Авторы: Маткаримов Б.Т., Арзымбетов Р.А., Игликов А., Шаяхметов А.М.

ГЛАВА 1. .....ПЕРЕБОР С ВОЗВРАТОМ (БАКТРАКИНГ).

Задача 1. Ферзи [http://acm.sgu.ru].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайт**

Даны два числа: n и k ($1 \le n \le 10$, $0 \le k \le n^2$). Требуется посчитать, сколькими способами можно расставить k ферзей на доске n x n?

Входные данные:

На вход программе подаются два числа: n и k.

Выходные данные:

Необходимо вывести одно число — количество расстановок.

Пример:

input.txt	output.txt
3 2	8
4 4	2

Комментарий.

Основная идея решения — перебор с возвратом. Он заключается в следующем: ставим ферзя в клетку (i, j) (которую перебираем) и пытаемся поставить остальных ферзей в клетки (x, y), где (x > i) || ((x == i) && (y > j)). При этом не следует ставить ферзей в клетки, уже находящиеся под ударом. Также, для ускорения работы следует сделать следующее отсечение: первого ферзя ставить только в клетки (i, j), где ($j \le n / 2$) (остальные варианты расстановки могут быть получены из найденных путем отражения).

Программа.

```
#include <stdio.h>

int n, k;
int pos[10];  // pos[i] содержит номер столбца, в который помещен ферзь, находящийся
              //на i-й строке
int used[10]; // used[i] показывает, занята ли i-я строка

// Возвращает количество расстановок count ферзей на рядах с номером >= row
int getcount(int row, int count) {
    int res = 0;
    for (int i = row; i <= n - count; i++) {
        for (int j = 0; j < n; j++)
            if (used[j] == 0) {
                int k;
                    // Проверить, не пробивается ли позиция (i, j) по диагонали
                for (k = 0; k < i; k++)
                    if ((pos[k] != -1) && ((pos[k] - j == i - k) ||
                           ( j - pos[k] == i - k))) break;
                if (k < i) continue;
             // Поставить ферзя в позицию (i, j)
                pos[i] = j;
                used[j] = 1;
             // Если текущий ферзь последний, то посчитать найденную расстановку
                if (count == 1) res++;
             // в противном случае - попытаться расставить остальных ферзей
                 else res += getcount(i + 1, count - 1);
             // Убрать ферзя из позиции (i, j)
                used[j] = 0;
            }
        pos[i] = -1;
    }
    return res;
}

int main() {
```

```
// Ввод и отсечение тривиальных случаев
    scanf("%d %d", &n, &k);
    if (k > n) {
        printf("0\n");
        return 0;
    }
    int count = 0;
    if (k == 0) count = 1;
    else if (k == 1) count = n * n;
    else if (k > 1) {
        int n2 = (n >> 1);
        bool odd = ((n % 2) == 1);
        for (int i = 0; i < n; i++) pos[i] = -1;
        for (int i = 0; i <= n - k; i++) {
            int curr_count = 0;
            for (int j = 0; j < n2; j++) {
             // Поставить первого ферзя в позицию (i, j) и посчитать
             // количество перестановок для такого случая
                pos[i] = j;
                used[j] = 1;
                curr_count += getcount(i + 1, k - 1);
             // Убрать ферзя из позиции (i, j)
                used[j] = 0;
            }
            curr_count += curr_count;
            if (odd) {
                pos[i] = j;
                used[j] = 1;
                curr_count += getcount(i + 1, k - 1);
                used[j] = 0;
            }
            pos[i] = -1;
            count += curr_count;
        }
    }
       // Вывод
    printf("%d\n", count);
    return 0;
}
```

Задача 2. Shtirlits [http://acm.sgu.ru].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**0.75 секунды**
Ограничение по памяти	**0.6 мегабайт**

There is a checkered field of size N x N cells ($1 \le N \le 3$). Each cell designates the territory of a state (i.e. N^2 states). Each state has an army. Let A[i, j] be the number of soldiers in the state which is located on i-th line and on j-th column of the checkered field ($1 \le i \le N$, $1 \le j \le N$, $0 \le A[i, j] \le 9$). For each state the number of neighbors, B[i, j], that have a larger army, is known. The states are neighbors if they have a common border (i.e. $0 \le B[i, j] \le 4$). Shtirlits knows matrix B. He has to determine the number of armies for all states (i.e. to find matrix A) using this information for placing forces before the war. If there are more than one solution you may output any of them.

Входные данные:

The first line contains a natural number N. Following N lines contain the description of matrix B - N numbers in each line delimited by spaces.

Выходные данные:

If a solution exists, the output file should contain N lines, which describe matrix A. Each line will contain N numbers delimited by spaces. If there is no solution, the file should contain NO SOLUTION.

Пример:

input.txt	output.txt
3 1 2 1 1 2 1 1 1 0	1 2 3 1 4 5 1 6 7

Комментарий.

Основная идея решения — перебор с возвратом. При этом нужно хранить и обновлять дополнительную матрицу, элемент (i, j) которой равен количеству ячеек (k, l), соседних с ячейкой (i, j), таких, что A[k, l] > A[i, j] (очевидно, что вначале он будет равен B[i, j]).

