражения:

```

vector<int> d2 (n);
l=0, r=-1;
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;
    while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;
    d2[i] = --k;
    if (i+k-1 > r)
        l = i-k, r = i+k-1;
}

```

Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига

Понятие декомпозиции Линдона

Определим понятие **декомпозиции Линдона** (Lyndon decomposition).

Строка называется **простой**, если она строго **меньше** любого своего собственного **суффикса**. Примеры простых строк: a, b, ab, aab, abb, ababb, abcd. Можно показать, что строка является простой тогда и только тогда, когда она строго **меньше** всех своих нетривиальных **циклических сдвигов**.

Далее, пусть дана строка S. Тогда **декомпозицией Линдона** строки S называется её разложение $S = W_1 W_2 \dots W_k$, где строки W_i просты, и при этом $W_1 \geq W_2 \geq \dots \geq W_k$.

Можно показать, что для любой строки S это разложение существует и единственno.

Алгоритм Дюваля

Алгоритм Дюваля (Duval's algorithm) находит для данной строки декомпозицию Линдона за время $O(N)$ с использованием $O(1)$ памяти (кроме самой строки S).

Пусть дана строка S длины N (как обычно, в 0-индексации). Требуется вывести позиции начала каждой простой строки в декомпозиции Линдона строки S.

Введём вспомогательное понятие предпростой строки. Страна T называется **предпростой**, если она имеет вид $T = W W W \dots W \text{Pref}W$, где W - некоторая простая строка, а $\text{Pref}W$ - некоторый префикс строки W.

Алгоритм Дюваля является жадным. В любой момент его работы строка S фактически разделена на три строки $S = S_1 S_2 S_3$, где в строке S_1 декомпозиция Линдона уже найдена и S_1 уже больше не используется алгоритмом; строка S_2 - это предпростая строка (причём длину простых строк внутри неё мы также запоминаем); строка S_3 - это ещё необработанная часть строки S. Каждый раз алгоритм Дюваля берёт первый символ строки S_3 и пытается дописать его к строке S_2 . При этом, возможно, для какого-то префикса строки S_2 декомпозиция Линдона становится известной, и эта часть переходит к строке S_1 .

Опишем теперь алгоритм **формально**. Во-первых, будет поддерживаться указатель i на начало строки S_2 . Внешний цикл алгоритма будет выполняться, пока $i < N$, т.е. пока вся строка S не перейдёт в строку S_1 . Внутри этого цикла создаются два указателя: указатель j на начало строки S_3 (фактически указатель на следующий символ-кандидат) и указатель k на текущий символ в строке S_2 , с которым будет производиться сравнение. Затем будем в цикле пытаться добавить символ $S[j]$ к строке S_2 , для чего необходимо произвести сравнение с символом $S[k]$. Здесь у нас возникают три различных случая:

- Если $S[j] = S[k]$, то мы можем дописать символ $S[j]$ к строке S_2 , не нарушив её "предпростоты". Следовательно, в этом случае мы просто увеличиваем указатели j и k на единицу.
- Если $S[j] > S[k]$, то, очевидно, строка $S_2 + S[j]$ станет простой. Тогда мы увеличиваем j на единицу, а k передвигаем обратно на i , чтобы следующий символ сравнивался с первым символом S_2 .
- Если $S[j] < S[k]$, то строка $S_2 + S[j]$ уже не может быть предпростой. Поэтому мы разбиваем предпростую строку S_2 на простые строки плюс "остаток" (префикс простой строки, возможно, пустой); простые строки добавляем в ответ (т.е. выводим их позиции, попутно передвигая указатель i), а "остаток" вместе с символом $S[j]$ переводим обратно в строку S_3 , и останавливаем выполнение внутреннего цикла. Тем самым мы на следующей итерации внешнего цикла заново обрабатываем остаток, зная, что он не мог образовать предпростую строку с предыдущими простыми строками. Осталось только заметить, что при выводе позиций простых строк нам нужно знать их длину; но она, очевидно, равна $j-k$.

Реализация

Приведём реализацию алгоритма Дюваля, которая будет выводить искомую декомпозицию Линдона строки S:

```

string s; // входная строка
int n = (int) s.length();
int i=0;
while (i < n) {
    int j=i+1, k=i;
    while (j < n && s[k] <= s[j]) {

```

```

if (s[k] < s[j])
    k = i;
else
    ++k;
++j;
}
while (i <= k) {
    cout << s.substr (i, j-k) << ' ';
    i += j - k;
}
}

```

Асимптотика

Сразу заметим, что для алгоритма Дювала требуется **O (1) памяти**, а именно три указателя i, j, k .

Оценим теперь **время работы** алгоритма.

Внешний цикл while делает не более N итераций, поскольку в конце каждой его итерации выводится как минимум один символ (а всего символов выводится, очевидно, ровно N).

Оценим теперь количество итераций **первого вложенного цикла while**. Для этого рассмотрим второй вложенный цикл `while` - он при каждом своём запуске выводит некоторое количество $r \geq 1$ копий одной и той же простой строки некоторой длины $p = j-k$. Заметим, что строка S_2 является предпростой, причём её простые строки имеют длину как раз p , т.е. её длина не превосходит $r p + p - 1$. Поскольку длина строки S_2 равна $j-i$, а указатель j увеличивается по единице на каждой итерации первого вложенного цикла `while`, то этот цикл выполнит не более $r p + p - 2$ итераций. Худшим случаем является случай $r = 1$, и мы получаем, что первый вложенный цикл `while` всякий раз выполняет не более $2 p - 2$ итераций. Вспоминая, что всего выводится N символов, получаем, что для вывода N символов требуется не более **$2 N - 2$ итераций** первого вложенного `while`-а.

Следовательно, **алгоритм Дювала выполняется за $O (N)$** .

Легко оценить и число сравнений символов, выполняемых алгоритмом Дювала. Поскольку каждая итерация первого вложенного цикла `while` производит два сравнения символов, а также одно сравнение производится после последней итерации цикла (чтобы понять, что цикл должен остановиться), то общее **число сравнений символов** не превосходит **$4 N - 3$** .

Нахождение наименьшего циклического сдвига

Пусть дана строка S . Построим для строки $S+S$ декомпозицию Линдана (мы можем это сделать за $O (N)$ времени и $O (1)$ памяти (если не выполнять конкатенацию в явном виде)). Найдём предпростой блок, который начинается в позиции, меньшей N (т.е. в первом экземпляре строки S), и заканчивается в позиции, большей или равной n (т.е. во втором экземпляре). Утверждается, что **позиция начала** этого блока и будет началом искомого циклического сдвига (в этом легко убедиться, воспользовавшись определением декомпозиции Линдана).

Начало предпростого блока найти просто - достаточно заметить, что указатель i в начале каждой итерации внешнего цикла `while` указывает на начало текущего предпростого блока.

Итого мы получаем такую **реализацию** (для упрощения кода она использует $O (N)$ памяти, явным образом дописывая строку к себе):

```

string min_cyclic_shift (string s) {
    s += s;
    int n = (int) s.length();
    int i=0, ans=0;
    while (i < n/2) {
        ans = i;
        int j=i+1, k=i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k) i += j - k;
    }
    return s.substr (ans, n/2);
}

```

Алгоритм Ахо-Корасик

Пусть дан набор строк в алфавите размера K суммарной длины M . Алгоритм Ахо-Корасик строит для этого набора строк бор, а затем по этому бору строит автомат, всё за $O (M)$ времени и $O (MK)$ памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых - это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время.

Данный алгоритм был предложен Альфредом Ахо (Aho) и Маргарет Корасик (Corasick) в 1975 г.

Бор. Построение бора

Формально, **бор** - это дерево с корнем в некоторой вершине Root, причём каждое ребро дерева подписано некоторой буквой. Если мы рассмотрим список рёбер, выходящих из данной вершины (кроме ребра, ведущего в предка), то все рёбра должны иметь разные метки.

Рассмотрим в боре любой путь из корня; выпишем подряд метки рёбер этого пути. В результате мы получим некоторую строку, которая соответствует этому пути. Если же мы рассмотрим любую вершину бора, то ей поставим в соответствие строку, соответствующую пути из корня до этой вершины.

Каждая вершина бора также имеет флаг leaf, который равен true, если в этой вершине оканчивается какая-либо строка из данного набора. Соответственно, **построить бор** по данному набору строк - значит построить такой бор, что каждой leaf-вершине будет соответствовать какая-либо строка из набора, и, наоборот, каждой строке из набора будет соответствовать какая-то leaf-вершина.

Опишем теперь, **как построить бор** по заданному набору строк за линейное время относительно их суммарной длины.

Введём структуру, соответствующую вершинам бора:

```
struct vertex {
    int next[K];
    bool leaf;
};

vertex t[NMAX+1];
int sz;
```

Т.е. мы будем хранить бор в виде массива t (количество элементов в массиве - это sz) структур vertex. Структура vertex содержит флаг leaf, и рёбра в виде массива $\text{next}[i]$, где $\text{next}[i]$ - указатель на вершину, в которую ведёт ребро по символу i , или -1, если такого ребра нет.

Вначале бор состоит только из одной вершины - корня (договоримся, что корень всегда имеет в массиве t индекс 0). Поэтому **инициализация** бора такова:

```
memset (t[0].next, 255, sizeof t[0].next);
sz = 1;
```

Теперь реализуем функцию, которая будет **добавлять в бор** заданную строку. Реализация крайне проста: мы встаём в корень бора, смотрим, есть ли из корня переход по букве $S[0]$: если переход есть, то просто переходим по нему в другую вершину, иначе создаём новую вершину и добавляем переход в эту вершину по букве $S[0]$. Затем мы, стоя в какой-то вершине, повторяем процесс для буквы $S[1]$, и т.д. После окончания процесса помечаем последнюю посещённую вершину флагом leaf = true.

```
void add_string (const string & s) {
    int v = 0;
    for (size_t i=0; i<s.length(); ++i) {
        char c = s[i] - 'a'; // в зависимости от алфавита
        if (t[v].next[c] == -1) {
            memset (t[sz].next, 255, sizeof t[sz].next);
            t[v].next[c] = sz++;
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}
```

Линейное время работы, а также линейное количество вершин в боре очевидны. Поскольку на каждую вершину приходится $O(K)$ памяти, то использование памяти есть $O(NK)$.

Потребление памяти можно уменьшить до линейного ($O(N)$), но за счёт увеличения асимптотики работы до $O(N \log K)$. Для этого достаточно хранить переходы next не массивом, а отображением $\text{map}<\text{char}, \text{int}>$.

Построение автомата

Пусть мы построили бор для заданного набора строк. Посмотрим на него теперь немного с другой стороны. Если мы рассмотрим любую вершину, то строка, которая соответствует ей, является префиксом одной или нескольких строк из набора; т.е. каждую вершину бора можно понимать как позицию в одной или нескольких строках из набора.

Фактически, вершины бора можно понимать как состояния **конечного детерминированного автомата**. Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние - т.е. в другую позицию в наборе строк. Например, если в боре находится только строка "abc" и мы стоим в состоянии 2 (которому соответствует строка "ab"), то под воздействием буквы "c" мы перейдём в состояние 3.

Т.е. мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние.

Более строго, пусть мы находимся в состоянии r , которому соответствует некоторая строка T , и хотим выполнить переход по символу C . Если в боре из вершины r есть переход по букве C , то мы просто переходим по этому ребру и попадаем в вершину, которой соответствует строка $T + C$. Если же такого ребра нет, то мы должны найти состояние, соответствующее **наиценнейшему** собственному суффиксу строки T (наиценнейшему из имеющихся в боре), и попытаться выполнить переход по букве C из него.

Например, пусть бор построен по строкам "ab" и "bc", и мы под воздействием строки "ab" перешли в некоторое состояние, являющееся листом. Тогда под воздействием буквы "c" мы вынуждены перейти в состояние, соответствующее строке "b", и только оттуда выполнить переход по букве "c".

Суффиксная ссылка для каждой вершины r - это вершина, в которой оканчивается **наиценнейший** собственный суффикс строки, соответствующей вершине r . Единственный особый случай - корень бора; суффиксную ссылку из него проведём в себя же. Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке.

Таким образом, мы свели задачу построения автомата к задаче нахождения суффиксных ссылок для всех вершин бора. Однако строить эти суффиксные ссылки мы будем, как ни странно, с помощью построенных в автомате переходов.

Заметим, что если мы хотим узнать суффиксную ссылку для некоторой вершины v , то мы можем перейти в предка r текущей вершины (r пусть С - буква, по которой из r есть переход в v), затем перейти по его суффиксной ссылке, а затем из неё выполнить переход в автомата по букве С.

Таким образом, задача нахождения перехода свелась к задаче нахождения суффиксной ссылки, а задача нахождения суффиксной ссылки - к задаче нахождения суффиксной ссылки и перехода, но уже для более близких к корню вершин. Мы получили рекурсивную зависимость, но не бесконечную, и, более того, разрешить которую можно за линейное время.

Перейдём теперь к **реализации**. Заметим, что нам теперь понадобится для каждой вершины хранить её предка r , а также символ pch , по которому из предка есть переход в нашу вершину. Также в каждой вершине будем хранить $int link$ - суффиксная ссылка (или -1 , если она ещё не вычислена), и массив $int go[K]$ - переходы в автомата по каждому из символов (опять же, если элемент массива равен -1 , то он ещё не вычислен). Приведём теперь полную реализацию всех необходимых функций:

```
struct vertex {
    int next[K];
    bool leaf;
    int p;
    char pch;
    int link;
    int go[K];
};

vertex t[NMAX+1];
int sz;

void init() {
    t[0].p = t[0].link = -1;
    memset(t[0].next, 255, sizeof t[0].next);
    memset(t[0].go, 255, sizeof t[0].go);
    sz = 1;
}

void add_string (const string & s) {
    int v = 0;
    for (size_t i=0; i<s.length(); ++i) {
        char c = s[i] - 'a';
        if (t[v].next[c] == -1) {
            memset(t[sz].next, 255, sizeof t[sz].next);
            memset(t[sz].go, 255, sizeof t[sz].go);
            t[sz].link = -1;
            t[sz].p = v;
            t[sz].pch = c;
            t[v].next[c] = sz++;
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}

int go (int v, char c);

int get_link (int v) {
    if (t[v].link == -1)
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go (get_link (t[v].p), t[v].pch);
    return t[v].link;
}

int go (int v, char c) {
    if (t[v].go[c] == -1)
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v==0 ? 0 : go (get_link (v), c);
    return t[v].go[c];
}
```

Нетрудно понять, что, за счёт запоминания найденных суффиксных ссылок и переходов, суммарное время нахождения всех суффиксных ссылок и переходов будет линейным.

Применения

Поиск всех строк из заданного набора в тексте

Дан набор строк, и дан текст. Требуется вывести все вхождения всех строк из набора в данный текст за время $O(Len + Ans)$, где Len - длина текста, Ans - размер ответа.

Построим по данному набору строк бор. Будем теперь обрабатывать текст по одной букве, перемещаясь соответствующим образом по дереву, фактически - по состояниям автомата. Изначально мы находимся в корне дерева. Пусть мы на очередном шаге мы находимся в состоянии v , и очередная буква текста c . Тогда следует переходить в состояние $go(v, c)$, тем самым либо увеличивая на 1 длину текущей совпадающей подстроки, либо уменьшая её, проходя по суффиксной ссылке.

Как теперь узнать по текущему состоянию v , имеется ли совпадение с какими-то строками из набора? Во-первых, понятно, что если мы стоим в помеченной вершине ($leaf=true$), то имеется совпадение с тем образцом, который в боре оканчивается в вершине v . Однако это далеко не

единственный возможный случай достижения совпадения: если мы, двигаясь по суффиксным ссылкам, мы можем достигнуть одной или нескольких помеченных вершин, то совпадение также будет, но уже для образцов, оканчивающихся в этих состояниях. Простой пример такой ситуации - когда набор строк - это { "dabc", "abc", "bc" }, а текст - это "dabc".

Таким образом, если в каждой помеченной вершине хранить номер образца, оканчивающегося в ней (или список номеров, если допускаются повторяющиеся образцы), то мы можем для текущего состояния за $O(N)$ найти номера всех образцов, для которых достигнуто совпадение, просто пройдя по суффиксным ссылкам от текущей вершины до корня. Однако это недостаточно эффективное решение, поскольку в сумме асимптотика получится $O(N \cdot Len)$. Однако можно заметить, что движение по суффиксным ссылкам можно соптимизировать, предварительно посчитав для каждой вершины ближайшую к ней помеченную вершину, достижимую по суффиксным ссылкам (это называется "функцией выхода"). Эту величину можно считать ленивой динамикой за линейное время. Тогда для текущей вершины мы сможем за $O(1)$ находить следующую в суффиксном пути помеченную вершину, т.е. следующее совпадение. Тем самым, на каждое совпадение будет тратиться $O(1)$ действий, и в сумме получится асимптотика $O(Len + Ans)$.

В более простом случае, когда надо найти не сами вхождения, а только их количество, можно вместо функции выхода посчитать ленивой динамикой количество помеченных вершин, достижимых из текущей вершины по суффиксным ссылкам. Эта величина может быть посчитана за $O(N)$ в сумме, и тогда для текущего состояния в мы сможем за $O(1)$ найти количество вхождений всех образцов в текст, оканчивающихся в текущей позиции. Тем самым, задача нахождения суммарного количества вхождений может быть решена нами за $O(Len)$.

Нахождение лексикографически наименьшей строки данной длины, не содержащей ни один из данных образцов

Дан набор образцов, и дана длина L . Требуется найти строку длины L , не содержащую ни один из образцов, и из всех таких строк вывести лексикографически наименьшую.

Построим по данному набору строк бор. Вспомним теперь, что те вершины, из которых по суффиксным ссылкам можно достичь помеченных вершин (а такие вершины можно найти за $O(N)$, например, ленивой динамикой), можно воспринимать как вхождение какой-либо строки из набора в заданный текст. Поскольку в данной задаче нам необходимо избегать вхождений, то это можно понимать как то, что в такие вершины нам заходить нельзя. С другой стороны, во все остальные вершины мы заходить можем. Таким образом, мы удаляем из автомата все "плохие" вершины, а в оставшемся графе автомата требуется найти лексикографически наименьший путь длины L . Эту задачу уже можно решить за $O(L)$, например, поиском в глубину.

Нахождение кратчайшей строки, содержащей вхождения одновременно всех образцов

Снова воспользуемся той же идеей. Для каждой вершины будем хранить маску, обозначающую образцы, для которых произошло вхождение в данной вершине. Тогда задачу можно переформулировать так: изначально находясь в состоянии ($V=root, Msk=0$), требуется дойти до состояния ($V,Msk=2^N-1$), где N - количество образцов. Переходы из состояния в состояние будут представлять собой добавление одной буквы к тексту, т.е. переход по ребру автомата в другую вершину с соответствующим изменением маски. Запустив обход в ширину на таком графе, мы найдём путь до состояния ($V,Msk=2^N-1$) наименьшей длины, что нам как раз и требовалось.

Нахождение лексикографически наименьшей строки длины L , содержащей данные образцы в сумме K раз

Как и в предыдущих задачах, посчитаем для каждой вершины количество вхождений, которое соответствует ей (т.е. количество помеченных вершин, достижимых из неё по суффиксным ссылкам). Переформулируем задачу таким образом: текущее состояние определяется тройкой чисел (V,Len,Cnt), и требуется из состояния ($Root,0,0$) прийти в состояние (V,L,K), где V - любая вершина. Переходы между состояниями - это просто переходы по рёбрам автомата из текущей вершины.

Поиск подстроки в строке с помощью Z- или Префикс-функции

Даны строки S и T . Требуется найти все вхождения строки S в текст T за $O(N)$, где N - суммарная длина строк S и T .

Алгоритм

Образуем строку $S\$T$, где $\$$ - некий разделитель, который не встречается ни в S , ни в T .

Вычислим для полученной строки **префикс-функцию** P за $O(N)$. Пройдёмся по массиву P , и рассмотрим все его элементы, которые равны $|S|$ (длине S). По определению префикс-функции, это означает, что в это месте оканчивается подстрока, совпадающая с $|S|$, т.е. искомое вхождение. Таким образом, мы нашли все вхождения. Этот алгоритм называется алгоритмом **КМП (Кнута-Морриса-Пратта)**.

Теперь решим эту же задачу с помощью **Z-функции**. Построим за $O(N)$ массив Z - Z-функцию строки $S\$T$. Пройдёмся по всем его элементам, и рассмотрим те из них, которые равны $|S|$. По определению, в этом месте начинается подстрока, совпадающая с S . Таким образом, мы нашли все вхождения.

Решение задачи "сжатие строки" за O (N)

Дана строка S. Требуется найти такую строку T, что строка S получается многократным повторением T. Из всех возможных T нужно выбрать наименьшую по длине.

Эту задачу очень просто решить за O (N) с помощью префикс-функции.

Итак, пусть массив P - префикс-функция строки S, которую можно вычислить за O (N).

Теперь рассмотрим значение последнего элемента P: P[N-1]. Если N делится на (N - P[N-1]), то ответ существует, и это N - P[N-1] первых букв строки S. Если же не делится, то ответа не существует.

Корректность этого метода легко понять. P[N-1] равно длине найденнейшего собственного суффикса строки S, совпадающего с префиксом S. Если ответ существует, то, очевидно, начальный кусок строки S длиной (N - P[N-1]) будет ответом, и, следовательно, N будет делиться на (N - P[N-1]). Если же ответа не существует, то (N - P[N-1]) будет равно какому-то непонятному значению, на которое N делиться не будет (иначе бы ответ существовал).

Реализация

```
int n = (int) s.length();
vector<int> p (n);
// ... здесь вычисление префикс-функции ...
int l = n - p[n-1];
if (n % l == 0)
    cout << s.substr (l);
else
    cout << "No Solution";
```

Sqrt-декомпозиция

Sqrt-декомпозиция — это метод, или структура данных, которая позволяет некоторые типичные операции (суммирование элементов подмассива, нахождение минимума/максимума и т.д.) за $O(\sqrt{n})$, что значительно быстрее, чем $O(n)$ для тривиального алгоритма.

Описание

Поставим задачу. Дан массив $A[0 \dots n - 1]$. Требуется реализовать такую структуру данных, которая сможет находить сумму элементов $A[l \dots r]$ для произвольных l и r за $O(\sqrt{n})$ операций.

Основная идея sqrt-декомпозиции заключается в том, что сделаем следующий **предпосчёт**: разделим массив A на блоки длины \sqrt{n} (разумеется, округлённому к целому), и в каждом блоке заранее предпосчитаем сумму элементов в нём. Пусть len — это длина блока ($len \approx \sqrt{n}$), а $cnt = \left\lceil \frac{n}{len} \right\rceil$ — количество блоков:

$$\underbrace{a_0 a_1 \dots a_{len-1}}_{b_0} \underbrace{a_{len} a_{len+1} \dots a_{2len-1}}_{b_1} \dots \underbrace{a_{(cnt-1)len} \dots a_n}_{b_{cnt-1}}$$

Через b_k мы обозначили сумму в k -ом блоке. Хотя последний блок может содержать меньше, чем len , элементов (если n не делится на len), но это не принципиально.

Итак, пусть эти значения b_k предварительно подсчитаны (для этого надо, очевидно, $O(n)$ операций). Что они могут дать при вычислении ответа на очередной запрос (l, r) ? Заметим, что если отрезок $[l; r]$ длинный, то в нём будет содержаться много блоков целиком, а для каждого такого блока сумму в нём мы уже знаем, поэтому на такие блоки нам не надо тратить операции:

$$\dots \overbrace{a_l \dots a_{klen-1}}^{sum=?} \underbrace{a_{klen} \dots a_{(k+1)len-1}}_{b_k} \underbrace{a_{(k+1)len} \dots a_{(k+2)len-1}}_{b_{k+1}} \dots \underbrace{a_{plen} \dots a_{(p+1)len-1}}_{b_p} \underbrace{a_{(p+1)len} \dots a_r}_{b_p} \dots$$

На этом рисунке видно, что для того чтобы посчитать сумму в отрезке $[l \dots r]$, надо просуммировать элементы только в двух "хвостах": $[l \dots (k+1)len-1]$ и $[(p+1)len \dots r]$, и просуммировать значения b_i во всех блоках, начиная с k и заканчивая p :

$$\sum_{i=l}^r a_i = \sum_{i=l}^{(k+1)len-1} a_i + \sum_{i=k}^p b_i + \sum_{i=(p+1)len}^r a_i$$

Тем самым мы экономим значительное количество операций. Действительно, размер каждого из "хвостов", очевидно, не превосходит длины

блока len , а количество блоков не превосходит cnt . Поскольку и len , и cnt мы выбирали $\approx \sqrt{n}$, то всего для вычисления суммы в отрезке $[l \dots r]$ нам понадобится лишь $O(\sqrt{n})$ операций.

Другие задачи

Мы рассматривали задачу нахождения суммы элементов массива в каком-то его подотрезке. Сразу заметим, что эту задачу можно чуть расширить: разрешим также **меняться** отдельным элементам массива A . Действительно, если меняется какой-то элемент a_i , то достаточно обновить значение b_k в том блоке, в котором этот элемент находится:

$$b_k := a_i - old_a_i_val \\ (k = i/\text{len})$$

С другой стороны, вместо задачи о сумме аналогично можно решать задачи о **минимальном, максимальном** элементах в отрезке. Если в этих задачах допускать изменения отдельных элементов, то тоже надо будет пересчитывать значение b_k того блока, которому принадлежит изменяющийся элемент, но пересчитывать уже полностью, проходом по всем элементам блока за $O(\text{len}) = O(\sqrt{n})$ операций.

Аналогичным образом sqrt-декомпозицию можно применять и для множества **других** подобных задач: нахождение количества нулевых элементов, первого ненулевого элемента, подсчёта количества определённых элементов, и т.д.

Есть и целый класс задач, когда происходят **изменения элементов на целом подотрезке**: прибавление или присвоение элементов на каком-то подотрезке массива A .

Например, нужно выполнять следующие два вида запросов: прибавить ко всем элементам некоторого отрезка $[l; r]$ величину δ , и узнавать значение отдельного элемента a_i . Тогда в качестве b_k положим ту величину, которая должна быть прибавлена ко всем элементам k -го блока (например, изначально все $b_k = 0$); тогда при выполнении запроса "прибавление" нужно будет выполнить прибавление ко всем элементам a_i "хвостов", а затем выполнить прибавление ко всем элементам b_i для блоков, целиком лежащих в отрезке $[l \dots r]$. А ответом на второй запрос, очевидно, будет просто $a_i + b_k$, где $k = i/\text{len}$. Таким образом, прибавление на отрезке будет выполняться за $O(\sqrt{n})$, а запрос отдельного элемента — за $O(1)$.