Программа.

```
#include <stdio.h>

int n, a[3][3], b[3][3], c[3][3];
// c[i][j] хранит количество элементов a[k][l], соседних с a[i][j] таких,
// что a[k][l] > a[i][j]

// Возвращает true, если удалось найти расстановку, начиная с ячейки (y, x) и
// false в противном случае
bool test(int y, int x) {
    int x2, y2;
// Определение следующей ячейки
    if (x < n - 1) {
        x2 = x + 1;
        y2 = y;
    } else {
        x2 = 0;
        y2 = y + 1;
    }
// Перебор значения a[y][x]
    for (int i = 0; i <= 9; i++) {
        a[y][x] = i;
// Обновление матрицы c
        if (x > 0) {
            if (i > a[y][x - 1]) c[y][x - 1]--;
            else if (i < a[y][x - 1]) c[y][x]--;
        }
        if (y > 0) {
            if (i > a[y - 1][x]) c[y - 1][x]--;
            else if (i < a[y - 1][x]) c[y][x]--;
        }
// Проверить, не нарушает ли поставленное число правила
        if (!(((x > 0) && (c[y][x - 1] < 0)) ||
            ((y > 0) && (c[y - 1][x] != 0)) ||
            (c[y][x] < 0) ||
            ((y == n - 1) &&
            (((x > 0) && (c[y][x - 1] != 0)) ||
            ((x == n - 1) && (c[y][x] != 0)))))) {
// и, если текущая ячейка последняя, то вернуть true
                if ((x == n - 1) && (y == n - 1)) return true;
// иначе - продолжить расстановку со следующей ячейки
                else if (test(y2, x2)) return true;
            }
// Восстановление матрицы c
        if (x > 0) {
            if (i > a[y][x - 1]) c[y][x - 1]++;
            else if (i < a[y][x - 1]) c[y][x]++;
        }
        if (y > 0) {
            if (i > a[y - 1][x]) c[y - 1][x]++;
            else if (i < a[y - 1][x]) c[y][x]++;
        }
    }
    a[y][x] = 0;
    return false;
}

int main() {
    int i, j;
// Ввод и инициализация дополнительной матрицы c
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        scanf("%d", &b[i][j]);
        c[i][j] = b[i][j];
```

```
        a[i][j] = 0;
    }
// Обработка тривиального случая
    if (n == 1) {
        if (b[0][0] == 0) printf("0\n");
        else printf("NO SOLUTION\n");
        return 0;
    }
// Основной перебор
    bool found = false;
    for (i = 0; (i <= 9) && !found; i++) {
        a[0][0] = i;
        found = test(0, 1);
    }
// Вывод результата
    if (found) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) printf("%d ", a[i][j]);
            putchar('\n');
        }
    } else printf("NO SOLUTION\n");
    return 0;
}
```

Задача 3. Sudoku [SEERC 2005].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Sudoku is a very simple task. A square table with 9 rows and 9 columns is divided to 9 smaller squares 3x3 as shown on the Figure. In some of the cells are written decimal digits from 1 to 9. The other cells are empty. The goal is to fill the empty cells with decimal digits from 1 to 9, one digit per cell, in such way that in each row, in each column and in each marked 3x3 subsquare, all the digits from 1 to 9 to appear. Write a program to solve a given Sudoku-task.

Входные данные:

The input data will start with the number of the test cases. For each test case, 9 lines follow, corresponding to the rows of the table. On each line a string of exactly 9 decimal digits is given, corresponding to the cells in this line. If a cell is empty it is represented by 0.

Выходные данные:

For each test case your program should print the solution in the same format as the input data. The empty cells have to be filled according to the rules. If solutions is not unique, then the program may print any one of them.

Пример:

input.txt	output.txt
1 103000509 002109400 000704000 300502006 060000050 700803004 000401000 009205800 804000107	143628579 572139468 986754231 391542786 468917352 725863914 237481695 619275843 854396127

Комментарий.
Основная идея решения — перебор с возвратом.

Программа.

```
#include <stdio.h>

char s[10][20]; // Хранит исходные данные и результат
// uh[i][j] показывает, использована ли цифра j в i-й строке
```

```
bool uh[10][10],
// uv[i][j] показывает, использована ли цифра j в i-м столбце
     uv[10][10],
// us[i][j][k] показывает, использована ли цифра k в квадрате 3x3 с координатами (i, j)
     us[10][10][10];
// Список пустых ячеек
struct {
    int i, j;
} e[1000];
// Количество пустых ячеек
int m;

// Возвращает true, если удалось расставить числа, начиная с k-й пустой ячейки
bool solve(int k) {
    int i, j, a, b, t;
// Если больше нет пустых ячеек, то у нас получилось
    if (k == m) return true;
// Координаты ячейки
    i = e[k].i; j = e[k].j;
// Координаты квадрата 3x3
    a = i / 3; b = j / 3;
// Перебор значений
    for (t = 1; t <= 9; ++t) if (!uh[i][t] && !uv[j][t] && !us[a][b][t]) {
// Ставим цифру t в ячейку (i, j)
        s[i][j] = '0' + t;
        uh[i][t] = uv[j][t] = us[a][b][t] = true;
// Пытаемся заполнить остальные ячейки
        if (solve(k + 1)) return true;
// Убираем цифру t из ячейки (i, j)
        uh[i][t] = uv[j][t] = us[a][b][t] = false;
        s[i][j] = '0';
    }
    return false;
}

int main() {
    int T, i, j, k, l, a, b;
    bool u[10];

    gets(s[0]);
    sscanf(s[0], "%d", &T);
    while (T--) {
// Ввод и инициализация
        for (i = 0; i < 9; ++i) gets(s[i]);
        for (i = 0; i < 9; ++i) for (j = 1; j <= 9; ++j)
                uv[i][j] = uh[i][j] = false;
        for (i = 0; i < 3; ++i) for (j = 0; j < 3; ++j) {
            for (k = 1; k <= 9; ++k) us[i][j][k] = false;
            for (int a = i * 3; a < i * 3 + 3; ++a)
            for (int b = j * 3; b < j * 3 + 3; ++b)
                us[i][j][s[a][b] - '0'] = true;
        }
        for (i = 0; i < 9; ++i) for (j = 0; j < 9; ++j)
            uv[j][s[i][j] - '0'] = uh[i][s[i][j] - '0'] = true;
        m = 0;
        for (i = 0; i < 9; ++i) for (j = 0; j < 9; ++j) if (s[i][j] == '0') {
            e[m].i = i; e[m].j = j; ++m;
        }
// Основной перебор
        if (m) solve(0);
// Вывод результата
        for (i = 0; i < 9; ++i) puts(s[i]);
    }
    return 0;
}
```

Задача 4. Electrical Outlets [NCPC 2005].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Roy's apartment has only one single wall outlet, so Roy can only power one of his electrical appliances at a time. However, that apartment had many more wall outlets, so he is not sure whether his power strips will provide him with enough outlets now.

Your task is to help Roy compute how many appliances he can provide with electricity, given a set of power strips. Note that without any power strips, Roy can power one single appliance through the wall outlet. Also, remember that a power strip has to be powered itself to be of any use.

Входные данные:

Input will start with a single integer $1 \le N \le 20$, indicating the number of test cases to follow. Then follow N lines, each describing a test case. Each test case starts with an integer $1 \le K \le 10$, indicating the number of power strips in the test case. Then follow, on the same line, K integers separated by single spaces, $O1\ O2 \ldots OK$, where $2 \le Oi \le 10$, indicating the number of outlets in each power strip.

Выходные данные:

Output one line per test case, with the maximum number of appliances that can be powered.