Наконец, можно комбинировать оба вида задач: изменение элементов на отрезке и ответ на запросы тоже на отрезке. Оба вида операций будут выполняться за $O(\sqrt{n})$. Для этого уже надо будет делать два "блоковых" массива b и c : один — для обеспечения изменений на отрезке, другой — для ответа на запросы.

Реализация

Приведём сначала простейшую реализацию sqrt-декомпозиции для задачи о нахождении суммы в произвольном подотрезке:

```
// входные данные
int n;
vector<int> a (n);

// предпосчёт
int bl = (int) sqrt (n + .0) + 1; // и размер блока, и количество блоков
vector<int> b (bl);
for (int i=0; i<n; ++i)
    b[i / bl] += a[i];

// ответ на запросы
for (;;) {
    int l, r; // входные данные - очередной запрос
    int sum = 0;
    for (int i=l; i<=r; )
        if (i % bl == 0 && i + bl - 1 <= r) {
            // если i указывает на начало блока, целиком лежащего в [l;r]
            sum += b[i / bl];
            i += bl;
        }
        else {
            sum += a[i];
            ++i;
        }
}
```

Недостатком этой реализации является то, что в ней неоправданно много операций деления (которые, как известно, выполняются значительно медленнее других операций). Вместо этого можно хранить номер текущего блока и позицию в нём, и увеличивать вместе с i эту позицию в текущем блоке; если позиция в блоке достигла его конца, то передвигаемся к следующему блоку. Для примера переделаем ту часть кода, в которой происходит вычисление ответа на запрос:

```
int l, r;
int sum = 0;
for (int i=l, block=l/bl, pos=l%bl; i<=r; )
    if (pos == 0 && i + bl - 1 <= r) {
        sum += b[block];
        i += bl;
        ++block;
    }
    else {
        sum += a[i];
        ++i;
        ++pos;
        if (pos == bl) {
            pos = 0;
            ++block;
        }
    }
}
```

```
}
```

Дерево Фенвика

Дерево Фенвика - это структура данных, дерево на массиве, обладающее следующими свойствами:

- 1) позволяет вычислять значение некоторой обратимой операции G на любом отрезке $[L; R]$ за время $O(\log N)$;
- 2) позволяет изменять значение любого элемента за $O(\log N)$;
- 3) требует $O(N)$ памяти, а точнее, ровно столько же, сколько и массив из N элементов;
- 4) легко обобщается на случай многомерных массивов.

Наиболее распространённое применение дерева Фенвика - для вычисления суммы на отрезке, т.е. функция $G(X_1, \dots, X_k) = X_1 + \dots + X_k$.

Дерево Фенвика было впервые описано в статье "A new data structure for cumulative frequency tables" (Peter M. Fenwick, 1994).

Описание

Для простоты описания мы предполагаем, что операция G , по которой мы строим дерево, - это **сумма**.

Пусть дан массив $A[0..N-1]$. Дерево Фенвика - массив $T[0..N-1]$, в каждом элементе которого хранится сумма некоторых элементов массива A :

```
Ti = сумма Aj для всех F(i) <= j <= i,
```

где $F(i)$ - некоторая функция, которую мы определим несколько позже.

Теперь мы уже можем написать **псевдокод** для функции вычисления суммы на отрезке $[0; R]$ и для функции изменения ячейки:

```
int sum (int r)
{
    int result = 0;
    while (r >= 0) {
        result += t[r];
        r = f(r) - 1;
    }
    return result;
}

void inc (int i, int delta)
{
    для всех j, для которых F(j) <= i <= j
    {
        t[j] += delta;
    }
}
```

Функция sum работает следующим образом. Вместо того чтобы идти по всем элементам массива A , она движется по массиву T , делая "прыжки" через отрезки там, где это возможно. Сначала она прибавляет к ответу значение суммы на отрезке $[F(R); R]$, затем берёт сумму на отрезке $[F(F(R)-1); F(R)-1]$, и так далее, пока не дойдёт до нуля.

Функция inc движется в обратную сторону - в сторону увеличения индексов, обновляя значения суммы T_j только для тех позиций, для которых это нужно, т.е. для всех j , для которых $F(j) <= i <= j$.

Очевидно, что от выбора функции F будет зависеть скорость выполнения обеих операций. Сейчас мы рассмотрим функцию, которая позволит достичь логарифмической производительности в обоих случаях.

Определим значение $F(X)$ следующим образом. Рассмотрим двоичную запись этого числа и посмотрим на его младший бит. Если он равен нулю, то $F(X) = X$. Иначе двоичное представление числа X оканчивается на группу из одной или нескольких единиц. Заменим все единицы из этой группы на нули, и присвоим полученное число значению функции $F(X)$.

Этому довольно сложному описанию соответствует очень простая формула:

```
F(X) = X & (X+1),
```

где $\&$ - это операция побитового логического "И".

Нетрудно убедиться, что эта формула соответствует словесному описанию функции, данному выше.

Нам осталось только научиться быстро находить такие числа j , для которых $F(j) <= i <= j$.

Однако нетрудно убедиться в том, что все такие числа j получаются из i последовательными заменами самого правого (самого младшего) нуля в двоичном представлении. Например, для $i = 10$ мы получим, что $j = 11, 15, 31, 63$ и т.д.

Как ни странно, такой операции (замена самого младшего нуля на единицу) также соответствует очень простая формула:

$$H(x) = x \mid (x+1),$$

где \mid - это операция побитового логического "ИЛИ".

Реализация дерева Фенвика для суммы для одномерного случая

```
vector<int> t;
int n;

void init (int nn)
{
    n = nn;
    t.assign (n, 0);
}

int sum (int r)
{
    int result = 0;
    for (; r >= 0; r = (r & (r+1)) - 1)
        result += t[r];
    return result;
}

void inc (int i, int delta)
{
    for (; i < n; i = (i | (i+1)))
        t[i] += delta;
}

int sum (int l, int r)
{
    return sum (r) - sum (l-1);
}

void init (vector<int> a)
{
    init ((int) a.size());
    for (unsigned i = 0; i < a.size(); i++)
        inc (i, a[i]);
}
```

Реализация дерева Фенвика для минимума для одномерного случая

Следует сразу заметить, что, поскольку дерево Фенвика позволяет найти значение функции в произвольном отрезке $[0;R]$, то мы никак не сможем найти минимум на отрезке $[L;R]$, где $L > 0$. Далее, все изменения значений должны происходить только в сторону уменьшения (опять же, поскольку никак не получится обратить функцию \min). Это значительные ограничения.

```
vector<int> t;
int n;

const int INF = 1000*1000*1000;

void init (int nn)
{
    n = nn;
    t.assign (n, INF);
}

int getmin (int r)
{
    int result = INF;
    for (; r >= 0; r = (r & (r+1)) - 1)
        result = min (result, t[r]);
    return result;
}

void update (int i, int new_val)
{
    for (; i < n; i = (i | (i+1)))
        t[i] = min (t[i], new_val);
}

void init (vector<int> a)
{
    init ((int) a.size());
    for (unsigned i = 0; i < a.size(); i++)
        update (i, a[i]);
}
```

Реализация дерева Фенвика для суммы для двумерного случая

Как уже отмечалось, дерево Фенвика легко обобщается на многомерный случай.

```
vector <vector <int> > t;
int n, m;

int sum (int x, int y)
{
    int result = 0;
    for (int i = x; i >= 0; i = (i & (i+1)) - 1)
        for (int j = y; j >= 0; j = (j & (j+1)) - 1)
            result += t[i][j];
    return result;
}

void inc (int x, int y, int delta)
{
    for (int i = x; i < n; i = (i | (i+1)))
        for (int j = y; j < m; j = (j | (j+1)))
            t[i][j] += delta;
}
```

Система непересекающихся множеств

Система непересекающихся множеств (disjoint set union, DSU) - это структура данных, которая может хранить несколько элементов, разделённых на несколько множеств (каждый элемент принадлежит ровно одному множеству, отсюда и название структуры). Каждое множество характеризуется своим представителем - одним из его элементов. Структура данных поддерживает следующие операции:

- **MakeSet (X)**

Добавляет в систему новый элемент X, который заносится в отдельное (новое) множество, представителем которого становится X. За O(1)

- **FindSet (X)**

Ищет множество, которому принадлежит элемент X, и возвращает его представителя.

За O(1) - амортизированная оценка

- **Union (X, Y)**

Ищет множества, к которым принадлежат элементы X и Y, и объединяет их в одно множество. Возвращает элемент, который становится представителем нового множества.

За O(1) - амортизированная оценка

Большинство фактов об этой структуре данных были установлены **Таржаном (Tarjan)** приблизительно в 1975 г.

Применение

Эта структура данных имеет несколько важных применений:

- Эффективная реализация алгоритма Крускала нахождения минимального остова
- Эффективная реализация для задачи минимума в автономном режиме
(т.е. нужно реализовать структуру данных, которая позволяет добавлять элементы из множества {1,..,N} и извлекать минимум; задача автономна в том смысле, что вся последовательность запросов известна к началу выполнения алгоритма)
- Эффективная реализация задачи о наименьшем общем предке (LCA) в автономном режиме
(т.е. вся последовательность запросов известна к началу выполнения алгоритма)

Примечание. Асимптотика

Следует заметить, что у операций FindSet и Union асимптотика несколько хуже, чем O(1). На самом деле, асимптотика для них равна O(alpha(N)), где alpha(N) - инверсия функции Аккермана:

```
alpha(N) = min { k : A_K(1) >= N }, где
A_K(J) = A_{K-1}^{J+1}(J) при K > 0, и
A_0(J) = J+1
```

Несколько первых значений функции alpha:

```
alpha(0)..alpha(2) = 0
alpha(3) = 1
alpha(4)..alpha(7) = 2
alpha(8)..alpha(2047) = 3
alpha(2048)..alpha(16512) = 4
```

Отсюда видно, что для всех мыслимых применений $\alpha(N) \leq 4$, а потому её можно считать константой, и приравнивать $O(1)$.

Описание алгоритмов

Пусть элементы X - это некоторые числа. Вся структура данных хранится в виде двух массивов: P и $Rank$.

Массив P содержит предков, т.е. $P[X]$ - это предок элемента X . Фактически, мы имеем древовидную структуру данных: двигаясь по предкам от любого элемента X , мы рано или поздно придём к представителю множества, к которому принадлежит X . В частности, если $P[X] = X$ для некоторого X , то это означает, что X является представителем множества, к которому он принадлежит, и, очевидно, X является корнем дерева.

Массив $Rank$ хранит ранги представителей, т.е. его значения имеют смысл только для элементов-представителей. Ранг некоторого элемента-представителя X - это верхняя граница его высоты в его дереве. Ранги используются в качестве эвристики в операции $Union$.

Теперь рассмотрим реализацию операций:

- **MakeSet (X)**

Эта операция очень проста - мы указываем, что $P[X] = X$, а ранг X равен 1.

- **FindSet (X)**

Будем двигаться от X по предкам, и рано или поздно мы найдём представителя. Однако важный момент - мы одновременно применяем следующую **эвристику**: у каждого элемента, который мы проходим, мы также исправляем P , указывая его сразу на найденного представителя. Т.е. фактически операция $FindSet$ двухпроходная: на первом проходе мы ищем представителя, а на втором исправляем значения P .

- **Union (X, Y)**

Сначала мы заменяем элементы X и Y на представителей их множеств, просто вызывая функцию $FindSet$. Мы объединяем два множества, присваивая $P[X] = Y$ или $P[Y] = X$. Однако выбор - что чему присваивается - осуществляется с помощью **эвристики**. Если ранги элементов X и Y отличны, то мы делаем корень с бо́льшим рангом родительским по отношению к корню с меньшим рангом. Если же ранги обоих элементов совпадают, то мы выбираем родителя произвольным образом, и увеличиваем его ранг на 1.

Следует ещё раз подчеркнуть важность двух **эвристик**, использованных в операциях $FindSet$ и $Union$. Без них асимптотика этих операций значительно ухудшится (до линейного вместо константного времени).

Реализация

```
vector<int> p, rank;

void init (int max_n)
{
    p.resize (max_n);
    for (int i=0; i<max_n; ++i)
        p[i] = i;
    rank.resize (max_n);
}

void make_set (int x)
{
    p[x] = x;
    rank[x] = 0;
}

int find_set (int x)
{
    if (x == p[x])    return x;
    return p[x] = find_set (p[x]);
}

void unite (int x, int y)
{
    x = find_set (x);
    y = find_set (y);
    if (rank[x] > rank[y])
        p[y] = x;
    else
    {
        p[x] = y;
        if (rank[x] == rank[y])
            ++rank[y];
    }
}
```

Рандомизация

Представленные выше алгоритмы были детерминированными. Однако, в некоторых случаях имеет смысл сделать операцию $Union$ **рандомизированной** - заменить эвристику по рангу на случайный выбор родительского узла. Тесты показывают, что такая реализация нисколько не отстает от детерминированного варианта, однако пишется и запоминается ещё легче:

```
vector<int> p;

void init (int max_n)
{
    p.resize (max_n);
    for (int i=0; i<max_n; ++i)
        p[i] = i;
}

void make_set (int x)
```

```

{
    p[x] = x;
}

int find_set (int x)
{
    if (x == p[x]) return x;
    return p[x] = find_set (p[x]);
}

void unite (int x, int y)
{
    x = find_set (x);
    y = find_set (y);
    if (rand() & 1)
        p[y] = x;
    else
        p[x] = y;
}

```

Дерево отрезков

Дерево отрезков - структура данных, которая позволяет реализовать за $O(\log N)$ операции следующего типа: нахождение суммы/минимума элементов массива в заданном отрезке ($A[L..R]$, где L и R - это параметры запроса), изменение одного элемента массива, изменение/прибавление элементов на отрезке ($A[L..R]$). При этом объём дополнительного используемой памяти составляет $O(N)$, или, если быть точным, не более $4N$.

Описание

Для простоты описания будем считать, что мы строим дерево отрезков для суммы.

Построим бинарное дерево T следующим образом. Корень дерева будет храниться в элементе $T[1]$. Он будет содержать сумму элементов $A[0..N-1]$, т.е. всего массива. Левый сын корня будет храниться в элементе $T[2]$ и содержать сумму первой половины массива $A: A[0..N/2]$, а правый сын - в элементе $T[3]$ и содержать сумму элементов $A[N/2+1..N-1]$. В общем случае, если $T[i]$ -й элемент содержит сумму элементов с L -го по R -ый, то его левым сыном будет элемент $T[i*2]$ и содержать сумму $A[L..(L+R)/2]$, а его правым сыном будет $T[i*2+1]$ и содержать сумму $A[(L+R)/2+1..R]$. Исключение, разумеется, составляют листья дерева - вершины, в которых $L = R$.

Далее, нетрудно заметить, что это дерево будет содержать $4N$ элементов (а высота дерева будет порядка $O(\log N)$). Поскольку значение в каждом элементе дерева однозначно определяется значениями в его сыновьях, то каждый элемент вычисляется за $O(1)$, а всё дерево **строится** за $O(N)$.

Рассмотрим теперь **операцию суммы** на некотором отрезке $[L; R]$. Мы встаём в корень дерева ($i=1$), и рекурсивно движемся вниз по этому дереву. Если в какой-то момент оказывается, что L и R совпадают с границами отрезка текущего элемента, то мы просто возвращаем значение текущего элемента T . Иначе, если отрезок $[L; R]$ целиком попадает в отрезок левого или правого сына текущего элемента, то мы рекурсивно вызываем себя из этого сына и найденное значение возвращаем. Наконец, если отрезок $[L; R]$ частично принадлежит и отрезку левого сына, и отрезку правого сына, то делим отрезок $[L; R]$ на два отрезка $[L; M]$ и $[M+1; R]$ так, чтобы первый отрезок целиком принадлежал отрезку левого сына, а второй отрезок - отрезку правого сына, и рекурсивно вызываем себя и от первого, и от второго отрезков, возвращая сумму найденных сумм. В итоге вся операция суммирования работает за $O(\log N)$.

Теперь рассмотрим **операцию изменения** значения некоторого элемента с индексом K . Будем спускаться по дереву от корня, ища тот лист, который содержит значение элемента $A[K]$. Когда мы найдём этот элемент, просто изменим соответствующее значение в массиве T и будем подниматься от текущего элемента обратно к корню, пересчитывая текущие значения T . Понятно, что таким образом мы изменим все значения в дереве, которые нужно изменить. Итого асимптотика $O(\log N)$.

Наконец, рассмотрим **операцию изменения на отрезке**. Для реализации этой операции нам понадобится немного модифицировать дерево. Пусть каждый элемент дерева, помимо суммы, будет содержать значение $Val[i]$: если все элементы массива A в текущем отрезке равны друг другу, то $Val[i]$ будет содержать это значение, а иначе он будет содержать некое значение "неопределённость". Изначально его можно просто заполнить значениями "неопределённость". А при выполнении операции изменения на отрезке мы будем спускаться по дереву, как в вышеописанном алгоритме суммирования, и если в какой-то момент L и R совпадут с границами текущего отрезка, то мы присвоим $Val[i]$ новое значение, которое мы хотим записать. Понятно, что теперь надо будет модифицировать операцию суммирования - если она в какой-то момент встречает $Val[i]$, отличное от "неопределённости", то она прекращает спуск по дереву и сразу возвращает нужное значение - действительно, результат уже определён значением $Val[i]$, а вот если мы продолжим спуск, то уже будем считывать неправильные, старые значения.

Операция прибавления на отрезке реализуется подобным образом, но несколько проще. В каждом элементе мы храним $Add[i]$ - значение, которое нужно прибавить ко всем элементам этого отрезка. Операция прибавления на отрезке будет модифицировать эти значения, а операция суммирования - просто прибавлять к ответу все встретившиеся значения Add .

Реализация

Например, рассмотрим дерево отрезков для суммы с одиночной модификацией:

```
vector<long long> t;
```

```

int n;

void build (const vector<int> & a, int i = 1, int l = 0, int r = n-1) {
    if (i == 1)
        t.resize (n*4 + 1);
    if (l == r)
        t[i] = a[l];
    else {
        int m = (l + r) / 2;
        build (a, i*2, l, m);
        build (a, i*2+1, m+1, r);
        t[i] = t[i*2] + t[i*2+1];
    }
}

long long sum (int l, int r, int i = 1, int tl = 0, int tr = n-1) {
    if (l == tl && r == tr)
        return t[i];
    int m = (tl + tr) / 2;
    if (r <= m)
        return sum (l, r, i*2, tl, m);
    if (l > m)
        return sum (l, r, i*2+1, m+1, tr);
    return sum (l, m, i*2, tl, m) + sum (m+1, r, i*2+1, m+1, tr);
}

void update (int pos, int newval, int i = 1, int l = 0, int r = n-1) {
    if (l == r)
        t[i] = newval;
    else {
        int m = (l + r) / 2;
        if (pos <= m)
            update (pos, newval, i*2, l, m);
        else
            update (pos, newval, i*2+1, m+1, r);
        t[i] = t[i*2] + t[i*2+1];
    }
}

```

Обобщение на двумерный случай

Пусть для определённости мы решаем такую задачу: дана матрица $A[1..N, 1..M]$, и поступают запросы вида (X_1, Y_1, X_2, Y_2) , и требуется найти минимум в матрице A среди элементов $A[X_1..X_2, Y_1..Y_2]$.

Обобщим дерево отрезков для решения этой задачи так, чтобы **строить его за $O(NM)$** , а **отвечать на каждый запрос** (в режиме он-лайн) **за $O(\log N \log M)$** .

А именно, сделаем дерево отрезков по координате X , но в каждой вершине I дерева (которой соответствуют некоторые L и R) будем хранить дерево отрезков по координате Y , которое построено для минимума среди значений матрицы в полосе $X=[L;R]$. Т.е.:

$T[I][J] = \min A[X][Y]$ по всем $X \in [L;R], Y \in [T;B]$,
где $T[1..4N][1..4M]$ – дерево отрезков,
 $[L;R]$ – отрезок, соответствующий вершине I в дереве отрезков по X ,
 $[T;B]$ – отрезок, соответствующий вершине J в дереве отрезков по Y

Построение дерева. При построении будем сначала делить отрезки по координате X , пока это возможно (пока L не станет равно R), а когда мы больше не сможем делить отрезок по X - начнём делить по Y . При этом деревья отрезков по координате Y строятся абсолютно так же, как и в одномерном случае - поскольку мы имеем $X=L=R$. А во всех остальных случаях, когда $L < R$ (т.е. когда мы делим по X), мы будем получать дерево отрезков в текущей вершине, объединяя два дерева отрезков по Y от потомков: от $X \in [L;M]$ и от $X \in [M+1;R]$; для этого достаточно пройтись по всем вершинам J обоих деревьев отрезков в потомках и выбирать минимум: $T[I][J] = \min(T[I^2][J], T[I^2+1][J])$. Ясно, что итоговая асимптотика составит $O(NM)$ - поскольку $O(N)$ тратится на построение дерева отрезков по X отдельно, $O(M)$ - на каждое дерево отрезков по Y , а всего деревьев отрезков по Y есть $O(N)$.

Ответ на запрос. Пусть поступает запрос вида (X_1, Y_1, X_2, Y_2) . Будем, так же, как и в одномерном дереве отрезков, выполнять поиск по дереву отрезка $[X_1, X_2]$. Как только в процессе разбиения по координате X мы нашли некоторый отрезок $[L;R]$ такой, что $X_1 = L$ и $X_2 = R$, то мы просто начинаем делить по Y , т.е. отвечать на запрос $[Y_1, Y_2]$ в дереве $T[]$. Таким образом, ответ на запрос складывается из поисков по двум деревьям отрезков (по X и по Y), и асимптотика получается $O(\log N \log M)$.

Реализация:

```

int n, m;
int a[500][500];
int t[4*500+1][4*500+1];

void build (int vx, int xl, int xr, int vy, int yl, int yr) {
    if (xl == xr)
        if (yl == yr)
            t[vx][vy] = a[xl][yl];
        else {
            int ym = (yl + yr) >> 1;
            build (vx, xl, xr, vy+vy, yl, ym);
            build (vx, xl, xr, vy+vy+1, ym+1, yr);
            t[vx][vy] = min (t[vx][vy+vy], t[vx][vy+vy+1]);
        }
    else {

```

```

int xm = (xl + xr) >> 1;
build (vx+vx, xl, xm, vy, yl, yr);
build (vx+vx+1, xm+1, xr, vy, yl, yr);
for (int i=0; i<=500*4; ++i)
    t[vx][i] = min (t[vx+vx][i], t[vx+vx+1][i]);
}

int tree_min (int vx, int xl, int xr, int txl, int txr, int vy, int yl, int yr, int tyl, int tyr) {
    if (xl == txl && xr == txr)
        if (yl == tyl && yr == tyr)
            return t[vx][vy];
        else {
            int tym = (tyl + tyr) >> 1;
            if (yr <= tym)
                return tree_min (vx, xl, xr, txl, txr, vy+vy, yl, yr, tyl, tym);
            else if (yl > tym)
                return tree_min (vx, xl, xr, txl, txr, vy+vy+1, yl, yr, tym+1, tyr);
            else
                return min (
                    tree_min (vx, xl, xr, txl, txr, vy+vy, yl, tym, tyl, tym),
                    tree_min (vx, xl, xr, txl, txr, vy+vy+1, tym+1, yr, tym+1, tyr)
                );
        }
    else {
        int txm = (txl + txr) >> 1;
        if (xr <= txm)
            return tree_min (vx+vx, xl, xr, txl, txm, vy, yl, yr, tyl, tyr);
        else if (xl > txm)
            return tree_min (vx+vx+1, xl, xr, txm+1, txr, vy, yl, yr, tyl, tyr);
        else
            return min (
                tree_min (vx+vx, xl, txm, txl, txm, vy, yl, yr, tyl, tyr),
                tree_min (vx+vx+1, txm+1, xr, txm+1, txr, vy, yl, yr, tyl, tyr)
            );
    }
}

int main() {
    ... чтение n, m, a ...
    build (1, 0, n-1, 1, 0, m-1);
    for (;;) {
        ... поступает запрос ...
        int x1, y1, x2, y2;
        --x1, --y1, --x2, --y2;
        printf ("%d\n", tree_min (1, x1, x2, 0, n-1, 1, y1, y2, 0, m-1));
    }
}

```

Декартово дерево (treap, дерамида)

Декартово дерево - это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) и дерамида (дерево+пирамида)).

Более строго, это структура данных, которая хранит пары (X, Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по X и бинарной пирамидой по Y . Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0, Y_0) , то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y < Y_0$.

Дерамиды были предложены Сиделем (Siedel) и Арагоном (Aragon) в 1996 г.

Преимущества такой организации данных

В том применении, которое мы рассматриваем (мы будем рассматривать дерамиды, поскольку декартово дерево - это фактически более общая структура данных), X 'ы являются ключами (и одновременно значениями, хранящимися в структуре данных), а Y 'и - называются **приоритетами**. Если бы приоритетов не было, то было бы обычное бинарное дерево поиска по X , и заданному набору X 'ов могло бы соответствовать много деревьев, некоторые из которых являются вырожденными (например, в виде цепочки), а потому чрезвычайно медленными (основные операции выполнялись бы за $O(N)$).

В то же время, **приоритеты позволяют однозначно** указать дерево, которое будет построено (разумеется, не зависящее от порядка добавления элементов) (это доказывается соответствующей теоремой). Теперь очевидно, что если **выбирать приоритеты случайно**, то этим мы добьёмся построения **невырожденных** деревьев в среднем случае, что обеспечит асимптотику $O(\log N)$ в среднем. Отсюда и

понятно ещё одно название этой структуры данных - **рандомизированное бинарное дерево поиска**.

Операции

Итак, treap предоставляет следующие операции:

- **Insert (X, Y)** - за $O(\log N)$ в среднем
Выполняет добавление в дерево нового элемента.
Возможен вариант, при котором значение приоритета Y не передаётся функции, а выбирается случайно (правда, нужно учесть, что оно не должно совпадать ни с каким другим Y в дереве).
- **Search (X)** - за $O(\log N)$ в среднем
Ищет элемент с указанным значением ключа X . Реализуется абсолютно так же, как и для обычного бинарного дерева поиска.
- **Erase (X)** - за $O(\log N)$ в среднем
Ищет элемент и удаляет его из дерева.
- **Build (X_1, \dots, X_N)** - за $O(N)$
Строит дерево из списка значений. Эту операцию можно реализовать за линейное время (в предположении, что значения X_1, \dots, X_N отсортированы), но здесь эта реализация рассматриваться не будет.
Здесь будет использоваться только простейшая реализация - в виде последовательных вызовов Insert, т.е. за $O(N \log N)$.
- **Union (T_1, T_2)** - за $O(M \log (N/M))$ в среднем
Объединяет два дерева, в предположении, что все элементы различны (впрочем, эту операцию можно реализовать с той же асимптотикой, если при объединении нужно удалять повторяющиеся элементы).
- **Intersect (T_1, T_2)** - за $O(M \log (N/M))$ в среднем
Находит пересечение двух деревьев (т.е. их общие элементы). Здесь реализация этой операции не будет рассматриваться.