Пример:

input.txt	output.txt
3 3 2 3 4 10 4 4 4 4 4 4 4 4 4 4 4 10 10 10 10	7 31 37

Комментарий.

Разобьем power strips на два множества S и P. Power strips из множества P будут питать только подключаемые appliances. Power strips из множества S могут питать как appliances, так и power strips из любого множества. При этом на питание самого множества S будет потрачено |S| - 1 розеток (одна дается сразу). Все power strips из множества P будут работать только, если $\Sigma(S) - (|S| - 1) \ge |P|$ (где $\Sigma(S) = O_{i1} + O_{i2} + \ldots O_{i|S|}$ и i_j принадлежит S). Тогда на appliances остается $f = \Sigma(S) - (|S| - 1) - |P|$ розеток. Теперь осуществляем перебор по всем разбиениям power strips на два множества (а их будет 2^k) и выбираем максимум по f. Временная сложность решения — $O(N \cdot 2^k \cdot K)$, что вполне нормально при заданных ограничениях.

Программа.

```
#include <stdio.h>

int main() {
    int T;
    scanf("%d", &T);
    while (T--) {
// Ввод
        int n, i, j, a[20];
        scanf("%d", &n);
        for (i = 0; i < n; ++i) scanf("%d", &a[i]);
// Перебор разбиений
        int max = 0;
        for (i = 0; i < 1 << n; ++i) {
            int c = 0, k = 0, p = 0, s = 0;
// Подсчет сумм и количество элементов в множествах
            for (j = 0; j < n; ++j)
// Если j в множестве P
                if (i & (1 << j)) {
                    p += a[j]; ++c;
// Если j в множестве S
                } else {
                    s += a[j]; ++k;
                }
// Можем ли мы запитать все power strips?
            s -= k - 1;
            if (s < c) continue;
```

```
// Подсчет количества оставшихся розеток
            p += s - c;
            if (p > max) max = p;
        }
// Вывод результата
        printf("%d\n", max);
    }
    return 0;
}
```

Задача 5. Firefighters [SEERC 2003].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

You are given an expression, containing the integer numbers between 1 and 999, simple mathematical operators (+, -, *, /), brackets, and question marks (?), representing the lost mathematical operators. For every expression given, your only task is to state if an expression can or cannot give the required result. In order to help you, the mathematician has chosen only expressions that have the following restrictions:

1. The expressions contain no more than 100 symbols;
2. The brackets enclose no more than 1 operator with his two operands. However, every one of these operands can be an expression in brackets;
3. The constants in the expressions have no sign, i.e. there are no negative numbers in expressions;
4. The maximum number of question marks in the expressions (the lost operators) is less or equal to 10.

The calculation should be performed using the following rules:

1. The operators * and / are of higher priority than the operators + and -. Parentheses may change the priorities as usually;
2. The operators +, -, *, and / are left associative, meaning that they group from left to right. If a, b and c are numbers: a*b*c = (a*b)*c, a/b/c = (a/b)/c, a/b*c = (a/b)*c, a+b+c = (a+b)+c, a-b+c = (a-b)+c, etc.

When dividing two integers, you should ignore the decimal fraction, for example consider the following equations: 2/5=0, 9/5=1, 100/6=16.

Входные данные:

The first line of the input contains an integer N – determining the number of equations. Next 2*N lines contain the equations. One equation is defined in two lines. The first line is the expression, defining the left side of the equation; second line is an integer result, defining the right side of the equation.

Выходные данные:

For every equation in the input file, write yes or no on separate lines on the standard output.

Пример:

input.txt	output.txt
3 1?((2*(3*4))+(5+6)) 35 1?2*3+4-14 0 1?3*4/5*6+12 11	yes no no

Комментарий.

Решение заключается в полном переборе всех возможных подстановок операций вместо вопросов (их будет не больше 4^{10}). При получении некоторого выражения его значение можно легко подсчитать, например, переведя в постфиксную запись.

Программа.

```
#include <stdio.h>
#include <ctype.h>

// Строка выражения
char s[1000];
// Выражение в постфиксной форме
```

```
struct {
    bool arg;
    int v;
} e[1000];
// Количество элементов в выражении
int ec;
// Стек для перевода и вычисления выражения
int st[1000];

// Возвращает приоритет операции
int prior(int op) {
    if (op == -1) return 0;
    if (op < 2) return 1;
    return 2;
}

// Переводит выражение s в постфиксную нотацию (алгоритм Дейкстры)
void parse(char* s) {
    int i, h, op;
    h = ec = i = 0;
    while (s[i] != 0) {
        if (isdigit(s[i])) {
            e[ec].arg = true;
            e[ec].v = 0;
            while ((s[i] != 0) && isdigit(s[i])) e[ec].v = e[ec].v * 10 + s[i++] - '0';
            ec++;
            continue;
        } else {
            if (s[i] == '(') st[h++] = -1;
            else if (s[i] == ')') {
                while (st[h - 1] != -1) {
                    e[ec].arg = false;
                    e[ec++].v = st[--h];
                }
                h--;
            } else {
                if (s[i] == '+') op = 0;
                else if (s[i] == '-') op = 1;
                else if (s[i] == '*') op = 2;
                else if (s[i] == '/') op = 3;
                else {
                    i++;
                    continue;
                }
                while ((h > 0) && (prior(st[h - 1]) >= prior(op))) {
                    e[ec].arg = false;
                    e[ec++].v = st[--h];
                }
                st[h++] = op;
            }
        }
        i++;
    }
    while (h > 0) {
        e[ec].arg = false;
        e[ec++].v = st[--h];
    }
}

// Вычисляет значение выражение и возвращает true, если оно равно n, и false
// в противном случае
bool calc(int n) {
    int h, i, a, b;
    for (h = i = 0; i < ec; i++) {
        if (e[i].arg) st[h++] = e[i].v;
        else {
            b = st[--h];
            a = st[--h];
            switch (e[i].v) {
            case 0:
                st[h++] = a + b;
                break;
            case 1:
```

```
                st[h++] = a - b;
                break;
            case 2:
                st[h++] = a * b;
                break;
            case 3:
                if (b == 0) return false;
                st[h++] = a / b;
                break;
            }
        }
    }
    return st[0] == n;
}

// Возвращает true, если есть значением выражения может быть n
bool test(char* s, int n) {
    int i;
    for (i = 0; s[i] != 0; i++) {
        if (s[i] == '?') {
            s[i] = '+';
            if (test(s, n)) return true;
            s[i] = '-';
            if (test(s, n)) return true;
            s[i] = '*';
            if (test(s, n)) return true;
            s[i] = '/';
            if (test(s, n)) return true;
            s[i] = '?';
            return false;
        }
    }
    parse(s);
    return calc(n);
}

int main() {
    int n, t;

    scanf("%d", &t);
    while (t > 0) {
        scanf("%s %d", s, &n);
        if (test(s, n)) printf("yes\n");
        else printf("no\n");
        t--;
    }
    return 0;
}
```

ГЛАВА 2. СТРУКТУРЫ ДАННЫХ.