Кроме того, за счёт того, что декартово дерево является бинарным деревом поиска по своим значениям, к нему применимы такие операции, как нахождение К-го по величине элемента, и, наоборот, определение номера элемента.

Описание реализации

С точки зрения реализации, каждый элемент содержит в себе X, Y и указатели на левого L и правого R сына.

Для реализации операций понадобится реализовать две вспомогательные операции: Split и Merge.

Split (T, X) - разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньшие по ключу X , а R содержит все элементы, большие X . Эта операция выполняется за $O(\log N)$. Реализация её довольно проста - очевидная рекурсия.

Merge (T_1, T_2) - объединяет два поддерева T_1 и T_2 , и возвращает это новое дерево. Эта операция также реализуется за $O(\log N)$. Она работает в предположении, что T_1 и T_2 обладают соответствующим порядком (все значения в первом меньше значений во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y . Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

Теперь очевидна реализация **Insert (X, Y)**. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по X), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше Y . Мы нашли позицию, куда будем вставлять наш элемент. Теперь вызываем Split (X) от найденного элемента (от элемента вместе со всем его поддеревом), и возвращаемое ею L и R записываем в качестве левого и правого сына добавляемого элемента.

Также понятна и реализация **Erase (X)**. Спускаемся по дереву (как в обычном бинарном дереве поиска по X), ища удаляемый элемент. Найдя элемент, мы просто вызываем Merge от его левого и правого сыновей, и возвращаемое ею значение ставим на место удаляемого элемента.

Операцию **Build** реализуем за $O(N \log N)$ просто с помощью последовательных вызовов Insert.

Наконец, операция **Union (T_1, T_2)**. Теоретически её асимптотика $O(M \log (N/M))$, однако на практике она работает очень хорошо, вероятно, с весьма малой скрытой константой. Пусть, не теряя общности, $T_1 \rightarrow Y > T_2 \rightarrow Y$, т.е. корень T_1 будет корнем результата. Чтобы получить результат, нам нужно объединить деревья $T_1 \rightarrow L, T_1 \rightarrow R$ и T_2 в два таких дерева, чтобы их можно было сделать сыновьями T_1 . Для этого вызовем Split ($T_2, T_1 \rightarrow X$), тем самым мы разобьём T_2 на две половинки L и R , которые затем рекурсивно объединим с сыновьями T_1 : Union ($T_1 \rightarrow L, L$) и Union ($T_1 \rightarrow R, R$), тем самым мы построим левое и правое поддеревья результата.

Реализация

Реализуем все описанные выше операции. Здесь для удобства введены другие обозначения - приоритет обозначается prior, значения - key.

```
struct item {
    int key, prior;
    item * l, * r;
    item() { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};

typedef item * pitem;

void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (key < t->key)
        split (t->l, key, l, t->r), r = t;
    else
        split (t->r, key, t->l, r), l = t;
}

void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r), t = it;
    else
        insert (t->r, it);
}
```

```

    insert (it->key < t->key ? t->l : t->r, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key)
        merge (t, t->l, t->r);
    else
        erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r) return l ? l : r;
    if (l->prior < r->prior) swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}

```

Поддержка размеров поддеревьев

Чтобы расширить функциональность декартового дерева, очень часто необходимо для каждой вершины хранить количество вершин в её поддереве - некое поле int cnt в структуре item. Например, с его помощью легко будет найти за $O(\log N)$ К-ый по величине элемент дерева, или, наоборот, за ту же асимптотику узнать номер элемента в отсортированном списке (реализация этих операций ничем не будет отличаться от их реализации для обычных бинарных деревьев поиска).

При изменении дерева (добавлении или удалении элемента и т.д.) должны соответствующим образом меняться и cnt некоторых вершин. Реализуем две функции - функция cnt() будет возвращать текущее значение cnt или 0, если вершина не существует, а функция upd_cnt() будет обновлять значение cnt для указанной вершины, при условии, что для её сыновей l и r эти cnt уже корректно обновлены. Тогда, понятно, достаточно добавить вызовы функции upd_cnt() в конец каждой из функций insert, erase, split, merge, чтобы постоянно поддерживать корректные значения cnt.

```

int cnt (pitem t) {
    return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}

```

Построение декартового дерева за $O(N)$ в оффлайн

TODO

Неявные декартовые деревья

Неявное декартово дерево - это простая модификация обычного декартового дерева, которая, тем не менее, оказывается очень мощной структурой данных. Фактически, неявное декартово дерево можно воспринимать как массив, над которым можно реализовать следующие операции (все за $O(\log N)$ в режиме онлайн):

- Вставка элемента в массив в любую позицию
- Удаление произвольного элемента
- Сумма, минимум/максимум на произвольном отрезке, и т.д.
- Прибавление, покраска на отрезке
- Переворот (перестановка элементов в обратном порядке) на отрезке

Ключевая идея заключается в том, что в качестве ключей key следует использовать **индексы** элементов в массиве. Однако явно хранить эти значения key мы не будем (иначе, например, при вставке элемента пришлось бы изменять key в $O(N)$ вершинах дерева).

Заметим, что фактически в данном случае ключ для какой-то вершины - это количество вершин, меньших неё. Следует заметить, что вершины, меньшие данной, находятся не только в её левом поддереве, но и, возможно, в левых поддеревьях её предков. Более строго, **неявный ключ** для некоторой вершины t равен количеству вершин cnt(t->l) в левом поддереве этой вершины плюс аналогичные величины cnt(p->l)+1 для каждого предка p этой вершины, при условии, что t находится в правом поддереве для p.

Ясно, как теперь быстро вычислять для текущей вершины её неявный ключ. Поскольку во всех операциях мы приходим в какую-либо вершину, спускаясь по дереву, мы можем просто накапливать эту сумму, передавая её функции. Если мы идём в левое поддерево - накапливаемая сумма не меняется, а если идём в правое - увеличивается на cnt(t->l)+1.

Приведём новые реализации функций split и merge:

```

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
}

```

```

    else
        merge (r->l, l, r->l), t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); // вычисляем неявный ключ
    if (key <= cur_key)
        split (t->l, l, t->l, key, add), r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
    upd_cnt (t);
}

```

Теперь перейдём к реализации различных дополнительных операций на неявных декартовых деревьях:

- **Вставка элемента.**

Пусть нам надо вставить элемент в позицию pos. Разобьём декартово дерево на две половинки: соответствующую массиву [0..pos-1] и массиву [pos..sz]; для этого достаточно вызвать `split (t, t1, t2, pos)`. После этого мы можем объединить дерево t1 с новой вершиной; для этого достаточно вызвать `merge (t1, t1, new_item)` (нетрудно убедиться в том, что все предусловия для `merge` выполнены). Наконец, объединим два дерева t1 и t2 обратно в дерево t - вызовом `merge (t, t1, t2)`.

- **Удаление элемента.**

Здесь всё ещё проще: достаточно найти удаляемый элемент, а затем выполнить `merge` для его сыновей l и r, и поставить результат объединения на место вершины t. Фактически, удаление из неявного декартова дерева не отличается от удаления из обычного декартова дерева.

- **Сумма/минимум** и т.п. на отрезке.

Во-первых, для каждой вершины создадим дополнительное поле f в структуре item, в котором будет храниться значение целевой функции для поддерева этой вершины. Такое поле легко поддерживать, для этого надо поступить аналогично поддержке размеров слт (создать функцию, вычисляющую значение этого поля, пользуясь его значениями для сыновей, и вставить вызовы этой функции в конце всех функций, меняющих дерево).

Во-вторых, нам надо научиться отвечать на запрос на произвольном отрезке [A;B]. Научимся выделять из дерева его часть, соответствующую отрезку [A;B]. Нетрудно понять, что для этого достаточно сначала вызвать `split (t, t1, t2, A)`, а затем `split (t2, t2, t3, B-A+1)`. В результате дерево t2 будет состоять из всех элементов в отрезке [A;B], и только них. Следовательно, ответ на запрос будет находиться в поле f вершины t2. После ответа на запрос дерево надо восстановить вызовами `merge (t, t1, t2)` и `merge (t, t, t3)`.

- **Прибавление/покраска** на отрезке.

Здесь мы поступаем аналогично предыдущему пункту, но вместо поля f будем хранить поле add, которое и будет содержать прибавляемую величину (или величину, которую красят всё поддерево этой вершины). Перед выполнением любой операции эту величину add надо "протолкнуть" - т.е. соответствующим образом изменить `t-l->add` и `t->r->add`, а у себя значение add снять. Тем самым мы добьёмся того, что ни при каких изменениях дерева информация не будет потеряна.

- **Переворот** на отрезке.

Этот пункт почти аналогичен предыдущему - нужно ввести поле bool rev, которое ставить в true, когда требуется произвести переворот в поддереве текущей вершины. "Проталкивание" поля rev заключается в том, что мы обмениваем местами сыновья текущей вершины, и ставим этот флаг для них.

Реализация. Приведём для примера полную реализацию неявного декартова дерева с переворотом на отрезке. Здесь для каждой вершины также хранится поле value - собственно значение элемента, стоящего в массиве на текущей позиции. Приведена также реализация функции `output()`, которая выводит массив, соответствующий текущему состоянию неявного декартова дерева.

```

typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap (it->l, it->r);
        if (it->l) it->l->rev ^= true;
        if (it->r) it->r->rev ^= true;
    }
}

void merge (pitem & t, pitem l, pitem r) {
    push (l);
    push (r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
    upd_cnt (t);
}

```

```

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add), r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
    upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-1+1);
    t2->rev ^= true;
    merge (t, t1, t2);
    merge (t, t, t3);
}

void output (pitem t) {
    if (!t) return;
    push (t);
    output (t->l);
    printf ("%d ", t->value);
    output (t->r);
}

```

Модификация стека и очереди для извлечения минимума за O (1)

Здесь мы рассмотрим три задачи: модифицирование стека с добавлением извлечения наименьшего элемента за O (1), аналогичное модифицирование очереди, а также применение их к задаче нахождения минимума во всех подотрезках фиксированной длины данного массива за O (N).

Модификация стека

Требуется добавить возможность извлечения минимума из стека за O (1), сохранив такой же асимптотику добавления и удаления элементов из стека.

Для этого будем хранить в стеке не сами элементы, а пары: элемент и минимум в стеке, начиная с этого элемента и ниже. Иными словами, если представить стек как массив пар, то

```

stack[i].second = min { stack[j].first }
j = 0..i

```

Понятно, что тогда нахождение минимума во всём стеке будет заключаться просто во взятии значения `stack.top().second`.

Также очевидно, что при добавлении нового элемента в стек величина `second` будет равна `min (stack.top().second, new_element)`. Удаление элемента из стека ничем не отличается от удаления из обычного стека, поскольку удаляемый элемент никак не мог повлиять на значения `second` для оставшихся элементов.

Реализация:

```
stack< pair<int,int> > st;
```

- Добавление элемента:

```

int minima = st.empty() ? new_element : min (new_element, st.top().second);
st.push (make_pair (new_element, minima));

```

- Извлечение элемента:

```

int result = st.top().first;
st.pop();

```

- Нахождение минимума:

```

minima = st.top().second;

```

Модификация очереди. Способ 1

Здесь рассмотрим простой способ модификации очереди, но имеющий тот недостаток, что модифицированная очередь реально может хранить не все элементы (т.е. при извлечении элемента из очереди нам надо будет знать значение элемента, который мы хотим извлечь). Ясно, что это весьма специфичная ситуация (обычно очередь нужна как раз для того, чтобы узнавать очередной элемент, а не наоборот), однако этот способ привлекателен своей простотой. Также этот метод применим к задаче о нахождении минимума в подотрезках (см. ниже).

Ключевая идея заключается в том, чтобы реально хранить в очереди не все элементы, а только нужные нам для определения минимума. А именно, пусть очередь представляет собой неубывающую последовательность чисел (т.е. в голове хранится наименьшее значение), причём, разумеется, не произвольную, а всегда содержащую минимум. Тогда минимум во всей очереди всегда будет являться первым её элементом. Перед добавлением нового элемента в очередь достаточно произвести "резку": пока в хвосте очереди находится элемент, больший нового элемента, будем удалять этот элемент из очереди; затем добавим новый элемент в конец очереди. Тем самым мы, с одной стороны, не нарушим порядка, а с другой стороны, не потеряем текущий элемент, если он на каком-либо последующем шаге окажется минимумом. Но при извлечении элемента из головы очереди его там, вообще говоря, может уже не оказаться - наша модифицированная очередь могла выкинуть этот элемент в процессе перестройки. Поэтому при удалении элемента нам надо знать значение извлекаемого элемента - если элемент с этим значением находится в голове очереди, то извлекаем его; иначе просто ничего не делаем.

Рассмотрим реализацию вышеописанных операций:

```
deque<int> q;
```

- Нахождение минимума:

```
    current_minimum = q.front();
```

- Добавление элемента:

```
while (!q.empty() && q.back() > added_element)
    q.pop_back();
q.push_back (added_element);
```

- Извлечение элемента:

```
if (!q.empty() && q.front() == removed_element)
    q.pop_front();
```

Понятно, что в среднем время выполнения всех этих операций есть $O(1)$.

Модификация очереди. Способ 2

Рассмотрим здесь другой способ модификации очереди для извлечения минимума за $O(1)$, который несколько более сложен для реализации, однако лишён основного недостатка предыдущего метода: все элементы очереди реально сохраняются в ней, и, в частности, при извлечении элемента не требуется знать его значение.

Идея заключается в том, чтобы свести задачу к задаче на стеках, которая уже была нами решена. Научимся моделировать очередь с помощью двух стеков.

Заведём два стека: s_1 и s_2 ; разумеется, имеются в виду стеки, модифицированные для нахождения минимума за $O(1)$. Добавлять новые элементы будет всегда в стек s_1 , а извлекать элементы - только из стека s_2 . При этом, если при попытке извлечения элемента из стека s_2 он оказался пустым, просто перенесём все элементы из стека s_1 в стек s_2 (при этом элементы в стеке s_2 получатся уже в обратном порядке, что нам и нужно для извлечения элементов; стек s_1 же станет пустым). Наконец, нахождение минимума в очереди будет фактически заключаться в нахождении минимума из минимума в стеке s_1 и минимума в стеке s_2 .

Тем самым, мы выполняем все операции по-прежнему за $O(1)$ (по той простой причине, что каждый элемент в худшем случае 1 раз добавляется в стек s_1 , 1 раз переносится в стек s_2 и 1 раз извлекается из стека s_2).

Реализация:

```
stack< pair<int,int> > s1, s2;
```

- Нахождение минимума:

```
if (s1.empty() || s2.empty())
    current_minimum = s1.empty ? s2.top().second : s1.top().second;
else
    current_minimum = min (s1.top().second, s2.top().second);
```

- Добавление элемента:

```
int minima = s1.empty() ? new_element : min (new_element, s1.top().second);
s1.push (make_pair (new_element, minima));
```

- Извлечение элемента:

```
if (s2.empty())
    while (!s1.empty()) {
        int element = s1.top().first;
        s1.pop();
        int minima = s2.empty() ? element : min (element, s2.top().second);
        s2.push (make_pair (element, minima));
    }
result = s2.top().first;
s2.pop();
```

Задача нахождения минимума во всех подотрезках фиксированной длины данного массива

Пусть дан массив A длины N, и дано число M ≤ N. Требуется найти минимум в каждом подотрезке длины M данного массива, т.е. найти:

$$\min_{0 \leq i \leq M-1} A[i], \quad \min_{1 \leq i \leq M} A[i], \quad \min_{2 \leq i \leq M+1} A[i], \quad \dots, \quad \min_{N-M \leq i \leq N-1} A[i]$$

Решим эту задачу за линейное время, т.е. O(N).

Для этого достаточно завести очередь, модифицированную для нахождения минимума за O(1), что было рассмотрено нами выше, причём в данной задаче подойдёт любой из двух методов реализации такой очереди. Далее решение уже понятно: добавим в очередь первые M элементов массива, найдём в ней минимум и выведем его, затем добавим в очередь следующий элемент, и извлечём из неё первый элемент массива, снова выведем минимум, и т.д. Поскольку все операции с очередью выполняются в среднем за константное время, то и асимптотика всего алгоритма получится O(N).

Стоит заметить, что реализация модифицированной очереди первым методом проще, однако для неё, вероятно, потребуется хранить весь массив (поскольку на i-ом шаге потребуется знать i-ый и (i-M)-ый элементы массива). При реализации очереди вторым методом массив A хранить явно не понадобится - только узнавать очередной, i-ый элемент массива.

Рандомизированная куча

Рандомизированная куча (randomized heap) — это куча, которая за счёт применения генератора случайных чисел позволяет выполнять все необходимые операции за логарифмическое ожидаемое время.

Кучей называется бинарное дерево, для любой вершины которого справедливо, что значение в этой вершине меньше либо равно значений во всех её потомках (это куча для минимума; разумеется, симметрично можно определить кучу для максимума). Таким образом, в корне кучи всегда находится минимум.

Стандартный набор операций, определяемый для куч, следующий:

- Добавление элемента
- Нахождение минимума
- Извлечение минимума (удаление его из дерева и возврат его значения)
- Слияние двух куч (возвращается куча, содержащая элементы обеих куч; дубликаты не удаляются)
- Удаление произвольного элемента (при известной позиции в дереве)

Рандомизированная куча позволяет выполнять все эти операции за ожидаемое время $O(\log n)$ при очень простой реализации.

Структура данных

Сразу опишем структуру данных, описывающую бинарную кучу:

```
struct tree {
    T value;
    tree * l, * r;
};
```

В вершине дерева хранится значение `value` некоторого типа `T`, для которого определён оператор сравнения (operator `<`). Кроме того, хранятся указатели на левого и правого сыновей (которые равны 0, если соответствующий сын отсутствует).

Выполнение операций

Нетрудно понять, что все операции над кучей сводятся к одной операции: **слиянию** двух куч в одну. Действительно, добавление элемента в кучу равносильно слиянию этой кучи с кучей, состоящей из единственного добавляемого элемента. Нахождение минимума вообще не требует никаких действий — минимумом просто является корень кучи. Извлечение минимума эквивалентно тому, что куча заменяется результатом слияния левого и правого поддерева корня. Наконец, удаление произвольного элемента аналогично удалению минимума: всё поддерево с корнем в этой вершине заменяется результатом слияния двух поддеревьев-сыновей этой вершины.

Итак, нам фактически надо реализовать только операцию слияния двух куч, все остальные операции тривиально сводятся к этой операции.

Пусть даны две кучи T_1 и T_2 , требуется вернуть их объединение. Понятно, что в корне каждой из этих куч находятся их минимумы, поэтому в корне результирующей кучи будет находиться минимум из этих двух значений. Итак, мы сравниваем, в корне какой из куч находится меньшее значение, его помещаем в корень результата, а теперь мы должны объединить сыновей выбранной вершины с оставшейся кучей. Если мы по какому-то признаку выберем одного из двух сыновей, то тогда нам надо будет просто объединить поддерево в корне с этим сыном с кучей. Таким образом, мы снова пришли к операции слияния. Рано или поздно этот процесс остановится (на это понадобится, понятно, не более чем сумма высот куч).

Таким образом, чтобы достичь логарифмической асимптотики в среднем, нам надо указать способ выбора одного из двух сыновей с тем, чтобы в среднем длина проходимого пути получалась бы порядка логарифма от количества элементов в куче. Нетрудно догадаться, что производить этот выбор мы будем **случайно**, таким образом, реализация операции слияния получается такой:

```
tree * merge (tree * t1, tree * t2) {
```

```

if (!t1 || !t2)
    return t1 ? t1 : t2;
if (t2->value < t1->value)
    swap (t1, t2);
if (rand() & 1)
    swap (t1->l, t1->r);
t1->l = merge (t1->l, t2);
return t1;
}

```

Здесь сначала проверяется, если хотя бы одна из куч пуста, то никаких действий по слиянию производить не надо. Иначе, мы делаем, чтобы куча t_1 была кучей с меньшим значением в корне (для чего обмениваем t_1 и t_2 , если надо). Наконец, мы считаем, что вторую кучу t_2 будем сливать с левым сыном корня кучи t_1 , поэтому мы случайным образом обмениваем левого и правого сыновей, а затем выполняем слияние левого сына и второй кучи.

Асимптотика

Введём случайную величину $h(T)$, обозначающую **длину случайного пути** от корня до листа (длина в числе рёбер). Понятно, что алгоритм `merge` выполняется за $O(h(T_1) + h(T_2))$ операций. Поэтому для исследования асимптотики алгоритма надо исследовать случайную величину $h(T)$.

Математическое ожидание

Утверждается, что математическое ожидание $h(T)$ оценивается сверху логарифмом от числа n вершин в этой куче:

$$Eh(T) \leq \log(n+1)$$

Доказывается это легко по индукции. Пусть L и R — соответственно левое и правое поддеревья корня кучи T , а n_L и n_R — количества вершин в них (понятно, что $n = n_L + n_R + 1$).

Тогда справедливо:

$$\begin{aligned} Eh(T) &= 1 + \frac{1}{2}(Eh(L) + Eh(R)) \leq 1 + \frac{1}{2}(\log(n_L + 1) + \log(n_R + 1)) = \\ &= 1 + \log \sqrt{(n_L + 1)(n_R + 1)} = \log 2\sqrt{(n_L + 1)(n_R + 1)} \leq \\ &\leq \log \frac{2((n_L + 1) + (n_R + 1))}{2} = \log(n_L + n_R + 2) = \log(n + 1) \end{aligned}$$

что и требовалось доказать.

Превышение ожидаемой оценки

Докажем, что вероятность превышения полученной выше оценки мала:

$$P\{h(T) > (c+1) \log n\} < \frac{1}{n^c}$$

для любой положительной константы c .

Обозначим через P множество путей от корня кучи до листьев, длина которых превосходит $(c+1) \log n$. Заметим, что для любого пути p длины $|p|$ вероятность того, в качестве случайного пути будет выбран именно он, равна $2^{-|p|}$. Тогда получаем:

$$P\{h(T) > (c+1) \log n\} = \sum_{p \in P} 2^{-|p|} < \sum_{p \in P} 2^{-(c+1) \log n} = |P|n^{-(c+1)} \leq n^{-c}$$

что и требовалось доказать.

Асимптотика алгоритма

Таким образом, алгоритм `merge`, а, значит, и все остальные выраженные через него операции, выполняется за $O(\log n)$ в среднем.

Более того, для любой положительной константы ϵ найдётся такая положительная константа c , что вероятность того, что операция потребует больше чем $c \log n$ операций, меньше $n^{-\epsilon}$ (это в некотором смысле описывает худшее поведение алгоритма).

Задача RMQ (Range Minimum Query - минимум на отрезке)

Дан массив $A[1..N]$. Поступают запросы вида (L, R) , на каждый запрос требуется найти минимум в массиве A , начиная с позиции L и заканчивая позицией R .

Приложения

Помимо непосредственного применения в самых разных задачах, можно отметить следующие:

- Задача LCA (наименьший общий предок)

Решение

Задача RMQ решается с помощью структур данных.

Из описанных на сайте структур данных можно выбрать:

- **Sqrt-декомпозиция** - отвечает на запрос за $O(\sqrt{N})$, препроцессинг за $O(N)$.
Преимущество в том, что это очень простая структура данных. Недостаток - асимптотика.
- **Дерево отрезков** - отвечает на запрос за $O(\log N)$, препроцессинг за $O(N)$.
Преимущество - хорошая асимптотика. Недостаток - больший объём кода по сравнению с другими структурами данных.
- **Дерево Фенвика** - отвечает на запрос за $O(\log N)$, препроцессинг за $O(N \log N)$
Преимущество - очень быстро пишется и работает тоже очень быстро. Но значительный недостаток - дерево Фенвика может отвечать только на запросы с $L = 1$, что для многих приложений неприменимо.

Примечание. "Препроцессинг" - это предварительная обработка массива A , фактически это построение структуры данных для данного массива.

Теперь предположим, что массив A **может изменяться** в процессе работы (т.е. также будут поступать запросы об изменении значения в некотором отрезке $[L; R]$). Тогда полученную задачу можно решить с помощью **Sqrt-декомпозиции** и **Дерева отрезков**.

Длиннейшая возрастающая подпоследовательность за $O(N^2)$

Это одна из основных задач на динамическое программирование.

Нередко задачу именуют как LIS (longest increasing subsequence).

Дана последовательность $M[1..N]$. Строим таблицу $A[1..N]$. Каждый её элемент A_i - длина длиннейшей возрастающей подпоследовательности, оканчивающейся точно в позиции i . Если мы построим эту таблицу, то ответ к задаче - наибольшее число из этой таблицы.

Само построение тоже элементарно: $A_i = \max(A_j + 1)$, среди всех $j < i$, для которых $M_j < M_i$. Первый элемент $A_1 = 1$, что очевидно.

Таким образом, алгоритм работает за $O(N^2)$.

Реализация

```
int main()
{
    vector<int> m; // последовательность
    int n; // её длина
    ... // считываем последовательность

    vector<int> a (n, 1); // таблица длин
    a[0] = 1;
    vector<int> pred (n, -1); // таблица предков, если надо вывести и саму подпоследовательность
    for (int i=1; i<n; i++)
        for (int j=0; j<i; j++)
            if (m[j] < m[i])
                if (a[j]+1 > a[i])
                {
                    a[i] = a[j]+1;
                    pred[i] = j;
                }

    // выводим длину последовательности
    cout << * max_element (a.begin(), a.end());

    // ищем и выводим саму подпоследовательность
    vector<int> result;
    for (int cur = int (max_element (a.begin(), a.end()) - a.begin()); cur != -1; cur = pred[cur])
        result.push_back (m[cur]);
    cout << endl;
    for (unsigned i=result.size(); i-- > 0; )
        cout << result[i] << ' ';
}
```

Длиннейшая возрастающая подпоследовательность за $O(N \log N)$

Здесь мы рассмотрим улучшенный алгоритм, который работает за более быструю асимптотику, чем описанный [здесь](#). Мы также будем применять метод динамического программирования, правда, немного по-другому поставим задачу.