Задача 6. Черный ящик [NEERC 1996].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Черный ящик организован наподобие примитивной базы данных. Он может хранить набор целых чисел и имеет выделенную переменную *i*. В начальный момент времени черный ящик пуст, а значение переменной i=0. Черный ящик обрабатывает последовательность поступающих команд (запросов). Существуют два типа запросов:

- ADD (x): положить в черный ящик элемент x;
- GET: увеличить значение *i* на 1 и выдать i-ый минимальный из содержащихся в черном ящике элементов. Напомним, что i-ым минимальным называется число, стоящее на *i*-ом месте после сортировки элементов черного ящика в порядке неубывания.

Требуется разработать эффективный алгоритм, обрабатывающий заданную последовательность запросов. Максимальное допустимое количество запросов ADD и GET — по 30000 каждого типа.

Последовательность запросов будем задавать двумя наборами чисел:

1.Последовательностью включаемых в черный ящик элементов $A(1), A(2),\ldots A(M)$. Значения A — целые числа, не превосходящие по абсолютной величине 2 000 000 000, $1 \le M \le 30000$.

2.Последовательностью $u(1), u(2),\ldots u(N)$, задающей количество содержащихся в черном ящике элементов в момент выполнения первой, второй, ..., N-ой команды GET.

Схема работы черного ящика предполагает, что последовательность натуральных чисел $u(1), u(2),\ldots u(N)$ упорядочена по неубыванию, $N \le M$ и для всех p $(1 \le p \le N)$ выполняется соотношение $p \le u(p) \le M$. Последнее следует из того, что для *p*-го элемента последовательности u мы выполняем запрос GET, выдающий *p*-ое минимальное число из набора $A(1),A(2),...,A(u(p))$.

Входные данные:

Файл исходных данных содержит (в указанном порядке): $M, N, A(1), A(2), ..., A(M), u(1), u(2), ..., u(N)$. Все числа разделяются пробелами и (или) символами перевода строки.

Выходные данные:

Вывести в выходной файл последовательность ответов черного ящика для заданной последовательности запросов. Числа в выходном файле могут разделяться пробелами и символами перевода строки.

Пример:

input.txt	output.txt
7 4 3 1 -4 2 8 -1000 2 1 2 6 6	3 3 1 2

Комментарий.

В кратце условие задачи можно записать так: построить некоторую структуру данных, в которую последовательно добавляются числа. Нам нужно уметь в любой момент времени находить i-ую порядковую статистику, где i меняется от запроса к запросу. Такая задача рассмотрена в [12]. Предлагается следующее решение:

1. Построим сбалансированное бинарное дерево, в которое и будем добавлять числа.
2. Каждый узел дерева будет содержать дополнительное поле, показывающее, сколько узлов находится в поддереве, корнем которого является этот узел.
3. Теперь для нахождения i-й порядковой статистики достаточно произвести один спуск по дереву.

Программа.

```
/* Приведенная функция возвращает i-ю порядковую статистику. Предполагается, что дерево
задано в виде массива структур. Поля l и r струкутры содержат порядковые номера левого и
правого сыновей текущего узла, поле k — значение, хранящееся в этом узле, s — размер
поддерева.*/

int t_os(int i) {
    int r, c;
```

```
    c = ROOT;
    r = t[t[c].l].s + 1;
    while (i != r) {
        if (i < r) c = t[c].l;
        else if (i > r) {
            i -= r;
            c = t[c].r;
        }
        r = t[t[c].l].s + 1;
    }
    return t[c].k;
}
```

Задача 7. Предъявите документы! [ЛКШ 2005].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

При рождении человеку выдается N документов, пронумерованных от 1 до N. Документ с номером i характеризуется своей важностью A_i и стоимостью B_i. В течение жизни человек принимает участие в M важных событиях, для участия в каждом из которых ему требуется отдать один из своих документов. Более того, для участия в событии с номером j необходимо отдать документ с важностью не меньше C_j и не больше D_j. При этом, естественно, человек каждый раз отдает самый дешевый документ из подходящих. Помогите человеку прожить жизнь, поучаствовав во всех важных событиях.

Входные данные:

Во входном файле содержится число N ($1 \le N \le 10^5$), затем N пар чисел A_i и B_i ($1 \le A_i;B_i \le 10^9$), затем число M ($1 \le M \le 10^5$), затем M пар чисел C_j и D_j ($1 \le C_j \le D_j \le 10^9$). Важности всех документов различны.

Выходные данные:

В выходной файл запишите M чисел — номера документов, которые необходимо отдать, j-е число обозначает номер документа, отдаваемый на j-м событии. Числа в строке разделяйте пробелом. В случае если жизнь прожить невозможно, выведите в выходной файл одно слово BOTVA.

Пример:

input.txt	output.txt
35 99 1 100 3 50 22 5 1 4	3 2
35 99 1 100 3 50 22 5 3 4	BOTVA

Комментарий.

В принципе, алгоритм решения задачи очевиден, однако, если его реализовывать напрямую, то оно не уложится в отведенное время. Более правильным будет создать структуру данных, которая будет позволять удалять элементы и в любой момент времени быстро находить миимальный элемент на заданном отрезке. Предположим, что значения A_i и B_i хранятся в структуре p[i] как поля a и b, тогда получаем следующий алгоритм:

1. Отсортировать массив p[i] по неубыванию a.
2. Построить описанную выше структуру данных по полю b.
3. Выполнить требуемые в задаче действия.

Структурой данный, позволяющей реализовать этот алгоритм, может быть, например, дерево отрезков, описанное в «Лекции по информатике» (Павлов).

Дерево отрезко — это полное двоичное дерево, каждый узел которого соответствует некоторому отрезку исходного массива, а именно: корень соответствует всему массиву, его сыновься — левой и правой половинам

массива, …, листья — одиночным элементам. В каждом узле нужно хранить минимальное значение на интервале, которому он соответствует.

Программа.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

const int inf = 2000000000;
// Дерево отрезков
struct {
  int l, r, x;
} a[400000];
// Исходные данные
struct {
  int a, b, i;
} p[400000];
// Результат
int r[400000];
// Количество документов
int n;

// Сортировка p по невозрастанию a
void sort(int l, int r) {
  int i, j, x, t;
  x = p[l + rand() % (r - l + 1)].a;
  i = l; j = r;
  do {
    while (p[i].a < x) i++;
    while (p[j].a > x) j--;
    if (i <= j) {
      t = p[i].a; p[i].a = p[j].a; p[j].a = t;
      t = p[i].b; p[i].b = p[j].b; p[j].b = t;
      t = p[i].i; p[i].i = p[j].i; p[j].i = t;
      i++; j--;
    }
  } while (i <= j);
  if (j > l) sort(l, j);
  if (i < r) sort(i, r);
}

// Инициализация дерева отрезков
int fill(int v, int l, int r) {
  int y;
  a[v].l = l; a[v].r = r;
  if (l == r) a[v].x = l;
  else {
    a[v].x = fill((v << 1) + 1, l, (l + r) >> 1);
    y = fill((v << 1) + 2, ((l + r) >> 1) + 1, r);
    if (p[y].b < p[a[v].x].b) a[v].x = y;
  }
  return a[v].x;
}

// Нахождение левой границы интервала
int findl(int x) {
  int l, r, c;
  l = -1; r = n;
  while (r - l > 1) {
    c = (l + r) >> 1;
    if (c == -1) l = c;
    else if (c == n) r = c;
    else if (p[c].a < x) l = c;
    else r = c;
  }
  return r;
}

// Нахождение правой границы интервала
int findr(int x) {
  int l, r, c;
  l = -1; r = n;
```

```
  while (r - l > 1) {
    c = (l + r) >> 1;
    if (c == -1) l = c;
    else if (c == n) r = c;
    else if (p[c].a <= x) l = c;
    else r = c;
  }
  return l;
}

int min(int a, int b) {
  return a < b ? a : b;
}

int max(int a, int b) {
  return a > b ? a : b;
}

// Нахождение минимального элемента на отрезке (l, r)
int find(int v, int l, int r) {
  int x, y;
  if ((a[v].l == l) && (a[v].r == r)) x = a[v].x;
  else {
  x = y = -1;
    if (a[(v << 1) + 1].r >= l) x = find((v << 1) + 1, l, min(a[(v << 1) + 1].r, r));
    if (a[(v << 1) + 2].l <= r) y = find((v << 1) + 2, max(a[(v << 1) + 2].l, l), r);
    if ((y != -1) && ((x == -1) || (p[y].b < p[x].b))) x = y;
  }
  return x;
}

// Удаление элемента x
void del(int x) {
  int y;
  p[x].b = inf;
  x += n - 1;
  while (x > 0) {
    y = (x - 1) >> 1;
    if (p[a[(y << 1) + 1].x].b < p[a[(y << 1) + 2].x].b) a[y].x = a[(y << 1) + 1].x;
    else a[y].x = a[(y << 1) + 2].x;
    x = y;
  }
}

int main() {
  int m, i, t, c, d;

  freopen("docs.in", "r", stdin);
  freopen("docs.out", "w", stdout);

// Ввод
  srand(time(0));
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    scanf("%d %d", &p[i].a, &p[i].b);
    p[i].i = i;
  }

// Сортировка и построение дерева отрезков
  if (n > 1) sort(0, n - 1);
  t = 0;
  while (1 << t < n) t++;
  for (; n < 1 << t; n++) p[n].a = p[n].b = inf;
  fill(0, 0, n - 1);

// Основной алгоритм
  scanf("%d", &m);
  for (i = t = 0; i < m; i++) {
    scanf("%d %d", &c, &d);
// Нахождение интервала
    c = findl(c); d = findr(d);
// Поиск минимума, если не найден - выход
    if (c <= d) r[t++] = find(0, c, d);
```

```
    if ((c > d) || (p[r[t - 1]].b == inf)) {
      printf("BOTVA\n");
      return 0;
    }
// Удаление документа
    del(r[t - 1]);
  }

// Вывод
  for (i = 0; i < t; i++) printf("%d ", p[r[i]].i + 1);
  return 0;
}
```

Задача 8. Feel Good! [NEERC 2005].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Bill is developing a new mathematical theory for human emotions. His recent investigations are dedicated to studying how good or bad days influent people's memories about some period of life.

A new idea Bill has recently developed assigns a non-negative integer value to each day of human life. Bill calls this value the emotional value of the day. The greater the emotional value is, the better the day was. Bill suggests that the value of some period of human life is proportional to the sum of the emotional values of the days in the given period, multiplied by the smallest emotional value of the day in it. This schema reflects that good on average period can be greatly spoiled by one very bad day.

Now Bill is planning to investigate his own life and find the period of his life that had the greatest value. Help him to do so.

Входные данные:

The first line of the input file contains n — the number of days of Bill's life he is planning to investigate ($1 \le n \le 100000$). The rest of the file contains n integer numbers $a_1, a_2, \ldots a_n$ ranging from 0 to 10^6 — the emotional values of the days. Numbers are separated by spaces and/or line breaks.

Выходные данные:

On the first line of the output file print the greatest value of some period of Bill's life. On the second line print two numbers l and r such that the period from l-th to r-th day of Bill's life (inclusive) has the greatest possible value. If there are multiple periods with the greatest possible value, then print any one of them.

Пример:

input.txt	output.txt
6 3 1 6 4 5 2	60 3 5

Комментарий.

Основная идея решения — перебор по минимальному элементу. То есть, допустим a_i будет минимальным, тогда нужно найти наибольший отрезок для которого это выполняется. Это можно сделать следующим образом:

1. Будем искать сначала левую, а потом правую границы отрезка. Пусть l_i — левая граница отрезка с минимальным элементов a_i, то есть $(i == 0) \,||\, (a_{i-1} < a_i)$.
2. После того как найдем l_i будем добавлять i в стек и поддерживать следующую особенность: для всех j, находящихся в стеке должно выполняться: a[j] < a[i].
3. Если указанное условие выполняется, то перед добавлением i в стек в вершине стека будет находиться элемент, левее искомого l_i на один (т.е. $l_i - 1$), и его нам нужно записать в l_i.
4. Повторяем аналогичные действия для нахождения правых границ, производим перебор по всем i и выбираем такое i, что $sum(l_i, r_i) \cdot a_i$ будет максимальным.

Программа.