Пусть по условию дан массив чисел A длиной N . Вычислим массив $D[0..N]$, каждый элемент которого $D[i]$ есть число, на которое оканчивается возрастающая подпоследовательность длины i (если таких чисел несколько, то берётся наименьшее). Очевидно, этот массив можно легко вычислить за $O(N^2)$:

```
// O (N2)
vector<int> d (n+1, INF);
d[0] = -INF;
for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
        if (d[j] < a[i] && a[i] < d[j+1])
            d[j+1] = a[i];
```

Однако теперь легко догадаться, как ускорить вычисление массива D - нужно использовать бинарный поиск. Действительно, вместо того, чтобы перебирать все возможные индексы j , мы воспользуемся очевидным свойством $D[i] \leq D[i+1]$ и применим бинарный поиск (в данном случае `upper_bound`):

```
// O (N log N)
vector<int> d (n+1, INF);
d[0] = -INF;
for (int i=0; i<n; i++)
{
    unsigned j = upper_bound (d.begin(), d.end(), a[i]) - d.begin() - 1;
    if (d[j] < a[i] && a[i] < d[j+1])
        d[j+1] = a[i];
}
```

Если теперь при обновлении массива D сохранять в дополнительных массивах предков, то становится легко найти саму подпоследовательность:

```
// O (N log N)
vector<int> d (n+1, INF);
d[0] = -INF;
vector<int> p (n);
vector<int> no (n+1);
no[0] = -1;
for (int i=0; i<n; i++)
{
    unsigned j = upper_bound (d.begin(), d.end(), a[i]) - d.begin() - 1;
    if (d[j] < a[i] && a[i] < d[j+1])
    {
        d[j+1] = a[i];
        p[i] = no[j];
        no[j+1] = i;
    }
}
vector<int> result;
for (int i=n; i>=1; i--)
    if (d[i] != INF)
    {
        for (int cur=no[i]; cur!=-1; cur=p[cur])
            result.push_back (cur);
        break;
    }
reverse (result.begin(), result.end());
for (unsigned i=0; i<result.size(); i++)
    cout << a[result[i]] << ", ";
```

К-ая порядковая статистика за O (N)

Пусть дан массив A длиной N и пусть дано число K. Задача заключается в том, чтобы найти в этом массиве K-ое по величине число, т.е. К-ую порядковую статистику.

Основная идея - использовать идеи алгоритма быстрой сортировки. Собственно, алгоритм несложный, сложнее доказать, что он работает в среднем за O (N), в отличие от быстрой сортировки.

Реализация в виде нерекурсивной функции:

```
template <class T>
T order_statistics (std::vector<T> a, unsigned n, unsigned k)
{
    using std::swap;
    for (unsigned l=1, r=n; ; )
    {

        if (r <= l+1)
        {
            // текущая часть состоит из 1 или 2 элементов -
            // легко можем найти ответ
            if (r == l+1 && a[r] < a[l])
                swap (a[l], a[r]);
            return a[k];
        }

        // упорядочиваем a[l], a[l+1], a[r]
        unsigned mid = (l + r) >> 1;
        swap (a[mid], a[l+1]);
        if (a[l] > a[r])
            swap (a[l], a[r]);
        if (a[l+1] > a[r])
            swap (a[l+1], a[r]);
        if (a[l] > a[l+1])
            swap (a[l], a[l+1]);

        // выполняем разделение
        // барьером является a[l+1], т.е. медиана среди a[l], a[l+1], a[r]
        unsigned
            i = l+1,
            j = r;
        const T
            cur = a[l+1];
        for (;;)
        {
            while (a[++i] < cur) ;
            while (a[--j] > cur) ;
            if (i > j)
                break;
            swap (a[i], a[j]);
        }

        // вставляем барьер
        a[l+1] = a[j];
        a[j] = cur;

        // продолжаем работать в той части,
        // которая должна содержать искомый элемент
        if (j >= k)
            r = j-1;
        if (j <= k)
            l = i;
    }
}
```

Следует заметить, что в стандартной библиотеке C++ этот алгоритм уже реализован - он называется `nth_element`.

Динамика по профилю. Задача "паркет"

Типичными задачами на динамику по профилю, являются:

- найти количество способов замощения поля некоторыми фигурами
- найти замощение с наименьшим количеством фигур
- найти замощение с минимальным количеством неиспользованных клеток
- найти замощение с минимальным количеством фигур такое, что в него нельзя добавить ещё одну фигуру

Задача "Паркет"

Имеется прямоугольная площадь размером NxM. Нужно найти количество способов замостить эту площадь фигурами 1x2 (пустых клеток не должно оставаться, фигуры не должны накладываться друг на друга).

Построим такую динамику: $D[l][Mask]$, где $l=1..N$, $Mask=0..2^M-1$. l обозначает количество строк в текущем поле, а $Mask$ - профиль последней строки в текущем поле. Если j -й бит в $Mask$ равен нулю, то в этом месте профиль проходит на "нормальном уровне", а если 1 - то здесь "выемка" глубиной 1. Ответом, очевидно, будет $D[N][0]$.

Строить такую динамику будем, просто перебирая все $l=1..N$, все маски $Mask=0..2^M-1$, и для каждой маски будем делать переходы вперёд, т.е. добавлять к ней новую фигуру всеми возможными способами.

Реализация:

```
int n, m;
vector < vector<long long> > d;

void calc (int x = 0, int y = 0, int mask = 0, int next_mask = 0)
{
    if (x == n)
        return;
    if (y >= m)
        d[x+1][next_mask] += d[x][mask];
    else
    {
        int my_mask = 1 << y;
        if (mask & my_mask)
            calc (x, y+1, mask, next_mask);
        else
        {
            calc (x, y+1, mask, next_mask | my_mask);
            if (y+1 < m && ! (mask & my_mask) && ! (mask & (my_mask << 1)))
                calc (x, y+2, mask, next_mask);
        }
    }
}

int main()
{
    cin >> n >> m;

    d.resize (n+1, vector<long long> (1<<m));
    d[0][0] = 1;
    for (int x=0; x<n; ++x)
        for (int mask=0; mask<(1<<m); ++mask)
            calc (x, 0, mask, 0);

    cout << d[n][0];
}
```

Нахождение наибольшей нулевой подматрицы за $O(NM)$

Дана матрица А размером $N \times M$, состоящая только из нулей и единиц. Требуется найти в ней наибольшую (по площади) подматрицу, состоящую только из нулей.

Здесь будет описано решение, работающее за линейное относительно размера матрицы время: $O(NM)$.

Описание

Пусть дана матрица A , имеющая N строк и M столбцов.

Будем последовательно двигаться по строчкам этой матрицы; пусть текущая строка - i . Посчитаем для всех элементов текущей строки величину $D[1..M]$, где $D[j]$ равно наибольшему номеру строки $\leq i$, в которой в j -ом столбце стоит единица; если такой строки нет, то полагаем $D[j]$ равным -1 . Иными словами, $D[j]$ указывает для элемента $A[i][j]$ ближайшую сверху единицу. (В частности, когда $A[i][j] = 1$, то $D[j] = i$)

Такую динамику легко посчитать для каждой строчки, если знать её значение для предыдущей строки.

Например, вот код, вычисляющий значения этой динамики для всех элементов матрицы:

```
vector<int> d (m, -1);
for (int i=0; i<n; ++i) {
    for (int j=0; j<m; ++j)
        if (a[i][j] == 1)
            d[j] = i;
    // вычислили для i-ой строки, можем использовать эти значения
}
```

Значения $D[j]$ позволят нам определить верхнюю границу для нулевой подматрицы, если мы будем ставить её так, чтобы её нижняя граница находилась в строке i .

Уже сейчас мы можем решить задачу за $O(NM^2)$ - просто перебирать в текущей строке левую и правую границы, и с помощью динамики $D[j]$ вычислять за $O(1)$ верхнюю границу нулевой подматрицы. Однако можно пойти дальше и значительно улучшить асимптотику.

Ясно, что наибольшая нулевая подматрица ограничена сверху как минимум одной единицей в некотором столбце, также она ограничена единицами слева и справа (можно считать, что за пределами матрицы A стоят одни единицы). Поэтому мы не пропустим наилучший ответ, если будем действовать следующим образом: в текущей строке i будем перебирать столбец j ; верхней границей нулевой подматрицы будем считать $D[j]$; для текущего j найдём ближайшую слева позицию k_1 такую, что $D[k_1] > D[j]$ (т.е. левая граница нулевой подматрицы), а также позицию k_2 такую, что $D[k_2] > D[j]$ (правую границу). Тогда величиной текущей нулевой подматрицы будет $(i - D[j]) * (D[k_2] - D[k_1] - 1)$. И из всех таких матриц надо будет выбрать наибольшую по площади, она и будет ответом.

Осталось научиться эффективно вычислять позиции k_1 и k_2 для текущего j , а именно, вычислять за $O(1)$. Для этого посчитаем две соответствующие динамики D_1 и D_2 .

Заведём стек St . Будем двигаться в текущей строке по столбцам j от 1 до M . Будем поддерживать инвариант, что в стеке St находятся номера столбцов, в которых величина D меньше $D[j]$. Очевидно, что при переходе от одного элемента к следующему мы должны просто удалить с вершины стека все столбцы, величина D в которых $\geq D[j]$, затем присвоить $D_1[j] = St.top$, и затем положить в стек j . Аналогично построим и вторую динамику D_2 , просто двигаться справа налево: j от M до 1. Ясно, что на вычисление каждой динамики потребуется $O(M)$ времени.

Итак, схема алгоритма такова: перебираем строку i , для каждой строки последовательно считаем динамики D , D_1 , D_2 , и проходом по всем столбцам j строим нулевые подматрицы, из которых выбираем наибольшую.

Реализация

Реализация алгоритма, которая считывает матрицу и выводит размер наибольшей нулевой подматрицы. Ясно, что добиться вывода самих координат подматрицы несложно.

```
int n, m;
cin >> n >> m;
vector < vector<char> > a (n, vector<char> (m));
for (int i=0; i<n; ++i)
    for (int j=0; j<m; ++j)
        cin >> a[i][j];

int ans = 0;
vector<int> d (m, -1);
vector<int> dl (m), dr (m);
stack<int> st;
for (int i=0; i<n; ++i) {
    for (int j=0; j<m; ++j)
        if (a[i][j] == 1)
            d[j] = i;
    while (!st.empty()) st.pop();
    for (int j=0; j<m; ++j) {
        while (!st.empty() && d[st.top()] <= d[j]) st.pop();
        dl[j] = st.empty() ? -1 : st.top();
        st.push (j);
    }
    while (!st.empty()) st.pop();
    for (int j=m-1; j>=0; --j) {
        while (!st.empty() && d[st.top()] <= d[j]) st.pop();
        dr[j] = st.empty() ? m : st.top();
        st.push (j);
    }
    for (int j=0; j<m; ++j)
        ans = max (ans, (i - d[j]) * (dr[j] - dl[j] - 1));
}
cout << ans;
```

Вычисление определителя методом Краута за $O(N^3)$

Здесь будет рассмотрена модификация метода Краута (Crout), позволяющая вычислить определитель матрицы за $O(N^3)$.

Собственно алгоритм Краута находит разложение матрицы A в виде $A = L U$, где L - нижняя, а U - верхняя треугольная матрицы. Без ограничения общности можно считать, что в L все диагональные элементы равны 1. Но, зная эти матрицы, легко вычислить определитель A : он будет равен произведению всех элементов, стоящих на главной диагонали матрицы U .

Имеется теорема, согласно которой любая обратимая матрица обладает LU-разложением, и притом единственным, тогда и только тогда, когда все её главные миноры отличны от нуля. Следует напомнить, что мы рассматриваем только такие разложения, в которых диагональ L состоит только из единиц; иначе же, вообще говоря, разложение не единственno.

Пусть A - матрица, N - её размер. Мы найдём элементы матриц L и U .

Сам алгоритм состоит из следующих шагов:

1. Положим $L_{ii} = 1$ для $i = 1, 2, \dots, N$
2. Для каждого $j = 1, 2, \dots, N$ выполним:
 1. Для $i = 1, 2, \dots, j$ найдём значение U_{ij} :
$$U_{ij} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}$$
где сумма по всем $k = 1, 2, \dots, i-1$.
 2. Далее, для $i = j+1, j+2, \dots, N$ имеем:
$$L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}) / U_{jj}$$
где сумма берётся по всем $k = 1, 2, \dots, j-1$.

Реализация

Код на Java (с использованием дробной длинной арифметики):

```
static BigInteger det (BigDecimal a [][], int n)
{
    try {
        for (int i=0; i<n; i++)
        {
            boolean nonzero = false;
            for (int j=0; j<n; j++)
                if (a[i][j].compareTo (new BigDecimal (BigInteger.ZERO)) > 0)
                    nonzero = true;
            if (!nonzero)
                return BigInteger.ZERO;
        }

        BigDecimal scaling [] = new BigDecimal [n];
        for (int i=0; i<n; i++)
        {
            BigDecimal big = new BigDecimal (BigInteger.ZERO);
            for (int j=0; j<n; j++)
                if (a[i][j].abs().compareTo (big) > 0)
                    big = a[i][j].abs();
            scaling[i] = (new BigDecimal (BigInteger.ONE)).divide
                (big, 100, BigDecimal.ROUND_HALF_EVEN);
        }

        int sign = 1;

        for (int j=0; j<n; j++)
        {

            for (int i=0; i<j; i++)
            {
                BigDecimal sum = a[i][j];
                for (int k=0; k<i; k++)
                    sum = sum.subtract (a[i][k].multiply (a[k][j]));
                a[i][j] = sum;
            }

            BigDecimal big = new BigDecimal (BigInteger.ZERO);
```

```

int imax = -1;
for (int i=j; i<n; i++)
{
    BigDecimal sum = a[i][j];
    for (int k=0; k<j; k++)
        sum = sum.subtract (a[i][k].multiply (a[k][j]));
    a[i][j] = sum;
    BigDecimal cur = sum.abs();
    cur = cur.multiply (scaling[i]);
    if (cur.compareTo (big) >= 0)
    {
        big = cur;
        imax = i;
    }
}

if (j != imax)
{

    for (int k=0; k<n; k++)
    {
        BigDecimal t = a[j][k];
        a[j][k] = a[imax][k];
        a[imax][k] = t;
    }

    BigDecimal t = scaling[imax];
    scaling[imax] = scaling[j];
    scaling[j] = t;

    sign = -sign;
}

if (j != n-1)
    for (int i=j+1; i<n; i++)
        a[i][j] = a[i][j].divide
            (a[j][j], 100, BigDecimal.ROUND_HALF_EVEN);

}

BigDecimal result = new BigDecimal (1);
if (sign == -1)
    result = result.negate();
for (int i=0; i<n; i++)
    result = result.multiply (a[i][i]);

return result.divide
    (BigDecimal.valueOf(1), 0, BigDecimal.ROUND_HALF_EVEN).toBigInteger();

}
catch (Exception e)
{
    return BigInteger.ZERO;
}
}

```

Метод Гаусса решения системы линейных уравнений

Дана система из N линейных уравнений с N неизвестными. Известно, что система имеет единственное решение (т.е. определитель её отличен от нуля). Требуется найти это решение.

Описание метода

Дана система:

$$\begin{aligned}
 a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n &= b_1 \\
 a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n &= b_2 \\
 \dots \\
 a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nn} x_n &= b_n
 \end{aligned}$$

Выполним следующий алгоритм.

На первом шаге найдём в первом столбце наибольший по модулю элемент, поставим уравнение с этим элементом на первую строчку (обменяв две соответствующие строки матрицы A и два соответствующих элемента вектора B), а затем будем отнимать это уравнение от всех остальных, чтобы в первом столбце все элементы (кроме первого) обратились в ноль. Например, при добавлении ко второй строке будем домножать первую строку на $-a_{21}/a_{11}$, при добавлении к третьей - на $-a_{31}/a_{11}$, и т.д.

На втором шаге найдём во втором столбце, начиная со второго элемента, наибольший по модулю элемент, поставим уравнение с этим элементом на вторую строчку, и будем отнимать это уравнение от всех остальных (в том числе и от первого), чтобы во втором столбце все элементы (кроме второго) обратились в ноль. Понятно, что эта операция никак не изменит первый столбец - ведь от каждой строки мы будем отнимать вторую строку, домноженную на некоторый коэффициент, а во второй строке в первом столбце стоит ноль.

Т.е. на i-ом шаге найдём в i-ом столбце, начиная с i-го элемента, наибольший по модулю элемент, поставим уравнение с этим элементом на i-ю строчку, и будем отнимать это уравнение от всех остальных. Понятно, что это никак не повлияет на все предыдущие столбцы (с первого по (i-1)-ый).

В конце концов, мы приведём систему к так называемому диагональному виду:

```
c11 x1 = d1
c22 x2 = d2
...
cnn xn = dn
```

Т.е. мы нашли решение системы.

Замечание 1. На каждой итерации найдётся хотя бы один ненулевой элемент, иначе система бы имела нулевой определитель, что противоречит условию.

Замечание 2. Требование, что на каждом шаге мы выбираем наибольший по модулю элемент, очень важно в смысле численной устойчивости метода. Если выбирать произвольный ненулевой элемент, то это может привести к гигантской погрешности, когда получившееся решение будет отличаться в разы от правильного.

Реализация

Для упрощения реализации матрица коэффициентов A и столбец свободных коэффициентов B будем хранить в одном векторе a.

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n, vector<double> (n+1));
... чтение n и a ...

for (int i=0; i<n; ++i) {
    int k = i;
    for (int j=i+1; j<n; ++j)
        if (abs (a[j][i]) > abs (a[k][i]))
            k = j;
    swap (a[i], a[k]);
    for (int j=i+1; j<=n; ++j)
        a[i][j] /= a[i][i];
    for (int j=0; j<n; ++j)
        if (j != i)
            for (int k=i+1; k<=n; ++k)
                a[j][k] -= a[i][k] * a[j][i];
}

for (int i=0; i<n; ++i)
    printf ("% .15lf\n", a[i][n]);
```

Применения

Помимо непосредственно решения систем, метод Гаусса находит применение и в других задачах:

- [Нахождение определителя матрицы](#)
- [Нахождение ранга матрицы](#)

Нахождение ранга матрицы

Ранг матрицы - это наибольшее число линейно независимых строк/столбцов матрицы. Ранг определён не только для квадратных матриц; пусть матрица прямоугольна и имеет размер NxM.

Также ранг матрицы можно определить как наибольший из порядков миноров матрицы, отличных от нуля.

Заметим, что если матрица квадратная и её определитель отличен от нуля, то ранг равен $N (= M)$, иначе он будет меньше. В общем случае, ранг матрицы не превосходит $\min(N, M)$.

Алгоритм

Искать ранг можно с помощью модифицированного [метода Гаусса](#). Будем выполнять абсолютно те же самые операции, что и при решении системы или нахождении её определителя, но если на каком-либо шаге в i -ом столбце среди невыбранных до этого строк нет ненулевых, то мы этот шаг пропускаем, а ранг уменьшаем на единицу (изначально ранг полагаем равным $\max(N, M)$). Иначе, если мы нашли на i -ом шаге строку с ненулевым элементом в i -ом столбце, то помечаем эту строку как выбранную, и выполняем обычные операции отнимания этой строки от остальных.

Реализация

```
const double EPS = 1E-9;

int rank = max(n,m);
vector<char> line_used (n);
for (int i=0; i<m; ++i) {
    int j;
    for (j=0; j<n; ++j)
        if (!line_used[j] && abs(a[j][i]) > EPS)
            break;
    if (j == n)
        --rank;
    else {
        line_used[j] = true;
        for (int p=i+1; p<m; ++p)
            a[j][p] /= a[j][i];
        for (int k=0; k<n; ++k)
            if (k != j && abs (a[k][i]) > EPS)
                for (int p=i+1; p<m; ++p)
                    a[k][p] -= a[j][p] * a[k][i];
    }
}
```

Вычисление определителя матрицы методом Гаусса

Пусть дана квадратная матрица A размером $N \times N$. Требуется вычислить её определитель.

Алгоритм

Воспользуемся идеями метода Гаусса решения систем линейных уравнений.

Будем выполнять те же самые действия, что и при решении системы линейных уравнений, исключив только деление текущей строки на $a[i][i]$ (точнее, само деление можно выполнять, но подразумевая, что число выносится за знак определителя). Тогда все операции, которые мы будем производить с матрицей, не будут изменять величину определителя матрицы, за исключением, быть может, знака (мы только обменяем местами две строки, что меняет знак на противоположный, или прибавляем одну строку к другой, что не меняет величину определителя).

Но матрица, к которой мы приходим после выполнения алгоритма Гаусса, является диагональной, и определитель её равен произведению элементов, стоящих на диагонали. Знак, как уже говорилось, будет определяться количеством обменов строк (если их нечётное, то знак определителя следует изменить на противоположный). Таким образом, мы можем с помощью алгоритма Гаусса вычислять определитель матрицы за $O(N^3)$.

Осталось только заметить, что если в какой-то момент мы не найдём в текущем столбце ненулевого элемента, то алгоритм следует остановить и вернуть 0.

Реализация

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n, vector<double> (n));
... чтение n и a ...

double det = 1;
for (int i=0; i<n; ++i) {
    int k = i;
    for (int j=i+1; j<n; ++j)
        if (abs (a[j][i]) > abs (a[k][i]))
            k = j;
```

```

if (abs (a[k][i]) < EPS) {
    det = 0;
    break;
}
swap (a[i], a[k]);
if (i != k)
    det = -det;
det *= a[i][i];
for (int j=i+1; j<=n; ++j)
    a[i][j] /= a[i][i];
for (int j=0; j<n; ++j)
    if (j != i && abs (a[j][i]) > EPS)
        for (int k=i+1; k<n; ++k)
            a[j][k] -= a[i][k] * a[j][i];
}

cout << det;

```

Интегрирование по формуле Симпсона

Требуется посчитать значение определённого интеграла:

$$\int_a^b f(x)dx$$

Решение, описываемое здесь, было опубликовано в одной из диссертаций **Томаса Симпсона** (Thomas Simpson) в 1743 г.

Формула Симпсона

Пусть N — некоторое натуральное число. Разобъём отрезок интегрирования $[a; b]$ на $2N$ равных частей:

$$x_i = ih, i = 0 \dots 2N,$$

$$h = \frac{b-a}{2N}$$

Теперь посчитаем интеграл отдельно на каждом из отрезков $[x_{2i-2}, x_{2i}], i = 1 \dots N$, а затем сложим все значения.

Итак, пусть мы рассматриваем очередной отрезок $[x_{2i-2}, x_{2i}], i = 1 \dots N$. Заменим функцию $f(x)$ на нём параболой, проходящей через 3 точки $(x_{2i-2}, x_{2i-1}, x_{2i})$. Такая парабола всегда существует и единственна. Её можно найти аналитически, затем останется только проинтегрировать выражение для неё, и окончательно получаем:

$$\int_{x_{2i-2}}^{x_{2i}} f(x)dx = (f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})) \frac{h}{3}$$

Складывая эти значения по всем отрезкам, получаем окончательную формулу Симпсона:

$$\int_a^b f(x)dx = (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 4f(x_{2N-1}) + f(x_{2N})) \frac{h}{3}$$

Погрешность

Погрешность, даваемая формулой Симпсона, не превосходит по модулю величины:

$$\frac{1}{180} h^4 (b-a) \max_{a \leq x \leq b} |f^{(4)}(x)|$$

Таким образом, погрешность имеет порядок уменьшения как $O(N^4)$.

Реализация

Здесь $f(x)$ — некоторая пользовательская функция.

```

double a, b; // входные данные
const int N = 1000*1000; // количество шагов (уже умноженное на 2)
double s = 0;
for (int i=0; i<=N; ++i) {
    double x = a + (b - a) * i / N;
    s += f(x) * (i==0 || i==N ? 1 : (i&1)==0 ? 2 : 4);
}

```

```
    }
    double delta = (b - a) / N;
    s *= delta / 3.0;
```

Метод Ньютона (касательных) для поиска корней

Это итерационный метод, изобретённый **Исааком Ньютоном (Isaak Newton)** около 1664 г. Впрочем, иногда этот метод называют методом Ньютона-Рафсона (Raphson), поскольку Рафсон изобрёл тот же самый метод на несколько лет позже Ньютона, однако его статья была опубликована намного раньше.

Задача заключается в следующем. Дано уравнение:

$$F(X) = 0$$

Требуется решить это уравнение, точнее, найти один из его корней (предполагается, что корень существует). Предполагается, что $F(X)$ непрерывна и дифференцируема на отрезке $[A;B]$.

Алгоритм

Входным параметром алгоритма, кроме функции $F(X)$, является также **начальное приближение** - некоторое X_0 , от которого алгоритм начинает идти.

Пусть уже вычислено X_i , вычислим X_{i+1} следующим образом. Проведём касательную к графику функции $F(X)$ в точке $X = X_i$, и найдём точку пересечения этой касательной с осью абсцисс. X_{i+1} положим равным найденной точке, и повторим весь процесс с начала.

Нетрудно получить следующее выражение:

$$X_{i+1} = X_i - F(X_i) / F'(X_i)$$

Интуитивно ясно, что если функция $F(X)$ достаточно "хорошая", а X_i находится достаточно близко от корня, то X_{i+1} будет находиться ещё ближе к искомому корню.

Скорость сходимости является **квадратичной**, что, условно говоря, означает, что число точных разрядов в приближенном значении X_i удваивается с каждой итерацией.

Применение для вычисления квадратного корня

Рассмотрим метод Ньютона на примере вычисления квадратного корня.

Если подставить $F(X) = \sqrt{X}$, то после упрощения выражения получаем:

$$X_{i+1} = (X_i + N / X_i) / 2$$

Первый типичный вариант задачи - когда дано дробное число N , и нужно подсчитать его корень с некоторой точностью EPS :

```
double n;
cin >> n;
const double EPS = 1E-15;
double x = 1;
for (;;) {
    double nx = (x + n / x) / 2;
    if (abs(x - nx) < EPS) break;
    x = nx;
}
printf ("% .15lf", x);
```

Другой распространённый вариант задачи - когда требуется посчитать целочисленный корень (для данного N найти наибольшее X такое, что $X^2 \leq N$). Здесь приходится немного изменять условие останова алгоритма, поскольку может случиться, что X начнёт "скакать" возле ответа. Поэтому мы добавляем условие, что если значение X на предыдущем шаге уменьшилось, а на текущем шаге пытается увеличиться, то алгоритм надо остановить.

```
int n;
cin >> n;
int x = 1;
bool decreased = false;
for (;;) {
    int nx = (x + n / x) >> 1;
    if (x == nx || nx > x && decreased) break;
    decreased = nx < x;
```

```

x = nx;
}
cout << x;

```

Наконец, приведём ещё третий вариант - для случая длинной арифметики. Поскольку число N может быть достаточно большим, то имеет смысл обратить внимание на начальное приближение. Очевидно, что чем оно ближе к корню, тем быстрее будет достигнут результат. Достаточно простым и эффективным будет брать в качестве начального приближения число $2^{\text{bits}/2}$, где bits - количество битов в числе N. Вот код на языке Java, демонстрирующий этот вариант:

```

BigInteger n; // входные данные

BigInteger a = BigInteger.ONE.shiftLeft (n.bitLength() / 2);
boolean p_dec = false;
for (;;) {
    BigInteger b = n.divide(a).add(a).shiftRight(1);
    if (a.compareTo(b) == 0 || a.compareTo(b) < 0 && p_dec) break;
    p_dec = a.compareTo(b) > 0;
    a = b;
}

```

Например, этот вариант кода выполняется для числа 10^{1000} за 60 миллисекунд, а если убрать улучшенный выбор начального приближения (просто начинать с 1), то будет выполняться примерно 120 миллисекунд.

Тернарный поиск

Постановка задачи

Пусть дана функция $f(x)$, **унимодальная** на некотором отрезке $[l; r]$. Под унимодальностью понимается один из двух вариантов. Первый: функция сначала строго возрастает, потом достигает максимума (в одной точке или целом отрезке), потом строго убывает. Второй вариант, симметричный: функция сначала убывает, достигает минимума, возрастает. В дальнейшем мы будем рассматривать первый вариант, второй будет абсолютно симметричен ему.

Требуется найти **максимум** функции $f(x)$ на отрезке $[l; r]$.

Алгоритм

Возьмём любые две точки m_1 и m_2 в этом отрезке: $l < m_1 < m_2 < r$. Посчитаем значения функции $f(m_1)$ и $f(m_2)$. Дальше у нас получается три варианта:

- Если окажется, что $f(m_1) < f(m_2)$, то искомый максимум не может находиться в левой части, т.е. в части $[l; m_1]$. В этом легко убедиться: если в левой точке функция меньше, чем в правой, то либо эти две точки находятся в области "подъёма" функции, либо только левая точка находится там. В любом случае, это означает, что максимум дальше имеет смысл искать только в отрезке $[m_1; r]$.
- Если, наоборот, $f(m_1) > f(m_2)$, то ситуация аналогична предыдущей с точностью до симметрии. Теперь искомый максимум не может находиться в правой части, т.е. в части $[m_2; r]$, поэтому переходим к отрезку $[l; m_2]$.
- Если $f(m_1) = f(m_2)$, то либо обе эти точки находятся в области максимума, либо левая точка находится в области возрастания, а правая — в области убывания (здесь существенно используется то, что возрастание/убывание строгие). Таким образом, в дальнейшем поиск имеет смысл производить в отрезке $[m_1; m_2]$, но (в целях упрощения кода) этот случай можно отнести к любому из двух предыдущих.

Таким образом, по результату сравнения значений функции в двух внутренних точках мы вместо текущего отрезка поиска $[l; r]$ находим новый отрезок $[l'; r']$. Повторим теперь все действия для этого нового отрезка, снова получим новый, строго меньший, отрезок, и т.д.

Рано или поздно длина отрезка станет маленькой, меньшей заранее определённой константы-точности, и процесс можно останавливать. Этот метод численный, поэтому после остановки алгоритма можно приближённо считать, что во всех точках отрезка $[l; r]$ достигается максимум; в качестве ответа можно взять, например, точку l .

Осталось заметить, что мы не накладывали никаких ограничений на выбор точек m_1 и m_2 . От этого способа, понятно, будет зависеть скорость сходимости (но и возникающая погрешность). Наиболее распространённый способ — выбирать точки так, чтобы отрезок $[l; r]$ делился ими на 3 равные части:

$$m_1 = l + \frac{r - l}{3}$$

$$m_2 = r - \frac{r - l}{3}$$

Впрочем, при другом выборе, когда m_1 и m_2 ближе друг к другу, скорость сходимости несколько увеличится.

Случай целочисленного аргумента

Если аргумент функции f целочисленный, то отрезок $[l; r]$ тоже становится дискретным, однако, поскольку мы не накладывали никаких ограничений на выбор точек m_1 и m_2 , то на корректность алгоритма это никак не влияет. Можно по-прежнему выбирать m_1 и m_2 так, чтобы они делили отрезок $[l; r]$ на 3 части, но уже равные только приблизительно.

Второй отличающийся момент — критерий остановки алгоритма. В данном случае тернарный поиск надо будет останавливать, когда станет $r - l < 3$, ведь в таком случае уже невозможно будет выбрать точки m_1 и m_2 так, чтобы были различными и отличались от l и r , и это может привести к зацикливанию. После того, как алгоритм тернарного поиска остановится и станет $r - l < 3$, из оставшихся нескольких точек-кандидатов $(l, l+1, \dots, r)$ надо выбрать точку с максимальным значением функции.

Реализация

Реализация для непрерывного случая (т.е. функция f имеет вид: double f (double x)):

```
double l = ..., r = ..., EPS = ...; // входные данные
while (r - l > EPS) {
    double m1 = l + (r - l) / 3,
           m2 = r - (r - l) / 3;
    if (f (m1) < f (m2))
        l = m1;
    else
        r = m2;
}
```

Здесь EPS — фактически, **абсолютная погрешность** ответа (не считая погрешностей, связанных с неточным вычислением функции).

Вместо критерия "while ($r - l > \text{EPS}$)" можно выбрать и такой критерий останова:

```
for (int it=0; it<iterations; ++it)
```

С одной стороны, придётся подобрать константу iterations , чтобы обеспечить требуемую точность (обычно достаточно нескольких сотен, чтобы достичь максимальной точности). Но зато, с другой стороны, число итераций перестаёт зависеть от абсолютных величин l и r , т.е. мы фактически с помощью iterations задаём требуемую **относительную погрешность**.

Биномиальные коэффициенты

Биномиальным коэффициентом C_n^k называется количество способов выбрать набор k предметов из n различных предметов без учёта порядка расположения этих элементов (т.е. количество неупорядоченных наборов).

Также биномиальные коэффициенты — это коэффициенты в разложении $(a + b)^n$ (т.н. бином Ньютона):

$$(a + b)^n = C_n^0 a^n + C_n^1 a^{n-1} b + C_n^2 a^{n-2} b^2 + \dots + C_n^k a^{n-k} b^k + \dots + C_n^n b^n$$

Считается, что эту формулу, как и треугольник, позволяющий эффективно находить коэффициенты, открыл Блез Паскаль (Blaise Pascal), живший в 17 в. Тем не менее, она была известна ещё китайскому математику Яну Хуэю (Yang Hui), жившему в 13 в. Возможно, её открыл персидский учёный Омар Хайям (Omar Khayyam). Более того, индийский математик Пингала (Pingala), живший ещё в 3 в. до н.э., получил близкие результаты. Заслуга же Ньютона заключается в том, что он обобщил эту формулу для степеней, не являющихся натуральными.

Вычисление

Аналитическая формула для вычисления:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Эту формулу легко вывести из задачи о неупорядоченной выборке (количество способов неупорядоченно выбрать k элементов из n элементов). Сначала посчитаем количество упорядоченных выборок. Выбрать первый элемент есть n способов, второй — $n - 1$, третий — $n - 2$, и так далее. В результате для числа упорядоченных выборок получаем формулу: $n(n-1)(n-2)\dots(n-k+1) = \frac{n!}{(n-k)!}$. К неупорядоченным выборкам легко перейти, если заметить, что каждой неупорядоченной выборке соответствует ровно $k!$ упорядоченных (т.к. это количество всевозможных перестановок k элементов). В результате, деля $\frac{n!}{(n-k)!}$ на $k!$, мы и получаем исходную формулу.

Рекуррентная формула (с которой связан знаменитый "треугольник Паскаля"):

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

Её легко вывести через предыдущую формулу.

Странно заметить особо, при $n < k$ значение C_n^k всегда полагается равным нулю.

Свойства

Биномиальные коэффициенты обладают множеством различных свойств, приведём наиболее простые из них:

- Правило симметрии:

$$C_n^k = C_n^{n-k}$$

- Внесение-вынесение:

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

- Суммирование по k :

$$\sum_{k=0}^n C_n^k = 2^n$$

- Суммирование по n :

$$\sum_{m=0}^n C_m^k = C_{n+1}^{k+1}$$

- Суммирование по n и k :

$$\sum_{k=0}^m C_{n+k}^k = C_{n+m+1}^m$$

- Суммирование квадратов:

$$(C_n^0)^2 + (C_n^1)^2 + \dots + (C_n^m)^2 = C_{2n}^m$$

- Взвешенное суммирование:

$$1C_n^1 + 2C_n^2 + \dots + nC_n^m = n2^{n-1}$$

- Связь с числами Фибоначчи:

$$C_n^0 + C_{n-1}^1 + \dots + C_{n-k}^k + \dots + C_0^m = F_{n+1}$$

Вычисления в программе

Вычисления по первой, непосредственной формуле, очень легко программировать, однако этот способ подвержен переполнениям даже при сравнительно небольших значениях n и k (даже если ответ вполне помещается в какой-нибудь тип данных, вычисление промежуточных факториалов может привести к переполнению). Поэтому очень часто этот способ можно применять только вместе с [Длинной арифметикой](#):

```
int C (int n, int k) {
    int res = 1;
    for (int i=n-k+1; i<=n; ++i)
        res *= i;
    for (int i=2; i<=k; ++i)
        res /= i;
}
```

Можно заметить, что в приведённой выше реализации в числителе и знаменателе стоит одинаковое количество сомножителей (k), каждый из которых не меньше единицы. Поэтому можно заменить нашу дробь на произведение k дробей, каждая из которых является вещественноненулевой. Однако, можно заметить, что после домножения текущего ответа на каждую очередную дробь всё равно будет получаться целое число (это, например, следует из свойства "внесения-вынесения"). Таким образом, получаем такую реализацию:

```
int C (int n, int k) {
    int res = 1;
    for (int i=1; i<=k; ++i)
        res = res * (n-k+i) / i;
    return res;
}
```

С использованием же рекуррентного соотношения можно построить таблицу биномиальных коэффициентов (фактически, треугольник Паскаля), из неё брать результат. Преимущество этого метода в том, что промежуточные результаты никогда не превосходят ответа, и для вычисления каждого нового элемента таблицы надо всего лишь одно сложение. Недостатком является медленная работа для больших N и K , если на самом деле таблица не нужна, а нужно единственное значение (потому что для вычисления C_n^k понадобится строить таблицу для всех C_i^j , $1 \leq i \leq n$, $1 \leq j \leq n$, или хотя бы до $1 \leq j \leq \min(i, 2k)$).

```
const int maxn = ...;
int C[maxn+1][maxn+1];
for (int n=0; n<=maxn; ++n) {
    C[n][0] = C[n][n] = 1;
    for (int k=1; k<n; ++k)
        C[n][k] = C[n-1][k-1] + C[n-1][k];
```

Если вся таблица значений не нужна, то, как нетрудно заметить, достаточно хранить от неё только две строки (текущую — n -ую строку и предыдущую — $n-1$ -ую).

Числа Каталана

Числа Каталана — числовая последовательность, встречающаяся в удивительном числе комбинаторных задач.

Эта последовательность названа в честь бельгийского математика Каталана (Catalan), жившего в 19 веке, хотя на самом деле она была известна ещё Эйлеру (Euler), жившему за век до Каталана.

Последовательность

Первые несколько чисел Каталана C_n (начиная с нулевого):

1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Числа Каталана встречаются в большом количестве задач комбинаторики. **n -ое число Каталана** — это:

- Количество корректных скобочных последовательностей, состоящих из n открывающих и n закрывающих скобок.
- Количество корневых бинарных деревьев с $n + 1$ листьями (вершины не пронумерованы).
- Количество способов полностью разделить скобками $n + 1$ множитель.
- Количество триангуляций выпуклого $n + 2$ -угольника (т.е. количество разбиений многоугольника непересекающимися диагоналями на треугольники).
- Количество способов соединить $2n$ точек на окружности n непересекающимися хордами.
- Количество неизоморфных полных бинарных деревьев с n внутренними вершинами (т.е. имеющими хотя бы одного сына).
- Количество монотонных путей из точки $(0, 0)$ в точку (n, n) в квадратной решётке размером $n \times n$, не поднимающихся над главной диагональю.
- Количество перестановок длины n , которые можно отсортировать стеком (можно показать, что перестановка является сортируемой стеком тогда и только тогда, когда нет таких индексов $i < j < k$, что $a_k < a_i < a_j$).
- Количество непрерывных разбиений множества из n элементов (т.е. разбиений на непрерывные блоки).
- Количество способов покрыть лесенку $1 \dots n$ с помощью n прямоугольников (имеется в виду фигура, состоящая из n столбцов, i -ый из которых имеет высоту i).

Вычисление

Имеются две формулы для чисел Каталана: рекуррентная и аналитическая. Поскольку мы считаем, что все приведённые выше задачи эквивалентны, то для доказательства формул мы будем выбирать ту задачу, с помощью которой это сделать проще всего.

Рекуррентная формула

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

Рекуррентную формулу легко вывести из задачи о правильных скобочных последовательностях.

Самой левой открывающей скобке l соответствует определённая закрывающая скобка r , которая разбивает формулу две части, каждая из которых в свою очередь является правильной скобочной последовательностью. Поэтому, если мы обозначим $k = r - l - 1$, то для любого фиксированного r будет ровно $C_k C_{n-1-k}$ способов. Суммируя это по всем допустимым k , мы и получаем рекуррентную зависимость на C_n .

Аналитическая формула

$$C_n = \frac{1}{n+1} C_{2n}^n$$

(здесь через C_n^k обозначен, как обычно, [биномиальный коэффициент](#)).

Эту формулу проще всего вывести из задачи о монотонных путях. Общее количество монотонных путей в решётке размером $n \times n$ равно C_{2n}^n . Теперь посчитаем количество монотонных путей, пересекающих диагональ. Рассмотрим какой-либо из таких путей, и найдём первое ребро, которое стоит выше диагонали. Отразим относительно диагонали весь путь, идущий после этого ребра. В результате получим монотонный путь в решётке $(n-1) \times (n+1)$. Но, с другой стороны, любой монотонный путь в решётке $(n-1) \times (n+1)$ обязательно пересекает диагональ, следовательно, он получен как раз таким способом из какого-либо (причём единственного) монотонного пути, пересекающего диагональ, в решётке $n \times n$. Монотонных путей в решётке $(n-1) \times (n+1)$ имеется C_{2n}^{n-1} . В результате получаем формулу:

$$C_n = C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$$

Ожерелья

Задача "ожерелья" — это одна из классических комбинаторных задач. Требуется посчитать количество различных ожерелей из n бусинок, каждая из которых может быть покрашена в один из k цветов. При сравнении двух ожерелей их можно поворачивать, но не переворачивать (т.е. разрешается сделать циклический сдвиг).

Решение

Решить эту задачу можно, используя [лемму Бернсайда и теорему Пойа](#). [Ниже идёт копия текста из этой статьи]

В этой задаче мы можем сразу найти группу инвариантных перестановок. Очевидно, она будет состоять из n перестановок:

$$\begin{aligned}\pi_0 &= 1 \ 2 \ 3 \ \dots \ n \\ \pi_1 &= 2 \ 3 \ \dots \ n \ 1 \\ \pi_2 &= 3 \ \dots \ n \ 1 \ 2 \\ &\dots \\ \pi_{n-1} &= n \ 1 \ 2 \ \dots \ (n-1)\end{aligned}$$

Найдём явную формулу для вычисления $C(\pi_i)$. Во-первых, заметим, что перестановки имеют такой вид, что в i -ой перестановке на j -ой позиции стоит $i + j$ (взятое по модулю n , если оно больше n). Если мы будем рассматривать циклическую структуру i -й перестановки, то увидим, что единица переходит в $1+i$, $1+i$ переходит в $1+2i$, $1+2i$ — в $1+3i$, и т.д., пока не придём в число $1+kn$; для остальных элементов выполняются похожие утверждения. Отсюда можно понять, что все циклы имеют одинаковую длину, равную $\text{lcm}(i, n)/i$, т.е. $n/\gcd(i, n)$ ("gcd" — наибольший общий делитель, "lcm" — наименьшее общее кратное). Тогда количество циклов в i -й перестановке будет равно просто $\gcd(i, n)$.

Подставляя найденные значения в теорему Пойа, получаем **решение**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\gcd(i, n)}$$

Можно оставить формулу в таком виде, а можно её свернуть ещё больше. Переидём от суммы по всем i к сумме только по делителям n . Действительно, в нашей сумме будет много одинаковых слагаемых: если i не является делителем n , то таковой делитель найдётся после вычисления $\gcd(i, n)$. Следовательно, для каждого делителя $d|n$ его слагаемое $k^{\gcd(d, n)} = k^d$ учтётся несколько раз, т.е. сумму можно представить в таком виде:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} C_d k^d$$

где C_d — это количество таких чисел i , что $\gcd(i, n) = d$. Найдём явное выражение для этого количества. Любое такое число i имеет вид: $i = dj$, где $\gcd(j, n/d) = 1$ (иначе было бы $\gcd(i, n) > d$). Вспоминая [функцию Эйлера](#), мы находим, что количество таких j — это величина функции Эйлера $\phi(n/d)$. Таким образом, $C_d = \phi(n/d)$, и окончательно получаем **формулу**:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

Расстановка слонов на шахматной доске

Требуется найти количество способов расставить K слонов на доске размером $N \times N$.

Алгоритм

Решать задачу будем с помощью [динамического программирования](#).

Пусть $D[i][j]$ — количество способов расставить j слонов на диагоналях до i -ой включительно, причём только тех диагоналях, которые того же цвета, что и i -ая диагональ. Тогда $i = 1..2N-1$, $j = 0..K$.

Диагонали занумеруем следующим образом (пример для доски 5×5):

чёрные : белые :

1	5	9	2	6	8
5	9	7	6	8	4
9	7	3	8	4	
—	—	—	—	—	—

Т.е. нечётные номера соответствуют чёрным диагоналям, чётные - белым; диагонали нумеруем в порядке увеличения количества элементов в них.

При такой нумерации мы можем вычислить каждое $D[i][j]$, основываясь только на $D[i-2][j]$ (двойка вычитается, чтобы мы рассматривали диагональ того же цвета).

Итак, пусть текущий элемент динамики - $D[i][j]$. Имеем два перехода. Первый - $D[i-2][j]$, т.е. ставим всех j слонов на предыдущие диагонали. Второй переход - если мы ставим одного слона на текущую диагональ, а остальных $j-1$ слонов - на предыдущие; заметим, что количество способов поставить слона на текущую диагональ равно количеству клеток в ней минус $j-1$, т.к. слоны, стоящие на предыдущих диагоналях, будут перекрывать часть направлений. Таким образом, имеем:

$$D[i][j] = D[i-2][j] + D[i-2][j-1] \cdot (\text{cells}(i) - j + 1)$$

где $\text{cells}(i)$ - количество клеток, лежащих на i -ой диагонали. Например, cells можно вычислять так:

```
int cells (int i) {
    if (i & 1)
        return i / 4 * 2 + 1;
    else
        return (i - 1) / 4 * 2 + 2;
}
```

Осталось определить базу динамики, тут никаких сложностей нет: $D[0][0] = 1$, $D[1][1] = 1$.

Наконец, вычислив динамику, найти собственно **ответ** к задаче несложно. Перебираем количество $i=0..K$ слонов, стоящих на чёрных диагоналях (номер последней чёрной диагонали - $2N-1$), соответственно $K-i$ слонов ставим на белые диагонали (номер последней белой диагонали - $2N-2$), т.е. к ответу прибавляем величину $D[2N-1][i] * D[2N-2][K-i]$.

Реализация

```
int n, k; // входные данные
if (k > 2*n-1) {
    cout << 0;
    return 0;
}

vector < vector<int> > d (n*2, vector<int> (k+2));
for (int i=0; i<n*2; ++i)
    d[i][0] = 1;
d[1][1] = 1;
for (int i=2; i<n*2; ++i)
    for (int j=1; j<=k; ++j)
        d[i][j] = d[i-2][j] + d[i-2][j-1] * (cells(i) - j + 1);

int ans = 0;
for (int i=0; i<=k; ++i)
    ans += d[n*2-1][i] * d[n*2-2][k-i];
cout << ans;
```

Правильные скобочные последовательности

Правильной скобочной последовательностью называется строка, состоящая только из символов "скобки" (чаще всего только круглые скобки, но здесь будет рассматриваться и общий случай), где каждой закрывающей скобке найдётся соответствующая открывающая (причём того же типа).

Здесь мы рассмотрим классические задачи на правильные скобочные последовательности (далее для краткости просто "последовательности"): проверка на правильность, количество последовательностей, генерация всех последовательностей, нахождение лексикографически следующей последовательности, нахождение K -ой последовательности в отсортированном списке всех последовательностей, и, наоборот, определение номера последовательности. Каждая из задач рассмотрена в двух случаях — когда разрешены скобки только одного типа, и когда нескольких типов.

Проверка на правильность

Пусть сначала разрешены скобки только одного типа, тогда проверить последовательность на правильность можно очень простым алгоритмом. Пусть depth — это текущее количество открытых скобок. Изначально $\text{depth} = 0$. Будем двигаться по строке слева направо,

если текущая скобка открывающая, то увеличим depth на единицу, иначе уменьшим. Если при этом когда-то получалось отрицательное число, или в конце работы алгоритма depth отлично от нуля, то данная строка не является правильной скобочной последовательностью, иначе является.

Если допустимы скобки нескольких типов, то алгоритм нужно изменить. Вместо счётчика depth следует создать стек, в который будем класть открывающие скобки по мере поступления. Если текущий символ строки — открывающая скобка, то кладём его в стек, а если закрывающая — то проверяем, что стек не пуст, и что на его вершине лежит скобка того же типа, что и текущая, и затем достаём эту скобку из стека. Если какое-либо из условий не выполнилось, или в конце работы алгоритма стек остался не пуст, то последовательность не является правильной скобочной, иначе является.

Таким образом, обе эти задачи мы научились решать за время $O(n)$.

Количество последовательностей

Количество правильных скобочных последовательностей с одним типом скобок можно вычислить как [число Каталана](#). Т.е. если есть n пар скобок (строка длины $2n$), то количество будет равно:

$$\frac{1}{n+1} C_{2n}^n$$

Пусть теперь имеется не один, а k типов скобок. Тогда каждая пара скобок независимо от остальных может принимать один из k типов, а потому мы получаем такую формулу:

$$\frac{1}{n+1} C_{2n}^n k^n$$

Нахождение всех последовательностей

Иногда требуется найти и вывести все правильные скобочные последовательности указанной длины n (в данном случае n — это длина строки).

Для этого можно начать с лексикографически первой последовательности "...(((...))...", а затем находить каждый раз лексикографически следующую последовательность с помощью алгоритма, описанного в следующем разделе.

Но если ограничения не очень большие (n до $10 - 12$), то можно поступить значительно проще. Найдём всевозможные перестановки этих скобок (для этого удобно использовать функцию `next_permutation()`), их будет C_{2n}^n , и каждую проверим на правильность вышеописанным алгоритмом, и в случае правильности выведем текущую последовательность.

Также процесс нахождения всех последовательностей можно оформить в виде рекурсивного перебора с отсечениями.

Нахождение следующей последовательности

Здесь рассматривается только случай одного типа скобок.

По заданной правильной скобочной последовательности требуется найти правильную скобочную последовательность, которая находится следующей в лексикографическом порядке после текущей (или выдать "No solution", если такой не существует).

Будем двигаться по последовательности справа налево и считать количество открывающих и закрывающих скобок. Если в какой-то момент мы стоим на открывающей скобке, а количество открывающих скобок строго меньше количества закрывающих, то мы нашли самую правую позицию, от которой мы можем начать изменять последовательность. Поставим в неё закрывающую скобку, затем максимально возможное количество открывающих скобок, а затем все оставшиеся закрывающие скобки. Если такой позиции мы не смогли найти, то ответа не существует.

Реализация:

```
string s;
cin >> s;
int n = (int) s.length();
string ans = "No solution";
for (int i=n-1, c1=0, c2=0; i>=0; --i) {
    if (s[i] == '(')
        ++c1;
    else
        ++c2;
    if (s[i] == '(' && c1 < c2) {
        ans = s.substr(0, i) + ')';
        for (int k=0; k<c1; ++k)
            ans += '(';
        for (int k=0; k<c2-1; ++k)
            ans += ')';
        break;
    }
}
cout << ans;
```

Таким образом, мы решили эту задачу за $O(n)$.

Номер последовательности

Здесь пусть N — количество пар скобок в последовательности. Требуется по заданной правильной скобочной последовательности найти её номер в списке лексикографически упорядоченных правильных скобочных последовательностей.

Научимся считать [динамику](#) $D[i][j]$, где i — длина последовательности (не обязательно правильной), j — баланс (т.е. разность между количеством открывающих и закрывающих скобок, рассматриваются только случаи, когда баланс неотрицателен), $D[i][j]$ — количество таких последовательностей.

Базой динамики является $D[0][0] = 1$, а переходы такие: пусть мы находимся в состоянии (i, j) , и попробуем добавить '(' или ')'. Если мы добавляем открывающую скобку, то переходим в состояние $(i + 1, j + 1)$, иначе — в $(i + 1, j - 1)$.

Таким образом, эту динамику мы можем посчитать за $O(n^2)$.

Сначала пусть допустимы только скобки **одного** типа.

Теперь посчитаем с помощью этой динамики номер последовательности. Для этого заведём счётчик `depth` глубины вложенности в скобки, и будем двигаться по последовательности слева направо. Если текущий символ $S[i]$ ($i = 0 \dots 2n - 1$) равен '(', то мы увеличиваем `depth` на 1 и переходим к следующему символу. Если же текущий символ равен ')', то мы должны прибавить к ответу $D[2n - i - 1][\text{depth} + 1]$, тем самым учитывая, что в этой позиции мог бы стоять символ '(' (который бы привёл к лексикографически меньшей последовательности, чем текущая); затем мы уменьшаем `depth` на единицу.

Пусть теперь разрешены скобки **нескольких** k типов. Тогда, ясно, мы должны на каждом шаге перебирать все скобки, которые меньше текущего символа, и прибавлять к ответу соответствующее значение $D[2n - i - 1][\text{ndepth}]$, однако его надо будет домножить на $k^{(2n-i-1)/2}$, чтобы учесть, что в этом остатке могли быть скобки различных типов, а парных скобок в этом остатке будет только $2n - i - 1 - \text{ndepth}$, поскольку `ndepth` скобок являются закрывающими для открывающих скобок, находящихся вне этого остатка (а потому их типы мы варьировать не можем).