```
import java.io.*;
import java.util.*;

public class f_feelgood {
    public static void main(String args[]) throws IOException {

// Ввод
        BufferedReader in = new BufferedReader(new FileReader("feelgood.in"));
```

```
        StreamTokenizer st = new StreamTokenizer(in);
        st.resetSyntax();
        st.wordChars('0', '9');
        st.wordChars('-', '-');
        st.wordChars('+', '+');
        while (st.nextToken() != StreamTokenizer.TT_WORD);
        int n = Integer.parseInt(st.sval);
        int a[] = new int[n];
        for (int i = 0; i < n; ++i) {
            while (st.nextToken() != StreamTokenizer.TT_WORD);
            a[i] = Integer.parseInt(st.sval);
        }
        in.close();

        int stack[] = new int[n];

// Нахождение левых границ
        int l[] = new int[n];
        int h = 0;
        stack[h++] = 0;
        l[0] = 0;
        for (int i = 1; i < n; ++i) {
            while ((h > 0) && (a[stack[h - 1]] >= a[i])) --h;
            if (h > 0) l[i] = stack[h - 1] + 1;
            else l[i] = 0;
            stack[h++] = i;
        }

// Нахождение правых границ
        int r[] = new int[n];
        h = 0;
        stack[h++] = n - 1;
        r[n - 1] = n - 1;
        for (int i = n - 2; i >= 0; --i) {
            while ((h > 0) && (a[stack[h - 1]] >= a[i])) --h;
            if (h > 0) r[i] = stack[h - 1] - 1;
            else r[i] = n - 1;
            stack[h++] = i;
        }

// Вычисление сумм (s[i] = a[0] + a[1] + .. + a[i])
        long s[] = new long[n];
        s[0] = a[0];
        for (int i = 1; i < n; ++i) s[i] = s[i - 1] + a[i];

// Нахождение максимума
        int j = -1;
        long max = -1;
        for (int i = 0; i < n; ++i) {
            long ts = (s[r[i]] - (l[i] > 0 ? s[l[i] - 1] : 0)) * a[i];
            if (ts > max) {
                max = ts;
                j = i;
            }
        }
        ++l[j]; ++r[j];

// Вывод
        PrintWriter out = new PrintWriter(new FileWriter("feelgood.out"));
        out.println(max);
        out.println(l[j] + " " + r[j]);
        out.close();
    }
}
```

Задача 9. Humble numbers [http://acm.uva.es].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

A number whose only prime factors are 2,3,5 or 7 is called a humble number. The sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, ... shows the first 20 humble numbers.

Write a program to find and print the n-th element in this sequence.

Входные данные:

The input consists of one or more test cases. Each test case consists of one integer n with ($1 \le n \le 5842$). Input is terminated by a value of zero for n.

Выходные данные:

For each test case, print one line saying "The nth humble number is *number*.". Depending on the value of n, the correct suffix "st", "nd", "rd", or "th" for the ordinal number nth has to be used like it is shown in the sample output.

Пример:

input.txt	output.txt
1	The 1st humble number is 1.
2	The 2nd humble number is 2.
3	The 3rd humble number is 3.
4	The 4th humble number is 4.
11	The 11th humble number is 12.
12	The 12th humble number is 14.
13	The 13th humble number is 15.
21	The 21st humble number is 28.
22	The 22nd humble number is 30.
23	The 23rd humble number is 32.
100	The 100th humble number is 450.
1000	The 1000th humble number is 385875.
5842	The 5842nd humble number is 2000000000.
0	

Комментарий.

Основная идея генерация всех humble numbers. Это можно сделать, используя структуру данных — кучу (описание в «Алгоритмы: Построение и анализ» (Корме, …)). Изначально в куче содержится один элемент — 1. 5842 раза будем выполнять следующее:

1. взять из кучи минимальный элемент (x), не равный предыдущему и запомнить его;
2. добавить в кучу элементы 2x, 3x, 5x, 7x.

Массив из найденных элементов x и будет являться списком humble numbers.

Следует отметить, что данное решение можно значительно оптимизировать по памяти, немного усложнив структуру данных и правила добавления.

Программа.

```
#include <stdio.h>
#include <string.h>

int h[5000]; // куча
int a[10000]; // humble numbers
int n;

// Проталкивание элемента в кучу
void h_fix(int i) {
    int x, c;
    x = h[i];
    while (i < n >> 1) {
        c = i << 1;
        if ((c < n - 1) && (h[c + 1] < h[c])) c++;
        if (x <= h[c]) break;
        h[i] = h[c];
        i = c;
```

```
    }
    h[i] = x;
}

// Извлечение минимального элемента из кучи
int h_extrmin() {
    int t;
    t = h[0];
    h[0] = h[n - 1];
    h[n - 1] = t;
    n--;
    h_fix(0);
    return h[n];
}

// Добавление элемента в кучу
void h_ins(int x) {
    int i;
    i = n++;
    while ((i > 0) && (h[i >> 1] > x)) {
        h[i] = h[i >> 1];
        i = i >> 1;
    }
    h[i] = x;
}

int main() {
    int i, c, p;
    char s[10];

// Генерация humble numbers
    h[0] = 1; n = 1; c = 0; p = -1;
    for (i = 0; i < 5842; i++) {
        a[c] = h_extrmin();
        while (a[c] <= p) a[c] = h_extrmin();
        if (2147483647 / 2 >= a[c]) h_ins(a[c] * 2);
        if (2147483647 / 3 >= a[c]) h_ins(a[c] * 3);
        if (2147483647 / 5 >= a[c]) h_ins(a[c] * 5);
        if (2147483647 / 7 >= a[c]) h_ins(a[c] * 7);
        p = a[c++];
    }