Нахождение k -ой последовательности

Здесь пусть n — количество пар скобок в последовательности. В данной задаче по заданному k требуется найти k -ую правильную скобочную последовательность в списке лексикографически упорядоченных последовательностей.

Как и в предыдущем разделе, посчитаем динамику $D[i][j]$ — количество правильных скобочных последовательностей длины i с балансом j .

Пусть сначала допустимы только скобки **одного** типа.

Будем двигаться по символам искомой строки, с 0-го по $2n - 1$ -ый. Как и в предыдущей задаче, будем хранить счётчик `depth` — текущую глубину вложенности в скобки. В каждой текущей позиции будем перебирать возможный символ — открывающую скобку или закрывающую. Пусть мы хотим поставить сюда открывающую скобку, тогда мы должны посмотреть на значение $D[i+1][\text{depth} + 1]$. Если оно $\geq k$, то мы ставим в текущую позицию открывающую скобку, увеличиваем `depth` на единицу и переходим к следующему символу. Иначе мы отнимаем от k величину $D[i+1][\text{depth} + 1]$, ставим закрывающую скобку и уменьшаем значение `depth`. В конце концов мы и получим искомую скобочную последовательность.

Реализация на языке Java с использованием длинной арифметики:

```
int n; BigInteger k; // входные данные

BigInteger d[][][] = new BigInteger [n*2+1][n+1];
for (int i=0; i<=n*2; ++i)
    for (int j=0; j<=n; ++j)
        d[i][j] = BigInteger.ZERO;
d[0][0] = BigInteger.ONE;
for (int i=0; i<n*2; ++i)
    for (int j=0; j<=n; ++j) {
        if (j+1 <= n)
            d[i+1][j+1] = d[i+1][j+1].add( d[i][j] );
        if (j > 0)
            d[i+1][j-1] = d[i+1][j-1].add( d[i][j] );
    }

String ans = new String();
if (k.compareTo( d[n*2][0] ) > 0)
    ans = "No solution";
else {
    int depth = 0;
    for (int i=n*2-1; i>=0; --i)
        if (depth+1 <= n && d[i][depth+1].compareTo( k ) >= 0) {
            ans += '(';
            ++depth;
        }
        else {
            ans += ')';
            if (depth+1 <= n)
                k = k.subtract( d[i][depth+1] );
            --depth;
        }
    }
}
```

Пусть теперь разрешён не один, а k **типов** скобок. Тогда алгоритм решения будет отличаться от предыдущего случая только тем, что мы должны домножать значение $D[i+1][\text{ndepth}]$ на величину $k^{(2n-i-1-\text{ndepth})/2}$, чтобы учесть, что в этом остатке могли быть скобки различных типов, а парных скобок в этом остатке будет только $2n - i - 1 - \text{ndepth}$, поскольку `ndepth` скобок являются закрывающими для открывающих скобок, находящихся вне этого остатка (а потому их типы мы варьировать не можем).

Реализация на языке Java для случая двух типов скобок — круглых и квадратных:

```
int n; BigInteger k; // входные данные

BigInteger d[][][] = new BigInteger [n*2+1][n+1];
for (int i=0; i<=n*2; ++i)
    for (int j=0; j<=n; ++j)
        d[i][j] = BigInteger.ZERO;
d[0][0] = BigInteger.ONE;
```

```

for (int i=0; i<n*2; ++i)
    for (int j=0; j<=n; ++j) {
        if (j+1 <= n)
            d[i+1][j+1] = d[i+1][j+1].add( d[i][j] );
        if (j > 0)
            d[i+1][j-1] = d[i+1][j-1].add( d[i][j] );
    }

String ans = new String();
int depth = 0;
char [] stack = new char[n*2];
int stacksz = 0;
for (int i=n*2-1; i>=0; --i) {
    BigInteger cur;
    // '('
    if (depth+1 <= n)
        cur = d[i][depth+1].shiftLeft( (i-depth-1)/2 );
    else
        cur = BigInteger.ZERO;
    if (cur.compareTo( k ) >= 0) {
        ans += '(';
        stack[stacksz++] = '(';
        ++depth;
        continue;
    }
    k = k.subtract( cur );
    // ')'
    if (stacksz > 0 && stack[stacksz-1] == '(' && depth-1 >= 0)
        cur = d[i][depth-1].shiftLeft( (i-depth+1)/2 );
    else
        cur = BigInteger.ZERO;
    if (cur.compareTo( k ) >= 0) {
        ans += ')';
        --stacksz;
        --depth;
        continue;
    }
    k = k.subtract( cur );
    // '['
    if (depth+1 <= n)
        cur = d[i][depth+1].shiftLeft( (i-depth-1)/2 );
    else
        cur = BigInteger.ZERO;
    if (cur.compareTo( k ) >= 0) {
        ans += '[';
        stack[stacksz++] = '[';
        ++depth;
        continue;
    }
    k = k.subtract( cur );
    // ']'
    ans += ']';
    --stacksz;
    --depth;
}

```

Количество помеченных графов

Дано число N вершин. Требуется посчитать количество G_N различных помеченных графов с N вершинами (т.е. вершины графа помечены различными числами от 1 до N , и графы сравниваются с учётом этой покраски вершин). Рёбра графа неориентированы, петли и кратные рёбра запрещены.

Рассмотрим множество всех возможных рёбер графа. Для любого ребра (i, j) положим, что $i < j$ (основываясь на неориентированности графа и отсутствии петель). Тогда множество всех возможных рёбер графа имеет мощность C_N^2 , т.е. $\frac{N(N - 1)}{2}$.

Поскольку любой помеченный граф однозначно определяется своими рёбрами, то количество помеченных графов с N вершинами равно:

$$G_N = 2^{\frac{N(N-1)}{2}}$$

Количество связных помеченных графов

По сравнению с предыдущей задачей мы дополнительно накладываем ограничение, что граф должен быть связным.

Обозначим искомое число через Conn_N .

Научимся, наоборот, считать количество **несвязных** графов; тогда количество связных графов получится как C_N минус найденное число. Более того, научимся считать количество **корневых** (т.е. с выделенной вершиной - корнем) **несвязных графов**; тогда количество несвязных графов будет получаться из него делением на N . Заметим, что, так как граф несвязный, то в нём найдётся компонента связности, внутри которой лежит корень, а остальной граф будет представлять собой ещё несколько (как минимум одну) компонент связности.

Переберём количество K вершин в этой компоненте связности, содержащей корень (очевидно, $K = 1 \dots N - 1$), и найдём количество таких графов. Во-первых, мы должны выбрать K вершин из N , т.е. ответ умножается на C_N^K . Во-вторых, компонента связности с корнем даёт множитель Conn_K . В-третьих, оставшийся граф из $N - K$ вершин является произвольным графом, а потому он даёт множитель G_{N-K} . Наконец, количество способов выделить корень в компоненте связности из K вершин равно K . Итого, при фиксированном K количество **корневых несвязных** графов равно:

$$K C_N^K \text{Conn}_K G_{N-K}$$

Значит, количество **несвязных** графов с N вершинами равно:

$$\frac{1}{N} \sum_{K=1}^{N-1} K C_N^K \text{Conn}_K G_{N-K}$$

Наконец, искомое количество **связных** графов равно:

$$\text{Conn}_N = G_N - \frac{1}{N} \sum_{K=1}^{N-1} K C_N^K \text{Conn}_K G_{N-K}$$

Количество помеченных графов с K компонентами связности

Основываясь на предыдущей формуле, научимся считать количество помеченных графов с N вершинами и K компонентами связности.

Сделать это можно с помощью динамического программирования. Научимся считать $D[N][K]$ — количество помеченных графов с N вершинами и K компонентами связности.

Научимся вычислять очередной элемент $D[N][K]$, зная предыдущие значения. Воспользуемся стандартным приёмом при решении таких задач: возьмём вершину с номером 1, она принадлежит какой-то компоненте, вот эту компоненту мы и будем перебирать. Переберём размер S этой компоненты, тогда количество способов выбрать такое множество вершин равно C_{N-1}^{S-1} (одну вершину — вершину 1 — перебирать не надо). Количество же способов построить компоненту связности из S вершин мы уже умеем считать — это Conn_S . После удаления этой компоненты из графа у нас остаётся граф с $N - S$ вершинами и $K - 1$ компонентами связности, т.е. мы получили рекуррентную зависимость, по которой можно вычислять значения $D[]$:

$$D[N][K] = \sum_{S=1}^N C_{N-1}^{S-1} \text{Conn}_S D[N-S][K-1]$$

Итого получаем примерно такой код:

```
int d[n+1][k+1]; // изначально заполнен нулями
d[0][0][0] = 1;
for (int i=1; i<=n; ++i)
    for (int j=1; j<=i && j<=k; ++j)
        for (int s=1; s<=i; ++s)
            d[i][j] += C[i-1][s-1] * conn[s] * d[i-s][j-1];
cout << d[n][k][n];
```

Разумеется, на практике, скорее всего, нужна будет [длинная арифметика](#).

Генерация сочетаний из N элементов

Сочетания из N элементов по K в лексикографическом порядке

Постановка задачи. Даны натуральные числа N и K . Рассмотрим множество чисел от 1 до N . Требуется вывести все различные его подмножества мощности K , причём в лексикографическом порядке.

Алгоритм весьма прост. Первым сочетанием, очевидно, будет сочетание $(1, 2, \dots, K)$. Научимся для текущего сочетания находить лексикографически следующее. Для этого в текущем сочетании найдём самый правый элемент, не достигший ещё своего наибольшего значения; тогда увеличим его на единицу, а всем последующим элементам присвоим наименьшие значения.

```

bool next_combination (vector<int> & a, int n) {
    int k = (int)a.size();
    for (int i=k-1; i>=0; --i)
        if (a[i] < n-k+i+1) {
            ++a[i];
            for (int j=i+1; j<k; ++j)
                a[j] = a[j-1]+1;
            return true;
        }
    return false;
}

```

С точки зрения производительности, этот алгоритм линеен (в среднем), если K не близко к N (т.е. если не выполняется, что $K = N - o(N)$). Для этого достаточно доказать, что сравнения " $a[i] < n-k+i+1$ " выполняются в сумме C_{n+1}^k раз, т.е. в $(N+1) / (N-K+1)$ раз больше, чем всего есть сочетаний из N элементов по K .

Сочетания из N элементов по K с изменениями ровно одного элемента

Требуется выписать все сочетания из N элементов по K , но в таком порядке, что любые два соседних сочетания будут отличаться ровно одним элементом.

Интуитивно можно сразу заметить, что эта задача похожа на задачу генерации всех подмножеств данного множества в таком порядке, когда два соседних подмножества отличаются ровно одним элементом. Эта задача непосредственно решается с помощью [Кода Грэя](#): если мы каждому подмножеству поставим в соответствие битовую маску, то, генерируя с помощью кодов Грэя эти битовые маски, мы и получим ответ.

Может показаться удивительным, но задача генерации сочетаний также непосредственно решается с помощью [кода Грэя](#). А именно, сгенерируем коды Грэя для чисел от 0 до 2^N-1 , и оставим только те коды, которые содержат ровно K единиц. Удивительный факт заключается в том, что в полученной последовательности любые две соседние маски (а также первая и последняя маски) будут отличаться ровно двумя битами, что нам как раз и требуется.

Докажем это.

Для доказательства вспомним факт, что последовательность $G(N)$ кодов Грэя можно получить следующим образом:

$$G(N) = 0G(N-1) \cup 1G(N-1)^R$$

т.е. берём последовательность кодов Грэя для $N-1$, дописываем в начало каждой маски 0, добавляем к ответу; затем снова берём последовательность кодов Грэя для $N-1$, инвертируем её, дописываем в начало каждой маски 1 и добавляем к ответу.

Теперь мы можем произвести доказательство.

Сначала докажем, что первая и последняя маски будут отличаться ровно в двух битах. Для этого достаточно заметить, что первая маска будет иметь вид $N-K$ нулей и K единиц, а последняя маска будет иметь вид: единица, потом $N-K-1$ нулей, потом $K-1$ единица. Доказать это легко по индукции по N , пользуясь приведённой выше формулой для последовательности кодов Грэя.

Теперь докажем, что любые два соседних кода будут отличаться ровно в двух битах. Для этого снова обратимся к формуле для последовательности кодов Грэя. Пусть внутри каждой из половинок (образованных из $G(N-1)$) утверждение верно, докажем, что оно верно для всей последовательности. Для этого достаточно доказать, что оно верно в месте "склеивания" двух половинок $G(N-1)$, а это легко показать, основываясь на том, что мы знаем первый и последний элементы этих половинок.

Приведём теперь наивную реализацию, работающую за 2^N :

```

int gray_code (int n) {
    return n ^ (n >> 1);
}

int count_bits (int n) {
    int res = 0;
    for (; n; n>>=1)
        res += n & 1;
    return res;
}

void all_combinations (int n, int k) {
    for (int i=0; i<(1<<n); ++i) {
        int cur = gray_code (i);
        if (count_bits (cur) == k) {
            for (int j=0; j<n; ++j)
                if (cur & (1<<j))
                    printf ("%d ", j+1);
            puts ("");
        }
    }
}

```

Стоит заметить, что возможна и в некотором смысле более эффективная реализация, которая будет строить всевозможные сочетания на ходу, и тем самым работать за $O(C_n^k n)$. С другой стороны, эта реализация представляет собой рекурсивную функцию, и поэтому для небольших n , вероятно, она имеет большую скрытую константу, чем предыдущее решение.

Собственно сама реализация - это непосредственное следование формуле:

$$G(N, K) = 0G(N-1, K) \cup 1G(N-1, K-1)^R$$

Эта формула легко получается из приведённой выше формулы для последовательности Грэя - мы просто выбираем подпоследовательность из подходящих нам элементов.

```

bool ans[MAXN];

void gen (int n, int k, int l, int r, bool rev) {
    if (k > n || k < 0) return;
    if (!n) {
        for (int i=0; i<n; ++i)
            printf ("%d", (int)ans[i]);
        puts ("");
        return;
    }
    ans[rev?r:l] = false;
    gen (n-1, k, !rev?l+1:l, !rev?r:r-1, rev);
    ans[rev?r:l] = true;
    gen (n-1, k-1, !rev?l+1:l, !rev?r:r-1, !rev);
}

void all_combinations (int n, int k) {
    gen (n, k, 0, n-1, false);
}

```

Лемма Бернсайда. Теорема Пойа

Лемма Бернсайда

Эта лемма была сформулирована и доказана **Бернсайдом** (Burnside) в 1897 г., однако было установлено, что эта формула была ранее открыта **Фробениусом** (Frobenius) в 1887 г., а ещё раньше - **Коши** (Cauchy) в 1845 г. Поэтому эта формула иногда называется леммой Бернсайда, а иногда - теоремой Коши-Фробениуса.

Лемма Бернсайда позволяет посчитать количество классов эквивалентности в некотором множестве, основываясь на некоторой его внутренней симметрии.

Объекты и представления

Проведём чёткую грань между количеством объектов и количеством представлений.

Одним и тем же объектам могут соответствовать различные представления, но, разумеется, любое представление соответствует ровно одному объекту. Следовательно, множество всех представлений разбивается на классы эквивалентности. Наша задача — в подсчёте именно числа объектов, или, что то же самое, количества классов эквивалентности.

Пример задачи: раскраска бинарных деревьев

Допустим, мы рассматриваем следующую задачу. Требуется посчитать количество способов раскрасить корневые бинарные деревья с n вершинами в 2 цвета, если у каждой вершины мы не различаем правого и левого сына.

Множество объектов здесь — это множество различных в этом понимании раскрасок деревьев.

Определим теперь множество представлений. Каждой раскраске поставим в соответствие задающую её функцию $f(v)$, где $v = 1 \dots n$, а $f(v) = 0 \dots 1$. Тогда множество представлений — это множество различных функций такого вида, и размер его, очевидно, равен 2^n . В то же время, на этом множестве представлений мы ввели разбиение на классы эквивалентности.

Например, пусть $n = 3$, а дерево таково: корень — вершина 1, а вершины 2 и 3 — её сыновья. Тогда следующие функции f_1 и f_2 считаются эквивалентными:

$$\begin{aligned} f_1(1) &= 0 & f_2(1) &= 0 \\ f_1(2) &= 1 & f_2(2) &= 0 \\ f_1(3) &= 0 & f_2(3) &= 1 \end{aligned}$$

Инвариантные перестановки

Почему эти две функции f_1 и f_2 принадлежат одному классу эквивалентности? Интуитивно это понятно — потому что мы можем переставить местами сыновей вершины 1, т.е. вершины 2 и 3, а после такого преобразования функции f_1 и f_2 совпадут. Но формально это означает, что найдётся такая **инвариантная перестановка** π (т.е. которая по условию задачи не меняет сам объект, а только его представление), такая, что:

$$f_2\pi \equiv f_1$$

Итак, исходя из условия задачи, мы можем найти все инвариантные перестановки, т.е. применяя которые мы не переходим из одного класса эквивалентности в другой. Тогда, чтобы проверить, являются ли две функции f_1 и f_2 эквивалентными (т.е. соответствуют ли они на самом деле одному объекту), надо для каждой инвариантной перестановки π проверить, не выполнится ли условие: $f_2\pi \equiv f_1$ (или, что то же самое, $f_1\pi \equiv f_2$). Если хотя бы для одной перестановки обнаружилось это равенство, то f_1 и f_2 эквивалентны, иначе они не эквивалентны.

Нахождение всех таких инвариантных перестановок, относительно которых наша задача инвариантна — это ключевой шаг для применения как леммы Бернсаида, так и теоремы Пойа. Понятно, что эти инвариантные перестановки зависят от конкретной задачи, и их нахождение — процесс чисто эвристический, основанный на интуитивных соображениях. Впрочем, в большинстве случаев достаточно вручную найти несколько "основных" перестановок, из которых все остальные перестановки могут быть получены их всевозможными произведениями (и эту, исключительно механическую, часть работы можно переложить на компьютер; более подробно это будет рассмотрено ниже на примере конкретной задачи).

Нетрудно понять, что инвариантные перестановки образуют **группу** — поскольку произведение любых инвариантных перестановок тоже является инвариантной перестановкой. Обозначим **группу инвариантных перестановок** через G .

Формулировка леммы

Для формулировки осталось напомнить одно понятие из алгебры. **Неподвижной точкой** f для перестановки π называется такой элемент, который инвариантен относительно этой перестановки: $f \equiv f\pi$. Например, в нашем примере неподвижными точками будут являться те функции f , которые соответствуют раскраскам, не меняющимся при применении к ним перестановки π (не меняющимся именно в формальном смысле равенства двух функций). Обозначим через $I(\pi)$ **количество неподвижных точек** для перестановки π .

Тогда **лемма Бернсаида** звучит следующим образом: количество классов эквивалентности равно сумме количеств неподвижных точек по всем перестановкам из группы G , делённой на размер этой группы:

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} I(\pi)$$

Хотя лемма Бернсаида сама по себе не так удобна для применения на практике (пока непонятно, как быстро искать величину $I(\pi)$), она наиболее ясно раскрывает математическую суть, на которой основана идея подсчёта классов эквивалентности.

Доказательство леммы Бернсаида

Описанное здесь доказательство леммы Бернсаида не так важно для её понимания и применения на практике, поэтому его можно пропустить при первом чтении.

Приведённое здесь доказательство является самым простым из известных и не использует теорию групп. Это доказательство было опубликовано Богартом (Bogart) и Кеннетом (Kenneth) в 1991 г.

Итак, нам нужно доказать следующее утверждение:

$$\text{ClassesCount}|G| = \sum_{\pi \in G} I(\pi)$$

Величина, стоящая справа — это не что иное, как количество "инвариантных пар" (f, π) , т.е. таких пар, что $f\pi \equiv f$. Очевидно, что в формуле мы имеем право изменить порядок суммирования — сделать внешнюю сумму по элементам f , а внутри неё поставить величину $J(f)$ — количество перестановок, относительно которых f инвариантна:

$$\text{ClassesCount}|G| = \sum_f J(f)$$

Для доказательства этой формулы составим таблицу, столбцы которой будут подписаны всеми значениями f_i , строки — всеми перестановками π_j , а в клетках таблицы будут стоять произведения $f_i\pi_j$. Тогда, если мы будем рассматривать столбцы этой таблицы как множества, то некоторые из них могут совпасть, и это будет как означать, что соответствующие этим столбцам f также эквивалентны. Таким образом, количество различных как множество столбцов равно искомой величине **ClassesCount**. Кстати говоря, с точки зрения теории групп столбец таблицы, подписанный некоторым элементом f_i — это орбита этого элемента; для эквивалентных элементов, очевидно, орбиты совпадают, и число различных орбит даёт именно **ClassesCount**.

Итак, столбцы таблицы сами распадаются на классы эквивалентности; зафиксируем теперь какой-либо класс и рассмотрим столбцы в нём. Во-первых, заметим, что в этих столбцах могут стоять только элементы f_i одного класса эквивалентности (иначе получилось бы, что некоторым эквивалентным преобразованием π_j мы перешли в другой класс эквивалентности, что невозможно). Во-вторых, каждый элемент f_i будет встречаться одинаковое число раз во всех столбцах (это также следует из того, что столбцы соответствуют эквивалентным элементам). Отсюда можно сделать вывод, что все столбцы внутри одного класса эквивалентности совпадают друг с другом как мульти множества.

Теперь зафиксируем произвольный элемент f . С одной стороны, он встречается в своём столбце ровно $J(f)$ раз (по самому определению $J(f)$). С другой стороны, все столбцы внутри одного класса эквивалентности одинаковы как мульти множества. Следовательно, внутри каждого столбца данного класса эквивалентности любой элемент g встречается ровно $J(g)$ раз.

Таким образом, если мы возьмём произвольным образом от каждого класса эквивалентности по одному столбцу и просуммируем количество элементов в них, то получим, с одной стороны, $\text{ClassesCount}|G|$ (это получается, просто умножив количество столбцов на их размер), а с другой стороны — сумму величин $J(f)$ по всем f (это следует из всех предыдущих рассуждений):

$$\text{ClassesCount}|G| = \sum_f J(f)$$

что и требовалось доказать.

Теорема Пойа. Простейший вариант

Теорема **Пойа** (Polya) является обобщением леммы Бернсаида, к тому же предоставляемая более удобный инструмент для нахождения количества классов эквивалентности. Следует отметить, что ещё до Пойа эта теорема была открыта и доказана Редфилдом (Redfield) в 1927 г., однако его публикация прошла незамеченной математиками того времени. Пойа независимо пришёл к тому же результату лишь в 1937 г., и его публикация была более удачной.

Здесь мы рассмотрим формулу, получающуюся как частный случай теоремы Пойа, и которую очень удобно использовать для вычислений на практике. Общая теорема Пойа в данной статье рассматриваться не будет.

Обозначим через $C(\pi)$ количество циклов в перестановке π . Тогда выполняется следующая формула (**частный случай теоремы Пойа**):

$$\text{ClassesCount} = \frac{1}{|G|} \sum_{\pi \in G} k^{C(\pi)}$$

где k — количество значений, которые может принимать каждый элемент представления $f(v)$. Например, в нашей задаче-примере (раскраска корневого бинарного дерева в 2 цвета) $k = 2$.

Доказательство

Эта формула является прямым следствием леммы Бернсайда. Чтобы получить её, нам надо просто найти явное выражение для величины $I(\pi)$, фигурирующую в лемме (напомним, это количество неподвижных точек перестановки π).

Итак, рассмотрим некоторую перестановку π и некоторый элемент f . Под действием перестановки π элементы f передвигаются, как известно, по циклам перестановки. Заметим, что так как в результате должно получаться $f \equiv f\pi$, то внутри каждого цикла перестановки должны находиться одинаковые элементы f . В то же время, для разных циклов никакой связи между значениями элементов не возникает. Таким образом, для каждого цикла перестановки π мы выбираем по одному значению (среди k вариантов), и тем самым мы получим все представления f , инвариантные относительно этой перестановки, т.е.:

$$I(\pi) = k^{C(\pi)}$$

где $C(\pi)$ — количество циклов перестановки.

Пример задачи: Ожерелья

Задача "ожерелья" — это одна из классических комбинаторных задач. Требуется посчитать количество различных ожерелий из n бусинок, каждая из которых может быть покрашена в один из k цветов. При сравнении двух ожерелий их можно поворачивать, но не переворачивать (т.е. разрешается сделать циклический сдвиг).

В этой задаче мы можем сразу найти группу инвариантных перестановок. Очевидно, она будет состоять из n перестановок:

$$\begin{aligned}\pi_0 &= 1 \ 2 \ 3 \ \dots \ n \\ \pi_1 &= 2 \ 3 \ \dots \ n \ 1 \\ \pi_2 &= 3 \ \dots \ n \ 1 \ 2 \\ &\dots \\ \pi_{n-1} &= n \ 1 \ 2 \ \dots \ (n-1)\end{aligned}$$

Найдём явную формулу для вычисления $C(\pi_i)$. Во-первых, заметим, что перестановки имеют такой вид, что в i -ой перестановке на j -ой позиции стоит $i + j$ (взятое по модулю n , если оно больше n). Если мы будем рассматривать циклическую структуру i -ой перестановки, то увидим, что единица переходит в $1+i$, $1+i$ переходит в $1+2i$, $1+2i$ — в $1+3i$, и т.д., пока не придёт в число $1+kn$; для остальных элементов выполняются похожие утверждения. Отсюда можно понять, что все циклы имеют одинаковую длину, равную $\text{lcm}(i, n)/i$, т.е. $n/\gcd(i, n)$ ("gcd" — наибольший общий делитель, "lcm" — наименьшее общее кратное). Тогда количество циклов в i -ой перестановке будет равно просто $\gcd(i, n)$.

Подставляя найденные значения в теорему Пойа, получаем **решение**:

$$\text{Ans} = \frac{1}{n} \sum_{i=1}^n k^{\gcd(i, n)}$$

Можно оставить формулу в таком виде, а можно её свернуть ещё больше. Перейдём от суммы по всем i к сумме только по делителям n . Действительно, в нашей сумме будет много одинаковых слагаемых: если i не является делителем n , то таковой делитель найдётся после вычисления $\gcd(i, n)$. Следовательно, для каждого делителя $d|n$ его слагаемое $k^{\gcd(d, n)} = k^d$ учитывается несколько раз, т.е. сумму можно представить в таком виде:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} C_d k^d$$

где C_d — это количество таких чисел i , что $\gcd(i, n) = d$. Найдём явное выражение для этого количества. Любое такое число i имеет вид: $i = dj$, где $\gcd(j, n/d) = 1$ (иначе было бы $\gcd(i, n) > d$). Вспоминая [функцию Эйлера](#), мы находим, что количество таких j — это величина функции Эйлера $\phi(n/d)$. Таким образом, $C_d = \phi(n/d)$, и окончательно получаем **формулу**:

$$\text{Ans} = \frac{1}{n} \sum_{d|n} \phi\left(\frac{n}{d}\right) k^d$$

Применение леммы Бернсайда совместно с программными вычислениями

Далеко не всегда удается чисто аналитическим путём получить явную формулу для количества классов эквивалентности. Во многих задачах количество перестановок, входящих в группу, может быть слишком большим для ручных вычислений, и вычислить аналитически количество циклов в них не представляется возможным.