// Ввод и вывод
    scanf("%d", &n);
    while (n != 0) {
        if ((n % 100 > 10) && (n % 100 < 20)) strcpy(s, "th");
        else if (n % 10 == 1) strcpy(s, "st");
        else if (n % 10 == 2) strcpy(s, "nd");
        else if (n % 10 == 3) strcpy(s, "rd");
        else strcpy(s, "th");
        printf("The %d%s humble number is %d.\n", n, s, a[n - 1]);
        scanf("%d", &n);
    }
    return 0;
}
```

Задача 10. Mobile phones [IOI 2001].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Предположим, что в регионе Тампере базовые станции обеспечения мобильной телефонной связи четвертого поколения действуют следующим образом. Регион поделен на квадраты. Квадраты образуют матрицу размера $S \times S$, строки и столбцы которой пронумерованы от 0 до S-1. В каждом квадрате находится базовая станция. Количество работающих мобильных телефонов внутри квадрата может меняться, так как телефоны могут перемещаться из одного квадрата в другой, и телефоны могут включаться или выключаться. В некоторые моменты времени каждая базовая станция передает головной базовой станции отчет об изменении количества работающих телефонов и свои координаты (номер строки и номер столбца соответственно).

Напишите программу, которая получает эти отчеты и отвечает на запросы о текущем общем количестве работающих мобильных телефонов в некоторой прямоугольной области.

Входные данные:

Целочисленные данные вводятся из стандартного входного потока. Входные данные кодируются следующим образом. Каждая строка содержит одну команду. Команда состоит из кода и набора параметров (целых чисел) в соответствии со следующей таблицей.

Команда	Параметры	Значение
0	S	Инициализирует матрицу размера S×S нулями. Эта команда выдается только один раз и должна быть первой командой.
1	X Y A	Прибавляет к количеству работающих мобильных телефонов в квадрате (X, Y) число A. Число A может быть как положительным, так и отрицательным.
2	L B R T	Запрашивает текущее суммарное количество работающих мобильных телефонов в квадратах (X,Y), где $L \le X \le R$, $B \le Y \le T$
3		Завершает программу. Эта команда выдается только один раз и должна быть последней.

Программа отвечает на каждый запрос (команду с кодом 2), записывая целые числа в стандартный выходной поток.

Значения всегда будут в допустимых пределах, так что нет необходимости их проверять. В частности, добавление отрицательного числа A не приведет к уменьшению количества телефонов в квадрате до значения, меньшего нуля. Индексы в матрице начинаются от 0, например, для матрицы 4×4, мы имеем $0 \le X \le 3$ и $0 \le Y \le 3$.

Выходные данные:

Для каждой команды с номером 2 выведите необходимое количество на отдельной строке.

Пример:

input.txt	output.txt
0 4 1 1 2 3 2 0 0 2 2 1 1 1 2 1 1 2 -1 2 1 1 2 3 3	3 4

Комментарий.

Решение данной задачи основывается на структуре данных — дерево Фенвика. Оно позволяет за логарифмическое время находить сумму на некотором отрезке массива, производя изменения в нем также за логарифмическое время. Более подробно оно описано в «Лекции по информатике» (Павлов). Здесь используется более сложная структура — двумерное дерево Фенвика.

Программа.

```
#include <stdio.h>

int n;
int a[1025][1025]; // Двумерное дерево Фенвика

// Следующий элемент
int next(int x) {
    return (x << 1) - (x & (x - 1));
}

// Предыдущий элемент
int prev(int x) {
    return x & (x - 1);
}

// Сумма в квадрате (l, b, r, t)
__int64 sum(int l, int b, int r, int t) {
    int i, j;
    __int64 s;
```

```
    s = 0;
    for (i = t; i > 0; i = prev(i))
        for (j = r; j > 0; j = prev(j)) s += a[i][j];
    for (i = t; i > 0; i = prev(i))
        for (j = l - 1; j > 0; j = prev(j)) s -= a[i][j];
    for (i = b - 1; i > 0; i = prev(i))
        for (j = r; j > 0; j = prev(j)) s -= a[i][j];
    for (i = b - 1; i > 0; i = prev(i))
        for (j = l - 1; j > 0; j = prev(j)) s += a[i][j];
    return s;
}

// Добавление b к элементу (x, y)
void add(int x, int y, int b) {
    int i, j;
    for (i = y; i <= n; i = next(i)) for (j = x; j <= n; j = next(j)) a[i][j] += b;
}

int main() {
    int c, i, j, b, x, y, l, r, t, s;