В таком случае следует вручную найти несколько "основных" перестановок, которых будет достаточно для порождения всей группы G . Далее можно написать программу, которая сгенерирует все перестановки группы G , посчитает в каждой из них количество циклов и подставит их в формулу.

Рассмотрим для примера **задачу о количестве раскрасок тора**. Имеется прямоугольный клетчатый лист бумаги $n \times m$ ($n < m$), некоторые из клеток покрашены в чёрный цвет. Затем из этого листа получают цилиндр, склеивая две стороны с длинами m . Затем из цилиндра получают тор, склеивая две окружности (базы цилиндра) без перекручивания. Требуется посчитать количество различных торов (лист был изначально покрашен произвольно), считая, что линии склеивания неразличимы, а тор можно поворачивать и переворачивать.

В данной задаче представлением можно считать лист бумаги $n \times m$, некоторые клетки которого покрашены в чёрный цвет. Нетрудно понять, что следующие виды преобразований сохраняют класс эквивалентности: циклический сдвиг строк листа, циклический сдвиг столбцов листа, поворот листа на 180 градусов; также интуитивно можно понять, что этих трёх видов преобразований достаточно для порождения всей группы инвариантных преобразований. Если мы каким-либо образом занумеруем клетки поля, то мы можем записать три перестановки

p_1, p_2, p_3 , соответствующие этим видам преобразований. Дальше остаётся только сгенерировать все перестановки, получающиеся как произведения этой. Очевидно, что все такие перестановки имеют вид $p_1^{i_1}p_2^{i_2}p_3^{i_3}$, где $i_1 = 0 \dots m - 1$, $i_2 = 0 \dots n - 1$, $i_3 = 0 \dots 1$.

Таким образом, мы можем написать реализацию решения этой задачи:

```
void mult (vector<int> & a, const vector<int> & b) {
    vector<int> aa (a);
    for (size_t i=0; i<a.size(); ++i)
        a[i] = aa[b[i]];
}

int cnt_cycles (vector<int> a) {
    int res = 0;
    for (size_t i=0; i<a.size(); ++i)
        if (a[i] != -1) {
            ++res;
            for (size_t j=i; a[j]!=-1; ) {
                size_t nj = a[j];
                a[j] = -1;
                j = nj;
            }
        }
    return res;
}

int main() {
    int n, m;
    cin >> n >> m;

    vector<int> p (n*m), p1 (n*m), p2 (n*m), p3 (n*m);
    for (int i=0; i<n*m; ++i) {
        p[i] = i;
        p1[i] = (i % n + 1) % n + i / n * n;
        p2[i] = (i / n + 1) % m * n + i % n;
        p3[i] = (m - 1 - i / n) * n + (n - 1 - i % n);
    }

    int sum = 0, cnt = 0;
    set <vector<int> > s;
    for (int i1=0; i1<n; ++i1) {
        for (int i2=0; i2<m; ++i2) {
            for (int i3=0; i3<2; ++i3) {
                if (!s.count(p)) {
                    s.insert (p);
                    ++cnt;
                    sum += 1 << cnt_cycles (p);
                }
                mult (p, p3);
            }
            mult (p, p2);
        }
        mult (p, p1);
    }

    cout << sum / cnt;
}
```

Игры на произвольных графах

Пусть игра ведётся двумя игроками на некотором графе G . Т.е. текущее состояние игры - это некоторая вершина графа, и из каждой вершины рёбра идут в те вершины, в которые можно пойти следующим ходом.

Мы рассматриваем самый общий случай - случай произвольного ориентированного графа с циклами. Требуется для заданной начальной позиции определить, кто выиграет при оптимальной игре обоих игроков (или определить, что результатом будет ничья).

Мы решим эту задачу очень эффективно - найдём ответы для всех вершин графа за линейное относительно количества рёбер время - $O(M)$.

Описание алгоритма

Про некоторые вершины графа заранее известно, что они являются выигрышными или проигрышными; очевидно, такие вершины не имеют

исходящих рёбер.

Имеем следующие факты:

- если из некоторой вершины есть ребро в проигрышную вершину, то эта вершина выигрышная;
- если из некоторой вершины все рёбра исходят в выигрышные вершины, то эта вершина проигрышная;
- если в какой-то момент ещё остались неопределённые вершины, но ни одна из них не подходит ни под первое, ни под второе правило, то все эти вершины - ничейные.

Таким образом, уже ясен алгоритм, работающий за асимптотику $O(NM)$ - мы перебираем все вершины, пытаемся к каждой применить первое либо второе правило, и если мы произвели какие-то изменения, то повторяем всё заново.

Однако этот процесс поиска и обновления можно значительно ускорить, доведя асимптотику до линейной.

Переберём все вершины, про которые изначально известно, что они выигрышные или проигрышные. Из каждой из них пустим следующий поиск в глубину. Этот поиск в глубину будет двигаться по обратным рёбрам. Прежде всего, он не будет заходить в вершины, которые уже определены как выигрышные или проигрышные. Далее, если поиск в глубину пытается пойти из проигрышной вершины в некоторую вершину, то её он помечает как выигрышную, и идёт в неё. Если же поиск в глубину пытается пойти из выигрышной вершины в некоторую вершину, то он должен проверить, все ли рёбра ведут из этой вершины в выигрышные. Этую проверку легко осуществить за $O(1)$, если в каждой вершине будем хранить счётчик рёбер, которые ведут в выигрышные вершины. Итак, если поиск в глубину пытается пойти из выигрышной вершины в некоторую вершину, то он увеличивает в ней счётчик, и если счётчик сравнялся с количеством рёбер, исходящих из этой вершины, то эта вершина помечается как проигрышная, и поиск в глубину идёт в эту вершину. Иначе же, если целевая вершина так и не определена как выигрышная или проигрышная, то поиск в глубину в ней не заходит.

Итого, мы получаем, что каждая выигрышная и каждая проигрышная вершина посещается нашим алгоритмом ровно один раз, а ничейные вершины и вовсе не посещаются. Следовательно, асимптотика действительно $O(M)$.

Реализация

Рассмотрим реализацию поиска в глубину, в предположении, что граф игры построен в памяти, степени исхода посчитаны и записаны в `degree` (это будет как раз счётчиком, он будет уменьшаться, если есть ребро в выигрышную вершину), а также изначально выигрышные или проигрышные вершины уже помечены.

```
vector<int> g [100];
bool win [100];
bool loose [100];
bool used[100];
int degree[100];

void dfs (int v) {
    used[v] = true;
    for (vector<int>::iterator i = g[v].begin(); i != g[v].end(); ++i)
        if (!used[*i]) {
            if (loose[v])
                win[*i] = true;
            else if (--degree[*i] == 0)
                loose[*i] = true;
            else
                continue;
            dfs (*i);
        }
}
```

Пример задачи. "Полицейский и вор"

Чтобы алгоритм стал более ясным, рассмотрим его на конкретном примере.

Условие задачи. Имеется поле размером $M \times N$ клеток, в некоторые клетки заходить нельзя. Известны начальные координаты полицейского и вора. Также на карте может присутствовать выход. Если полицейский окажется в одной клетке с вором, то выиграл полицейский. Если же вор окажется в клетке с выходом (и в этой клетке не стоит полицейский), то выигрывает вор. Полицейский может ходить в 8 направлениях, вор - только в 4 (вдоль осей координат). И полицейский, и вор могут пропустить свой ход. Первым ход делает полицейский.

Построение графа. Построим граф игры. Мы должны формализовать правила игры. Текущее состояние игры определяется координатами полицейского P , вора T , а также булева переменная $Pstep$, которая определяет, кто будет делать следующий ход. Следовательно, вершина графа определена тройкой $(P, T, Pstep)$. Граф построить легко, просто соответствуя условию.

Далее нужно определить, какие вершины являются выигрышными или проигрышными изначально. Здесь есть **тонкий момент**.

Выигрышность/проигрышность вершины помимо координат зависит и от $Pstep$ - чей сейчас ход. Если сейчас ход полицейского, то вершина выигрышная, если координаты полицейского и вора совпадают; вершина проигрышная, если она не выигрышная и вор находится на выходе. Если же сейчас ход вора, то вершина выигрышная, если вор находится на выходе, и проигрышная, если она не выигрышная и координаты полицейского и вора совпадают.

Единственный момент, который нужно решить - строить **граф явно или** делать это "**на ходу**", прямо в поиске в глубину. С одной стороны, если строить график предварительно, то будет меньше вероятность ошибиться. С другой стороны, это увеличит объём кода, да и время работы будет в несколько раз медленнее, чем если строить график "на ходу".

Реализация всей программы:

```
struct state {
    char p, t;
    bool pstep;
};

vector<state> g [100][100][2];
// 1 = policeman coords; 2 = thief coords; 3 = 1 if policeman's step or 0 if thief's.
bool win [100][100][2];
bool loose [100][100][2];
bool used[100][100][2];
int degree[100][100][2];
```

```

void dfs (char p, char t, bool pstep) {
    used[p][t][pstep] = true;
    for (vector<state>::iterator i = g[p][t][pstep].begin(); i != g[p][t][pstep].end(); ++i)
        if (!used[i->p][i->t][i->pstep]) {
            if (loose[p][t][pstep])
                win[i->p][i->t][i->pstep] = true;
            else if (--degree[i->p][i->t][i->pstep] == 0)
                loose[i->p][i->t][i->pstep] = true;
            else
                continue;
            dfs (i->p, i->t, i->pstep);
        }
}

int main() {

    int n, m;
    cin >> n >> m;
    vector<string> a (n);
    for (int i=0; i<n; ++i)
        cin >> a[i];

    for (int p=0; p<n*m; ++p)
        for (int t=0; t<n*m; ++t)
            for (char pstep=0; pstep<=1; ++pstep) {
                int px = p/m, py = p%m, tx=t/m, ty=t%m;
                if (a[px][py]=='*' || a[tx][ty]=='*') continue;

                bool & wwin = win[p][t][pstep];
                bool & lloose = loose[p][t][pstep];
                if (pstep)
                    wwin = px==tx && py==ty, lloose = !wwin && a[tx][ty] == 'E';
                else
                    wwin = a[tx][ty] == 'E', lloose = !wwin && px==tx && py==ty;
                if (wwin || lloose) continue;

                state st = { p, t, !pstep };
                g[p][t][pstep].push_back (st);
                st.pstep = pstep != 0;
                degree[p][t][pstep] = 1;

                const int dx[] = { -1, 0, 1, 0, -1, -1, 1, 1 };
                const int dy[] = { 0, 1, 0, -1, -1, 1, -1, 1 };
                for (int d=0; d<(pstep?8:4); ++d) {
                    int ppx=px, ppy=py, ttx=tx, tty=ty;
                    if (pstep)
                        ppx += dx[d], ppy += dy[d];
                    else
                        ttx += dx[d], tty += dy[d];
                    if (ppx>=0 && ppx<n && ppy>=0 && ppy<m && a[ppx][ppy]!='*' &&
                        ttx>=0 && ttx<n && tty>=0 && tty<m && a[ttx][tty]!='*')
                    {
                        g[ppx*m+ppy][ttx*m+tty][!pstep].push_back (st);
                        ++degree[p][t][pstep];
                    }
                }
            }
        }

    for (int p=0; p<n*m; ++p)
        for (int t=0; t<n*m; ++t)
            for (char pstep=0; pstep<=1; ++pstep)
                if ((win[p][t][pstep] || loose[p][t][pstep]) && !used[p][t][pstep])
                    dfs (p, t, pstep!=0);

    int p_st, t_st;
    for (int i=0; i<n; ++i)
        for (int j=0; j<m; ++j)
            if (a[i][j] == 'C')
                p_st = i*m+j;
            else if (a[i][j] == 'T')
                t_st = i*m+j;

    cout << (win[p_st][t_st][true] ? "WIN" : loose[p_st][t_st][true] ? "LOSS" : "DRAW");
}

```

Теория Шпрага-Гранди. Ним

Введение

Теория Шпрага-Гранди — это теория, описывающая так называемые **равноправные** (*impartial*) игры, т.е. игры, в которых разрешённые ходы и выигрышность/проигрышность зависят только от состояния игры, но не от того, какой игрок ходил. Иными словами, первый и второй игрок различаются только тем, что первый игрок первым совершает ход, а в остальном они равноправны.

Кроме того, предполагается, что игроки располагают всей информацией (в том числе и о положении соперника).

Предполагается, что игра **конечна**, т.е. при любой игре игроки рано или поздно придут в позицию, из которой нет переходов в другие позиции. Эта позиция является проигрышной для игрока, который должен делать ход из этой позиции. Соответственно, она является выигрышной для игрока, пришедшего в эту позицию. Понятно, ничейных исходов в такой игре не бывает.

Иными словами, такую игру можно полностью описать **ориентированным ациклическим графом**: вершинами в нём являются состояния игры, а рёбрами — переходы из одного состояния игры в другое в результате хода текущего игрока (повторимся, в этом первой и второй игрок равноправны). Одни или несколько вершин не имеют исходящих рёбер, они являются проигрышными вершинами (для игрока, который должен совершать ход из такой вершины).

Поскольку ничейных исходов не бывает, то все состояния игры распадаются на два класса: **выигрышные и проигрышные**. Выигрышные — это такие состояния, что найдётся ход текущего игрока, который приведёт к поражению другого игрока даже при его оптимальной игре. Соответственно, проигрышные состояния — это состояния, из которых все переходы ведут в состояния, приводящие к победе второго игрока, несмотря на "сопротивление" первого игрока. Иными словами, выигрышным будет состояние, из которого есть хотя бы один переход в проигрышное состояние, а проигрышным является состояние, из которого все переходы ведут в выигрышные состояния (или из которого вообще нет переходов).

Наша задача — для любой заданной игры провести классификацию состояний этой игры, т.е. для каждого состояния определить, выигрышное оно или проигрышное.

Теорию таких игр независимо разработали Роланд Шпраг (Roland Sprague) в 1935 г. и Патрик Майкл Гранди (Patrick Michael Grundy) в 1939 г.

Игра "Ним"

Эта игра является одним из примеров описываемых выше игр. Однако, как мы увидим чуть позже, **любая** из рассматриваемых нами игр на самом деле эквивалентна игре "ним" (nim), поэтому изучение этой игры автоматически позволяет нам решать все остальные игры.

Игра "ним" представляет собой следующее. Есть несколько кучек, в каждой из которых по несколько камней (как минимум по одному). За один ход игрок может взять из любой кучки любое строго положительное число камней и выбросить их. Соответственно, когда все кучки опустеют, корректных ходов больше не будет, и текущий игрок проигрывает.

Итак, состояние игры ним однозначно описывается неупорядоченным набором натуральных чисел. За один ход разрешается строго уменьшить любое из чисел (если в результате число станет нулем, то оно удаляется из набора).

Решение этой игры опубликовал в 1901 г. Чарльз Бутон (Charles Bouton).

Теорема. Текущий игрок имеет выигрышную стратегию тогда и только тогда, когда XOR-сумма размеров кучек отлична от нуля. В противном случае текущий игрок находится в проигрышном состоянии. (XOR-суммой чисел a_i называется выражение $a_1 \oplus a_2 \oplus \dots \oplus a_n$, где \oplus — операция побитового исключающего или)

Доказательство. Обозначим через $x_k, k = 1 \dots n$ размеры кучек до хода игрока, а через y_k — после хода. Понятно, что эти два вектора различаются ровно в одном элементе. Обозначим через s и t XOR-суммы до и после хода игрока, т.е.:

$$\begin{aligned} s &= x_1 \oplus \dots \oplus x_n \\ t &= y_1 \oplus \dots \oplus y_n \end{aligned}$$

Тогда, по свойству операции \oplus , можем записать:

$$t = s \oplus x_p \oplus y_p$$

где p — номер кучки, в которой совершил ход игрок.

Доказывать теорему будем теперь по индукции. Предполагаем, что теорема верна для всех состояний, в которые мы можем перейти из текущего состояния (поскольку игра ациклична, то это предположение корректно). Тогда доказательство распадается на две части: если $s = 0$, то надо доказать, что текущее состояние проигрышно, т.е. все переходы из него ведут в состояния с $t \neq 0$. Если же $s \neq 0$, то надо доказать, что найдётся переход, приводящий нас в состояние с $t = 0$.

- Пусть $s = 0$. Если из текущего состояния нет переходов, то доказывать нечего (текущее состояние и так проигрышно). Иначе рассмотрим любой переход. Тогда по вышеприведённой формуле получаем для него:

$$t = x_p \oplus y_p$$

Но поскольку $x_p \neq y_p$, то $t \neq 0$, что и означает, что новое состояние будет выигрышным, что и требовалось доказать.

- Пусть $s \neq 0$. Покажем, какой ход надо совершить, чтобы прийти в проигрышное состояние (т.е. с $t = 0$).

Рассмотрим битовую запись числа s . Возьмём старший ненулевой бит, пусть его номер равен d . Пусть k — номер того из чисел x_k , у которого d -ый бит отличен от нуля (такой найдётся, поскольку в их XOR-сумме s этот бит отличен от нуля). Тогда положим $y_k = x_k \oplus s$; утверждается, что это и есть искомый ход.

Сначала надо проверить, что это ход корректный, т.е. что $y_k < x_k$. Однако это верно, поскольку все биты, старшие d -го, у x_k и y_k совпадают, а в d -ом бите у y_k будет ноль, а у x_k будет единица.

Теперь посчитаем, какая XOR-сумма получится при этом ходе:

$$t = s \oplus x_k \oplus y_k = s \oplus x_k \oplus (s \oplus x_k) = 0$$

Таким образом, действительно существует ход, приводящий в проигрышное состояние, что и требовалось доказать.

Следствие. Любое состояние ним-игры можно заменить эквивалентным состоянием, состоящим из единственной кучки размера, равного XOR-сумме размеров кучек в старом состоянии.

Эквивалентность любой игры ниму. Теорема Шпрага-Гранди

Здесь мы покажем, как любой (из класса рассматриваемых нами) игре поставить в соответствие ним. Иными словами, любому состоянию любой игры можно поставить в соответствие ним-кучку. Если размер кучки равен нулю, то текущее состояние проигрышно, иначе выигрышно.

Лемма (о ниме с увеличениями). Рассмотрим следующий модифицированный ним: позиция по-прежнему задаётся набором натуральных чисел, однако за один ход теперь можно помимо произвольного уменьшения одного из чисел, также увеличивать некоторые из чисел на некоторые указанные величины (но при этом правила увеличения таковы, что игра остаётся корректной, т.е. никогда не станет бесконечной). Утверждается, что такой модифицированный ним эквивалентен обычному ниму, причём в том же состоянии. **Доказательство.** Фактически нам надо доказать, что наличие дополнительных увеличивающих ходов ничего не меняет. Действительно, пусть текущее состояние выигрышно; тогда для текущего игрока нет никакого смысла использовать этот ход (ведь если текущий игрок увеличит какое-то из чисел, то второй игрок всегда сможет ответить, уменьшив это число обратно до старого значения). Если же текущее состояние проигрышно, то тем более наличие или отсутствие этого хода ничего не может изменить. Что и требовалось доказать.

Теорема Шпрага-Гранди. Рассмотрим любое состояние G некоторой игры; пусть из него есть переходы в некоторые состояния $G_i, i = 1 \dots k$. Утверждается, что состоянию G можно поставить в соответствие кучку нима некоторого размера m (и эти две игры будут эквивалентны). Более того, это число m можно находить индуктивно: если мы найдём эти числа m_i для каждого из новых состояний G_i , то m равно:

$$m = \text{mex}\{m_1, \dots, m_k\}$$

(результатом операции **mex** (minimum excludant) от множества чисел является наименьшее неотрицательное число, не встречающееся в этом множестве)

Доказательство. Доказывать теорему будем по индукции: пусть она верна для всех новых состояний G_i . Докажем её для текущего состояния G . Положим

$$m = \text{mex}\{m_1, \dots, m_k\}$$

и докажем, что состояние игры ним с единственной кучкой размера m эквивалентно состоянию G нашей игры.

Поскольку m определялось как **mex** ним-чисел, соответствующих новым состояниям, то получается, что для каждого $0 \leq m' < m$ существует переход из G в некоторое состояние G' , которому эквивалентна ним-игра с единственной кучкой m' . Кроме того, по определению операции **mex**, могут существовать и переходы в состояния, которым соответствуют $m' > m$.

Таким образом, получается, что состояние G эквивалентно ниму с увеличениями с единственной кучкой размера m : ведь мы можем произвести любое уменьшение числа m , но и, возможно, можем произвести некоторые увеличения. Но по предыдущей лемме, ним с увеличениями эквивалентен обычному ниму в том же состоянии.

Таким образом, состояние G нашей игры эквивалентно ниму с единственной кучкой размера m , что и требовалось доказать.

Применение теоремы Шпрага-Гранди

Опишем наконец целостный алгоритм, применимый к любой игре (из рассматриваемого нами класса) для определения выигрышности/проигрышности текущего состояния G .

Функция, которая каждому состоянию игры ставит в соответствие ним-число, называется **функцией Гранди**.

- Если из текущего состояния есть переходы в состояния $G_i, i = 1 \dots k$, то для каждого из них (фактически, рекурсивно) вычисляем функцию Гранди m_i . Тогда функция Гранди для состояния G будет равна наименьшему неотрицательному числу, не совпадающему ни с одним из чисел m_i :

$$m = \text{mex}\{m_1, \dots, m_k\}$$

- Если текущее состояние представляет собой сумму независимых игр (как, например, ним из нескольких кучек представляет собой сумму нимов из каждой кучки по отдельности), то опять же, вычисляем функцию Гранди m_i для каждой из этих игр в отдельности. Функция Гранди для состояния G будет равна XOR-сумме этих чисел m_i :

$$m = m_1 \oplus \dots \oplus m_k$$

Первое применение непосредственно вытекает из теоремы Шпрага-Гранди. Второе применение получается следующим образом: сначала по теореме Шпрага-Гранди каждую из независимых "под-игр" заменяем ним-кучкой; затем по теореме Бутона находим ним-число для нима из этих нескольких кучек.

Также следует отметить следующий момент. Для **больших игр**, т.е. игр, в которых число состояний слишком велико, чтобы для каждого посчитать функцию Гранди, на помощь приходит следующий факт. Очень часто в выписанных таблицах первых значений функции Гранди можно найти определённые закономерности, которые сохраняются и для всей бесконечной таблицы. Кроме того, очень часто функция Гранди оказывается периодичной.

Литература

- John Horton Conway. *On numbers and games* [1979]

Задача Джонсона при $N = 1$

Постановка задачи

Пусть имеется M деталей, которые нужно обработать на одном станке. Про каждую i -ую деталь известно, что она обрабатывается станком за T_i единиц времени, а за простой i -ой заявки начисляется штраф, который равен значению некоторой функции $F_i(t)$. Требуется найти такой порядок обработки деталей, чтобы минимизировать полученный штраф.

Математическая модель

Пусть $R = (i_1, i_2, \dots, i_M)$ - некоторая перестановка из N натуральных чисел, которая определяет порядок обработки заявок. Обозначим через $F(R)$ суммарный штраф, получаемый за обработку заявок в порядке, определяемом перестановкой R . Тогда имеем:

$$F(R) = F_{i_1}(T_{i_1}) + F_{i_2}(T_{i_2}) + \dots + F_{i_M}(T_{i_M})$$

Теперь задача Джонсона ставится как задача поиска такой перестановки, на которой достигает минимальное значение критерий задачи F .

Случай линейных функций штрафов

Пусть функции штрафов линейные, т.е. имеют вид:

$$F_i(t) = C_i t$$

Рассмотрим две перестановки $R = (1, 2, \dots, i, i+1, \dots, M)$ и $Q = (1, 2, \dots, i+1, i, \dots, M)$, т.е. в них поменяли местами i и $i+1$ заявки.

Тогда нетрудно показать, что $F(R) - F(Q) = C_{i+1} T_i - C_i T_{i+1}$.

Понятно, что если эта разность положительна, то $F(R) > F(Q)$, и заявку с номером $i+1$ нужно обслужить раньше заявки с номером i .

Теперь понятен алгоритм решения этой задачи в случае линейных функций штрафов:

Решение с помощью перестановочного приёма

Вариант 1.

Сортируем заявки с помощью любого алгоритма сортировки, при этом операцию сравнения двух заявок определяем следующим образом: если $C_j T_i < C_i T_j$, то заявка с номером j должна идти раньше заявки с номером i , иначе наоборот.

Вариант 2.

Сортируем заявки по неубыванию отношения C_i / T_i .

Сложность алгоритма.

Очевидно, что при использовании быстрой сортировки сложность алгоритма составляет $O(M \log M)$.

Теорема Лившица-Кладова

Теорема Лившица-Кладова утверждает, что **перестановочный приём применим для следующих функций штрафа и только для них:**

- $F_i(t) = C_i t + B_i$
- $F_i(t) = C_i e^{At} + B_i$, где $A > 0$
- $F_i(t)$ - монотонно возрастающие функции

Случай остальных функций

Для случая функций, не удовлетворяющих условиям теоремы Лившица-Кладова, столь же эффективного алгоритма не существует. В таких случаях эту задачу приходится решать методом линейного программирования.

Реализация решения перестановочным приёмом

Здесь я приведу реализацию решения перестановочным приёмом в его первом варианте.

В данном случае функция штрафа является линейной. Входными данными являются T_i и C_i .

```
struct item
{
    int c, t;
    int index;
};

bool operator < (const item & a, const item & b)
{
    return (a.t * b.c < b.t * a.c);
}

int main()
{
    int n;
    cin >> n;
    vector<item> a (n);
    for (int i=0; i<n; ++i)
    {
```

```

    cin >> a[i].t >> a[i].c;
    a[i].index = i+1;
}

sort (a.begin(), a.end());

for (int i=0; i<n; ++i)
    cout << a[i].index << ' ';
}

```

Задача Джонсона при $N = 2$

Постановка задачи

Пусть имеется M деталей, которые нужно обработать на двух станках: каждую деталь сначала на первом, а затем на втором станке. Про каждую i -ю деталь известно, что она обрабатывается первым станком за A_i единиц времени, а вторым станком - за B_i единиц. Кроме того, порядок обработки деталей на первом станке и на втором станке должен совпадать. Требуется найти такой порядок обработки деталей, чтобы потратить наименьшее время.

Математическая модель

Обозначим через X_i время простоя второго станка непосредственно перед началом обработки детали с номером i .

Тогда критерием оптимальности задачи Джонсона при $N = 2$ станет функционал:

$$F(X) = \text{СУММА } X_i \Rightarrow \min.$$

Математическое решение задачи

Расписание обработки деталей на станках задаётся перестановкой R натуральных чисел от 1 до M .

Например, при $R = (1, 2, \dots, M)$ мы имеем:

$$X_1 = A_1$$

$$X_2 = \max(A_1 + A_2 - B_1 - X_1, 0)$$

...

и тогда

$$F(X) = \text{СУММА } X_i = \max($$

$$A_1 + \dots + A_M - B_1 - \dots - B_{M-1},$$

...,

$$A_1 + \dots + A_i - B_1 - \dots - B_{i-1},$$

...,

$$A_1)$$

Теперь рассмотрим две перестановки $R = (1, 2, \dots, i, i+1, \dots, M)$ и $Q = (1, 2, \dots, i+1, i, \dots, M)$, т.е. поменяли местами детали i и $i+1$.

Предположим, что $F(R) > F(Q)$. Тогда имеем:

$\max($

$$A_1 + \dots + A_i - B_1 - \dots - B_{i-1},$$

$$A_1 + \dots + A_{i+1} - B_1 - \dots - B_i,$$

$) > \max($

$$A_1 + \dots + A_{i-1} + A_{i+1} - B_1 - \dots - B_{i-1},$$

$$A_1 + \dots + A_{i+1} - B_1 - \dots - B_{i-1} - B_{i+1},$$

)

Теперь вычтем из правой и левой частей величину $A_1 + \dots + A_{i+1} - B_1 - \dots - B_{i-1}$. После несложных преобразований получим:

$$\min(A_{i+1}, B_i) < \min(A_i, B_{i+1})$$

Итак, последнее неравенство выполняется тогда и только тогда, когда $F(R) > F(Q)$, т.е. деталь с номером $i+1$ должна обрабатываться раньше детали с номером i .

Описание алгоритма решения задачи

Вариант 1.

Сортируем детали, при этом операцию сравнения i -ой и $i+1$ -ой деталей определяем следующим образом:

если $\min(A_{i+1}, B_i) < \min(A_i, B_{i+1})$, то деталь с номером $i+1$ должна идти раньше, иначе - наоборот.

Вариант 2. (непосредственно следует из первого варианта)

1. Найти минимальную величину среди всех A_i и B_i .

2. Если минимум достигается на A_i , то деталь с номером i нужно ставить на обработку самой первой. Если же на B_i , то деталь с номером i ставится на обработку самой последней.

3. Найденная деталь исключается из рассмотрения. Переходим к шагу 1.

Этот, второй, вариант алгоритма называется алгоритмом Джонсона.

Сложность алгоритма Джонсона, очевидно, составляет $O(N \log N)$ при использовании быстрой сортировки.

Реализация алгоритма Джонсона

Здесь я приведу пример реализации алгоритма Джонсона.

Входные данные - A_i и B_i .

Помимо искомого порядка деталей, программа также вычисляет минимальное время, требуемое для обработки всех деталей.

```
int main()
{
    int n;
    cin >> n;

    vector<pair<double, double>> a(n);
    for (int i=0; i<n; ++i)
        cin >> a[i].first >> a[i].second;

    vector<pair<double, pair<int, int>> b;
    b.reserve(n*2);
    for (int i=0; i<n; ++i)
    {
        b.push_back(make_pair(a[i].first, make_pair(i, 0)));
        b.push_back(make_pair(a[i].second, make_pair(i, 1)));
    }

    sort(b.begin(), b.end());

    vector<char> used(n);
    vector<int> res_time[2];
    res_time[0].reserve(n);
    res_time[1].reserve(n);
    for (int i=0; i<n*2; ++i)
    {
        if (!used[b[i].second.first])
        {
            used[b[i].second.first] = true;
            res_time[b[i].second.second].push_back(b[i].second.first);
        }
    }

    vector<int> result;
    result.reserve(n);
    copy(res_time[0].begin(), res_time[0].end(), back_inserter(result));
    copy(res_time[1].rbegin(), res_time[1].rend(), back_inserter(result));

    double time1 = 0, time2 = 0;
    for (int i=0; i<n; ++i)
    {
        int cur = result[i];
        time1 += a[cur].first;
        if (time2 < time1)
            time2 = time1;
        time2 += a[cur].second;
    }
    double res_time = max(time1, time2);

    printf("%.5lf\n", res_time);
    for (int i=0; i<n; ++i)
        printf("%d ", result[i]+1);
    printf("\n");
}
```

Оптимальный выбор заданий при известных временах завершения и длительностях выполнения

Пусть дан набор заданий, у каждого задания известен момент времени, к которому это задание нужно завершить, и длительность выполнения этого задания. Процесс выполнения какого-либо задания нельзя прерывать до его завершения. Требуется составить такое расписание, чтобы выполнить наибольшее число заданий.

Решение

Алгоритм решения - жадный. Отсортируем все задания по их крайнему сроку, и будем рассматривать их по очереди в порядке убывания крайнего срока. Также создадим некую структуру данных Q , в которую мы будем постепенно помещать задания, и извлекать из структуры задание с наименьшим временем выполнения (например, можно использовать `set` или `priority_queue`). Изначально Q пустая. Пусть мы рассматриваем i -ое задание. Сначала поместим его в Q . Рассмотрим отрезок времени между сроком завершения i -го задания и сроком завершения $(i-1)$ -го задания - отрезок некоторой длины T . Будем извлекать из Q задания и помещать на выполнение в этом отрезке, пока отрезок T не станет равным нулю. Важный момент - если в какой-то момент времени очередное извлечённое из структуры задание можно успеть частично выполнить в отрезке T , то мы выполняем это задание частично - именно настолько, насколько это возможно, т.е. в течение T единиц времени, а оставшуюся часть задания помещаем обратно в Q . По окончании этого алгоритма мы выберем оптимальное решение (или, по крайней мере, одно из нескольких решений).

Итоговая асимптотика алгоритма - $O(N \log N)$.

Реализация

```
int n;
vector<pair<int,int>> a; // пары крайний срок - длительность
... чтение n и a ...

sort(a.begin(), a.end());

typedef set<pair<int,int>> t_s;
t_s s;
vector<int> result;
for (int i=n-1; i>=0; --i) {
    int t = a[i].first - (i ? a[i-1].first : 0);
    s.insert (make_pair (a[i].second, i));
    while (t && !s.empty()) {
        t_s::iterator it = s.begin();
        if (it->first <= t) {
            t -= it->first;
            result.push_back (it->second);
        }
        else {
            s.insert (make_pair (it->first - t, it->second));
            t = 0;
        }
        s.erase (it);
    }
}

for (size_t i=0; i<result.size(); ++i)
    cout << result[i] << ' ';
```

Задача Иосифа

Условие задачи. Даны натуральные n и k . По кругу выписывают все натуральные числа от 1 до n . Сначала отсчитывают k -ое число, начиная с первого, и удаляют его. Затем от него отсчитывают k чисел и k -ое удаляют, и т.д. Процесс останавливается, когда остаётся одно число. Требуется найти это число.

Задача была поставлена **Иосифом Флавием** (Flavius Josephus) ещё в 1 веке (правда, в несколько более узкой формулировке: при $k = 2$). Решать эту задачу можно моделированием. Простейшее моделирование будет работать $O(n^2)$. Используя [Дерево отрезков](#), можно произвести моделирование за $O(n \log n)$.

Решение за $O(n)$

Попытаемся найти закономерность, выражающую ответ для задачи $J_{n,k}$ через решение предыдущих задач.

С помощью моделирования построим таблицу значений, например, такую:

$n \setminus k$	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	2	1	2	1	2	1	2	1	2	1
3	3	3	2	2	1	1	3	3	2	2
4	4	1	1	2	2	3	2	3	3	4
5	5	3	4	1	2	4	4	1	2	4
6	6	5	1	5	1	4	5	3	5	2
7	7	7	4	2	6	3	5	4	7	5
8	8	1	7	6	3	1	4	4	8	7
9	9	3	1	1	8	7	2	3	8	8
10	10	5	4	5	3	3	9	1	7	8

И здесь достаточно отчётливо видна следующая **закономерность**:

$$\begin{aligned} J_{n,k} &= (J_{(n-1),k} + k - 1) \% n + 1 \\ J_{1,k} &= 1 \end{aligned}$$

Здесь 1-индексация несколько портит элегантность формулы, если нумеровать позиции с нуля, то получится очень наглядная формула:

$$J_{n,k} = (J_{(n-1),k} + k) \% n = \sum_{i=1}^n k \% i$$

Итак, мы нашли решение задачи Иосифа, работающее за $O(n)$ операций.

Простая **рекурсивная реализация** (в 1-индексации):

```
int joseph (int n, int k) {
    return n>1 ? (joseph (n-1, k) + k - 1) % n + 1 : 1;
}
```

Нерекурсивная форма:

```
int joseph (int n, int k) {
    int res = 0;
    for (int i=1; i<=n; ++i)
        res = (res + k) % i;
    return res + 1;
}
```

Решение за $O(k \log n)$

Для сравнительно небольших k можно придумать более оптимальное решение, чем рассмотренная выше динамика за $O(n)$. Если k небольшое, то даже интуитивно понятно, что тот алгоритм делает много лишних действий: рассматривая числа $k, 2k, 3k, \dots$, и лишь только когда он достигнет n и совершил переход по кругу из числа n в число 1, — только тогда происходят сильные изменения, до этого же алгоритм просто шагает шагами длины k . Соответственно, попытаемся избавиться от этих ненужных шагов, т.е. научимся удалять сразу все числа, кратные k , за одну операцию.

Сложность здесь в том, что после удаления этих чисел у нас получится задача с меньшим n , но стартовой позицией не в первом числе, а, возможно, где-то в другом месте. Поэтому, вызывая рекурсивно себя от задачи с новым n , мы затем должны перевести результат в нашу систему нумерации из его собственной. Впрочем, сложности здесь чисто технические — надо аккуратно и кратко реализовать этот пересчёт.

Также отдельно надо разбирать случай, когда n станет меньше k . В этом случае вышеописанная оптимизация выродится в бесконечный цикл, поэтому для разбора этого случая можно просто взять решение старой динамикой.

Реализация (для удобства в 0-индексации):

```
int joseph (int n, int k) {
    if (n == 1) return 0;
    if (k == 1) return n-1;
    if (k > n) return (joseph (n-1, k) + k) % n;
    int cnt = n / k;
    int res = joseph (n - cnt, k);
    res -= n % k;
    if (res < 0) res += n;
    else res += res / (k - 1);
    return res;
}
```

Оценим **асимптотику** этого алгоритма. Сразу заметим, что случай $n < k$ разбирается у нас старым решением, которое отработает в данном случае за $O(k)$. Теперь рассмотрим сам алгоритм. Фактически, на каждой его итерации вместо n чисел мы получаем примерно $n \left(1 - \frac{1}{k}\right)$ чисел, поэтому общее число x итераций алгоритма примерно можно найти из уравнения:

$$n \left(1 - \frac{1}{k}\right)^x = 1,$$

логарифмируя его, получаем:

$$\ln n + x \ln \left(1 - \frac{1}{k}\right) = 0,$$
$$x = -\frac{\ln n}{\ln \left(1 - \frac{1}{k}\right)},$$

пользуясь разложением логарифма в ряд Тейлора, получаем приблизительную оценку:

$$x \approx k \ln n$$

Таким образом, асимптотика алгоритма действительно $O(k \log n)$.

Аналитическое решение для $k = 2$

В этом частном случае (в котором и была поставлена эта задача Иосифом Флавием) задача решается значительно проще.

В случае чётного n получаем, что будут вычеркнуты все чётные числа, а потом останется задача для $\frac{n}{2}$, тогда ответ для n будет получаться из ответа для $\frac{n}{2}$ умножением на два и вычитанием единицы (за счёт сдвига позиций):

$$J_{2n,2} = 2J_{n,2} - 1$$

Аналогично, в случае нечётного n будут вычеркнуты все чётные числа, затем первое число, и останется задача для $\frac{n-1}{2}$, и с учётом сдвига позиций получаем вторую формулу:

$$J_{2n+1,2} = 2J_{n,2} + 1$$

При реализации можно непосредственно использовать эту рекуррентную зависимость. Можно эту закономерность перевести в другую форму: $J_{n,2}$ представляют собой последовательность всех нечётных чисел, "перезапускающуюся" с единицы всякий раз, когда n оказывается степенью двойки. Это можно записать и в виде одной формулы:

$$J_{n,2} = 1 + 2^{\lfloor \log_2 n \rfloor}$$

Аналитическое решение для $k > 2$

Несмотря на простой вид задачи и большое количество статей по этой и смежным задачам, простого аналитического представления решения задачи Иосифа до сих пор не найдено. Для небольших k выведены некоторые формулы, но, по-видимому, все они трудноприменимы на практике (например, см. Halbeisen, Hungerbuhler "The Josephus Problem" и Odlyzko, Wilf "Functional iteration and the Josephus problem").

Игра Пятнашки: существование решения

Напомним, что игра представляет собой поле 4 на 4, на котором расположены 15 фишечек, пронумерованных числами от 1 до 15, а одно поле оставлено пустым. Требуется, передвигая на каждом шаге какую-либо фишку на свободную позицию, прийти в конце концов к следующей позиции:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	○

Игру Пятнашки ("15 puzzle") изобрёл в 1880 г. Нойес Чэпман (Noyes Chapman).

Существование решения

Здесь мы рассмотрим такую задачу: по данной позиции на доске сказать, существует ли последовательность ходов, приводящая к решению, или нет.

Пусть дана некоторая позиция на доске:

a_1	a_2	a_3	a_4
a_5	a_6	a_7	a_8
a_9	a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}	a_{16}

где один из элементов равен нулю и обозначает пустую клетку $a_z = 0$.

Рассмотрим перестановку:

$$a_1 a_2 \dots a_{z-1} a_{z+1} \dots a_{15} a_{16}$$

(т.е. перестановка чисел, соответствующая позиции на доске, без нулевого элемента)

Обозначим через N количество инверсий в этой перестановке (т.е. количество таких элементов a_i и a_j , что $i < j$, но $a_i > a_j$).

Далее, пусть K — номер строки, в которой находится пустой элемент (т.е. в наших обозначениях $K = (z - 1) \text{ div } 4 + 1$).

Тогда, **решение существует тогда и только тогда, когда $N + K$ чётно.**

Реализация

Проиллюстрируем указанный выше алгоритм с помощью программного кода:

```
int a[16];
for (int i=0; i<16; ++i)
    cin >> a[i];

int inv = 0;
for (int i=0; i<16; ++i)
    if (a[i])
        for (int j=0; j<i; ++j)
            if (a[j] > a[i])
                ++inv;
for (int i=0; i<16; ++i)
    if (a[i] == 0)
        inv += 1 + i / 4;

puts ((inv & 1) ? "No Solution" : "Solution Exists");
```

Доказательство

Джонсон (Johnson) в 1879 г. доказал, что если $N + K$ нечётно, то решения не существует, а Стори (Story) в том же году доказал, что все позиции, для которых $N + K$ чётно, имеют решение.

Однако оба эти доказательства были достаточно сложны.

В 1999 г. Арчер (Archer) предложил значительно более простое доказательство (скачать его статью можно [здесь](#)).

Дерево Штерна-Броко. Ряд Фарея

Дерево Штерна-Броко

Дерево Штерна-Броко — это изящная конструкция, позволяющая построить множество всех неотрицательных дробей. Она была независимо открыта немецким математиком Морицем Штерном (Moritz Stern) в 1858 г. и французским часовщиком Ахиллом Броко (Achille Brocot) в 1861 г. Впрочем, по некоторым данным, эта конструкция была открыта ещё древнегреческим учёным Эратосфеном (Eratosthenes).

На нулевой итерации у нас есть две дроби:

$$\frac{0}{1}, \frac{1}{0}$$

(вторая величина, строго говоря, дробью не является; её можно понимать как несократимую дробь, обозначающую бесконечность)

Дальше, на каждом **последующей** итерации берётся этот список дробей и между каждыми двумя соседними дробями $\frac{a}{b}$ и $\frac{c}{d}$ вставляется их **медианта**, т.е. дробь $\frac{a+c}{b+d}$.

Так, на первой итерации текущее множество будет таким:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

На второй:

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

На третьей:

$$\begin{array}{ccccccccc} 0 & 1 & 1 & 2 & 1 & 3 & 2 & 3 & 1 \\ \hline 1 & 3 & 2 & 3 & 1 & 2 & 1 & 1 & 0 \end{array}$$

Продолжая этот процесс до **бесконечности**, утверждается, можно получить множество **всех** неотрицательных дробей. Более того, все получаемые дроби будут **различными** (т.е. в текущем множестве каждая дробь встречается не более одного раза), **несократимыми** (числители и знаменатели будут получаться взаимно простыми). Наконец, все дроби будут автоматически **упорядоченными** по возрастанию. Доказательство всех этих замечательных свойств дерева Штерна-Броко будет приведено чуть ниже.

Осталось только привести изображение самого дерева Штерна-Броко (пока мы описывали его с помощью меняющегося множества). В корне этого бесконечного дерева находится дробь $\frac{1}{1}$, а слева и справа от дерева находятся дроби $\frac{0}{1}$ и $\frac{1}{0}$. Любая вершина дерева имеет двух сыновей, каждый из которых получается как медианта своего левого предка и правого предка:

□

Доказательство

Упорядоченность. Она доказывается очень просто: заметим, что медианта двух дробей всегда находится между ними, т.е.:

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

при условии, что

$$\frac{a}{b} \leq \frac{c}{d}$$

Доказывается это просто приведением трёх дробей к общему знаменателю.

Поскольку на нулевой итерации упорядоченность имела место, то она будет сохраняться и на каждой новой итерации.

Несократимость. Для этого покажем, что на любой итерации для любых двух соседних в списке дробей $\frac{a}{b}$ и $\frac{c}{d}$ выполняется:

$$bc - ad = 1$$

Действительно, вспоминая [Диофантовы уравнения с двумя неизвестными](#) ($ax + by = c$), получаем из этого утверждения, что $\gcd(a, b) = \gcd(c, d) = 1$, что нам и требуется.

Итак, нам надо доказать истинность утверждения $bc - ad = 1$ на любой итерации. Докажем его также по индукции. На нулевой итерации это свойство выполнялось (в чём нетрудно убедиться). Теперь пусть оно было выполнено на предыдущей итерации, покажем, что оно выполнено на текущей итерации. Для этого надо рассмотреть тройку дробей-соседей в новом списке:

$$\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$$

Для них условия принимают вид:

$$\begin{aligned} b(a+c) - a(b+d) &= 1, \\ c(b+d) - d(a+c) &= 1 \end{aligned}$$

Однако истинность этих условий очевидна, при условии истинности $bc - ad = 1$. Таким образом, действительно, это свойство выполнено и на текущей итерации, что и требовалось доказать.

Наличие всех дробей. Доказательство этого свойства тесно связано с алгоритмом нахождения дроби в дереве Штерна-Броко. Учитывая, что в дереве Штерна-Броко все дроби упорядочены, получаем, что для любой вершины дерева в её левом поддереве находятся дроби, меньшие её, а в правом — большие её. Отсюда получаем очевидный алгоритм поиска какой-либо дроби в дереве Штерна-Броко: вначале мы находимся в корне; сравниваем нашу дробь с дробью, записанной в текущей вершине: если наша дробь меньше, то переходим в левое поддерево, если наша дробь больше — переходим в правое, а если совпадает — нашли дробь, поиск завершён.

Чтобы доказать, что бесконечное дерево Штерна-Броко содержит все дроби, достаточно показать, что этот алгоритм поиска дроби завершится за конечное число шагов для любой заданной дроби. Этот алгоритм можно понимать так: у нас есть текущий отрезок $\left[\frac{a}{b}; \frac{c}{d}\right]$, в котором мы ищем нашу дробь $\frac{x}{y}$. Изначально $\frac{a}{b} = \frac{0}{1}$, $\frac{c}{d} = \frac{1}{0}$. На каждом шаге дробь $\frac{x}{y}$ сравнивается с медиантой концов отрезка, т.е. с $\frac{a+c}{b+d}$, и в зависимости от этого мы либо останавливаем поиск, либо переходим в левую или правую часть отрезка. Если бы алгоритм поиска дроби работал бесконечно долго, то следующие условия были бы выполнены на каждой итерации:

$$\frac{a}{b} < \frac{x}{y} < \frac{c}{d}$$

Но их можно переписать в таком виде:

$$\begin{aligned} bx - ay &\geq 1, \\ cy - dx &\geq 1 \end{aligned}$$

(здесь использовалось то, что они целочисленны, поэтому из > 0 следует ≥ 1)

Тогда, умножая первое на $c+d$, а второе — на $a+b$, и складывая их, получаем:

$$(c+d)(bx - ay) + (a+b)(cy - dx) \geq a + b + c + d$$

Раскрывая скобки слева и учитывая, что $bc - ad = 1$ (см. доказательство предыдущего свойства), окончательно получаем:

$$x + y \geq a + b + c + d$$

А поскольку на каждой итерации хотя бы одна из переменных a, b, c, d строго возрастает, то процесс поиска дроби $\frac{x}{y}$ будет содержать не более $x + y$ итераций, что и требовалось доказать.

Алгоритм построения дерева

Чтобы построить любое поддерево дерева Штерна-Бреко, достаточно знать только левого и правого предков. Изначально, на первом уровне, левым предком является $\frac{0}{1}$, а правым — $\frac{1}{0}$. По ним можно вычислить дробь в текущей вершине, а затем запуститься от левого и правого сыновей (левому сыну передав себя в качестве правого предка, а правому сыну — в качестве левого предка).

Псевдокод этой процедуры, пытающейся построить всё бесконечное дерево:

```
void build (int a = 0, int b = 1, int c = 1, int d = 0, int level = 1) {
    int x = a+c, y = b+d;
    ... вывод текущей дроби x/y на уровне дерева level
    build (a, b, x, y, level + 1);
    build (x, y, c, d, level + 1);
}
```

Алгоритм поиска дроби

Алгоритм поиска дроби был уже описан при доказательства того, что дерево Штерна-Бреко содержит все дроби, повторим его здесь. Этот алгоритм — фактически алгоритм бинарного поиска, или алгоритм поиска заданного значения в бинарном дереве поиска. Изначально мы стоим в корне дерева. Стоя в текущей вершине, мы сравниваем нашу дробь с дробью в текущей вершине. Если они совпадают, то процесс останавливается — мы нашли дробь в дереве. Иначе, если наша дробь меньше дроби в текущей вершине, то переходим в левого сына, иначе — в правого.

Как было доказано в свойстве о том, что дерево Штерна-Бреко содержит все неотрицательные дроби, при поиске дроби $\frac{x}{y}$ алгоритм совершил не более $x + y$ итераций.

Приведём реализацию, которая возвращает путь до вершины, содержащей заданную дробь $\frac{x}{y}$, возвращая его в виде последовательности символов 'L'/'R': если текущий символ равен 'L', то это обозначает переход в дереве в левого сына, а иначе — в правого (изначально мы стоим в корне дерева, т.е. в вершине с дробью $\frac{1}{1}$). На самом деле, такая последовательность символов, существующая и однозначно определяющая любую неотрицательную дробь, называется **системой счисления Штерна-Бреко**.

```
string find (int x, int y, int a = 0, int b = 1, int c = 1, int d = 0) {
    int m = a+c, n = b+d;
    if (x == m && y == n)
        return "";
    if (x * n < y * m)
        return 'L' + find (x, y, a, b, m, n);
    else
        return 'R' + find (x, y, m, n, c, d);
}
```

Иrrациональным числам в системе счисления Штерна-Бреко будут соответствовать бесконечные последовательности символов; если известна какая-то наперёд заданная точность, то можно ограничиться некоторым префиксом этой бесконечной последовательности. В процессе этого бесконечного поиска иррациональной дроби в дереве Штерна-Бреко алгоритм будет каждый раз находить простую дробь (с постепенно возрастающими знаменателями), обеспечивающую лучшее приближение этого иррационального числа (это применение как раз важно в часовой технике, и в связи с этим Ахилл Бреко и открыл это дерево).

Последовательность Фарея

Последовательностью Фарея порядка n называется множество всех несократимых дробей между 0 и 1, знаменатели которых не превосходят n , причём дроби упорядочены в порядке возрастания.

Эта последовательность названа в честь английского геолога Джона Фарея (John Farey), который попытался в 1816 г. доказать, что в ряде Фарея любая дробь является медиантой двух соседних. Насколько известно, его доказательство было неверным, а правильное доказательство предложил несколько позже Коши (Cauchy). Впрочем, ещё в 1802 г. математик Харос (Haros) в одной из своих работ пришёл практически к тем же результатам.

Последовательности Фарея обладают и множеством собственных интересных свойств, однако наиболее очевидна их **связь с деревом Штерна-Бреко**: фактически, последовательность Фарея получается удалением некоторых ветвей из дерева. Или можно говорить, что для получения последовательности Фарея нужно взять множество дробей, получаемое при построении дерева Штерна-Бреко на бесконечной итерации, и оставить в этом множестве только дроби со знаменателями, не превосходящими n и числителями, не превосходящими знаменатели.

Из алгоритма построения дерева Штерна-Бреко следует и аналогичный **алгоритм** для последовательностей Фарея. На нулевой итерации включим в множество только дроби $\frac{0}{1}$ и $\frac{1}{1}$. На каждой следующей итерации мы между каждыми двумя соседними дробями вставляем их медианту, если её знаменатель не превосходит n . Рано или поздно в множестве перестанут происходить какие-либо изменения, и процесс можно останавливать — мы нашли искомую последовательность Фарея.

Вычислим **длину** последовательности Фарея. Последовательность Фарея порядка n содержит все элементы последовательности Фарея порядка $n - 1$, а также все несократимые дроби со знаменателями, равными n , но это количество, как известно, равно $\phi(n)$. Таким образом, длина L_n последовательности Фарея порядка n выражается по формуле:

$$L_n = L_{n-1} + \phi(n)$$

или, раскрывая рекурсию:

$$L_n = 1 + \sum_{k=1}^n \phi(k)$$

Литература

- Роналд Грэхем, Дональд Кнут, Орен Паташник. **Конкретная математика. Основание информатики** [1998]