    scanf("%d", &c);
    while (c != 3) {
        if (c == 0) {
            scanf("%d", &n);
            for (i = 1; i <= n; i++) for (j = 1; j <= n; j++) a[i][j] = 0;
        } else if (c == 1) {
            scanf("%d %d %d", &x, &y, &b);
            add(x + 1, y + 1, b);
        } else {
            scanf("%d %d %d %d", &l, &b, &r, &t);
            printf("%I64d\n", sum(l + 1, b + 1, r + 1, t + 1));
        }
        scanf("%d", &c);
    }
    return 0;
}
```

Задача 11. Manager [SEERC 2002].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

One of the programming paradigm in parallel processing is the producer/consumer paradigm that can be implemented using a system with a "manager" process and several "client" processes. The clients can be producers, consumers, etc. The manager keeps a trace of client processes. Each process is identified by its cost that is a strictly positive integer in the range 1 .. 10000. The number of processes with the same cost cannot exceed 10000. The queue is managed according to three types of requests, as follows:

- a x - add to the queue the process with the cost x;
- r - remove a process, if possible, from the queue according to the current manager policy;
- p i - enforce the policy i of the manager, where i is 1 or 2. The default manager policy is 1
- e - ends the list of requests.

There are two manager policies:

- 1 - remove the minimum cost process
- 2 - remove the maximum cost process

The manager will print the cost of a removed process only if the ordinal number of the removed process is in the removal list.

Your job is to write a program that simulates the manager process.

Входные данные:

Each data set in the input has the following format:

- the maximum cost of the processes
- the length of the removal list
- the removal list - the list of ordinal numbers of the removed processes that will be displayed; for example 1 4 means that the cost of the first and fourth removed processes will be displayed
- the list of requests each on a separate line.

Each data set ends with an e request. The data sets are separated by empty lines.

Выходные данные:

The program prints the cost of each process that is removed, provided that the ordinal number of the remove request is in the list and the queue is not empty at that moment. If the queue is empty the program prints -1. The results are printed on separate lines. An empty line separates the results of different data sets.

Пример:

input.txt	output.txt
5 2 1 3 a 2 a 3 r a 4 p 2 r a 5 r e	2 5

Комментарий.

Данную задачу можно решить несколькими способами: используя сбалансированное двоичное дерево, две кучи или слоеные списки (о слоеных списках см. algolist.manual.ru). Здесь приводится решение на основе декартова дерева.

Программа.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

bool rl[10000000];
struct {
  int l, r, p, k, a;
} t[2000000];
int th, tc;

// Инициализация дерева
void t_init(int a) {
  t[0].k = a; t[0].a = rand(); t[0].r = t[0].l = t[0].p = -1;
  tc = 1; th = 0;
}

// Левый поворот в вершине x
void t_lrot(int x) {
  int y;
  y = t[x].r;
  t[x].r = t[y].l;
  if (t[x].r != -1) t[t[x].r].p = x;
  t[y].p = t[x].p;
  if (x == th) th = y;
  else if (x == t[t[x].p].l) t[t[x].p].l = y;
  else t[t[x].p].r = y;
  t[x].p = y;
  t[y].l = x;
}

// Правый поворот в вершине x
void t_rrot(int x) {
  int y;
  y = t[x].l;
  t[x].l = t[y].r;
  if (t[x].l != -1) t[t[x].l].p = x;
  t[y].p = t[x].p;
  if (x == th) th = y;
  else if (x == t[t[x].p].l) t[t[x].p].l = y;
```

```
  else t[t[x].p].r = y;
  t[x].p = y;
  t[y].r = x;
}

// Исправление дерева после вставки
void t_fix(int x) {
  while ((t[x].p != -1) && (t[t[x].p].a > t[x].a))
    if (x == t[t[x].p].l) t_rrot(t[x].p);
    else t_lrot(t[x].p);
}

// Вставка элемента
void t_ins(int a) {
  int p, i;
  if (th == -1) t_init(a);
  else {
    p = -1; i = th;
    while (i != -1) {
      p = i;
      if (t[i].k >= a) i = t[i].l;
      else i = t[i].r;
    }
    if (a <= t[p].k) t[p].l = tc; else t[p].r = tc;
    t[tc].p = p; t[tc].l = t[tc].r = -1;
    t[tc].k = a; t[tc].a = rand();
    t_fix(tc++);
  }
}

void t_del(int x) {
  t[x].a = 1000000;
  while ((t[x].l != -1) || (t[x].r != -1))
    if (t[x].l == -1) t_lrot(x);
    else if (t[x].r == -1) t_rrot(x);
    else if (t[t[x].l].a < t[t[x].r].a) t_rrot(x);
    else t_lrot(x);
  if (x == th) th = -1;
  else if (x == t[t[x].p].l) t[t[x].p].l = -1;
  else t[t[x].p].r = -1;
}

// Возвращает минимальный элемент
int t_min() {
  int i;
  if (th == -1) return -1;
  i = th;
  while (t[i].l != -1) i = t[i].l;
  return i;
}

// Возвращает максимальный элемент
int t_max() {
  int i;
  if (th == -1) return -1;
  i = th;
  while (t[i].r != -1) i = t[i].r;
  return i;
}

int main() {
  int mc, rlc, i, p, a, k;
  char s[1000];

  srand(time(0));

  while (scanf("%d %d", &mc, &rlc) == 2) {
    for (i = 0; i < 10000000; i++) rl[i] = false;
    for (i = 0; i < rlc; i++) {
      scanf("%d", &a);
      if (a < 10000000) rl[a - 1] = true;
    }
    i = 0;
```

```
    th = -1;
    scanf("%s", s);
    p = 1;
    while (s[0] != 'e') {
      if (s[0] == 'a') {
        scanf("%d", &a);
        t_ins(a);
      } else if (s[0] == 'r') {
        if (p == 1) {
          k = t_min();
          t_del(k);
          if (rl[i])
            if (k == -1) printf("-1\n");
            else printf("%d\n", t[k].k);
        } else {
          k = t_max();
          t_del(k);
          if (rl[i])
            if (k == -1) printf("-1\n");
            else printf("%d\n", t[k].k);
        }
        i++;
      } else if (s[0] == 'p') scanf("%d", &p);
      scanf("%s", s);
    }
    putchar('\n');
  }

  return 0;
}
```

Задача 12. Железная дорога [ЛКШ 2004].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Государственные Байтландские железные дороги решили идти в ногу со временем и решили ввести новейшую связь InterCITY. Из-за недостатка эффективных двигателей, чистых и прямых дорог возможно было установить только одну такую связь. Еще одним препятствием был недостаток современных компьютерных систем для брони мест. Ваша задача составить основной блок одной из многих несовершенных систем.

Для простоты будем считать, что InterCITY работает на пути проходящим через C городов, перечисленных последовательно 1, 2, .., C.(1 - номер начального города, C - номер конечного города. В поезде всего S мест, поэтому в нем не может ехать более S пассажиров. Между любыми двумя станциями должны ехать не более S пассажиров.

Система крайне проста. Она должна отвечать на заявки либо согласием, либо отказом. Заявка принята, если на всем отрезки пути, есть достаточное количество свободных мест. В противном случае, заявка должна быть отвергнута. Частичный прием заявки не допускается. Например, для части маршрута или для меньшего количества пассажиров. Если заявка принята, то количество свободных мест должно быть пересчитано. Заявки обрабатываются в порядке их поступления.

Входные данные:

В первой строке записаны три целых числа c, s и r (1<=c<=60 000, 1<=s<=60 000, 1<=r<=60 000), разделенные одним пробелом. Числа обозначают соответственно: количество городов на железнодорожной линии, количество мест в поезде, и количестве просьб. На следующих r линиях записаны последовательно просьбы. В i+1 строке описана i-тая просьба.

Описание состоит из трех целых чисел o, d и n (1 <= o < d <=c, 1<=n<=s), разделенных одним пробелом. Они обозначают: номер начальной станции, номер конечной станциии и попрошенное количество мест, соответственно.

Выходные данные:

Ваша программа должна выдать r строк. На i-той строке должен быть точно один символ: T (для "да") - если i-тая просьба принята, N (для "нет") - в противном случае.

Пример:

input.txt	output.txt
4 6 4	T

1 4 2 1 3 2 2 4 3 1 2 3	T N N

Комментарий.

Допустим, имеется массив X такой, что X[i] — количество пассажиров, едущих от станции i к станции i + 1. Тогда для принятия заявки на отрезке [o, d - 1] все числа должны быть не больше s – n, или, что аналогично, максимальное из них не больше s – n. После принятия заявки все числа на этом отрезке надо увеличить на n. Оба действия (поиск максимума и интервальное изменение) позволяет выполнять дерево отрезков с дополнительным полем s, показывающим, насколько надо увеличить каждый из элементов отрезка, соответствующего данному узлу.

Программа.

```
#include <stdio.h>

// Дерево отрезков
struct {
  int l, r, m, s;
} a[1 << 19];

int min(int a, int b) {
  return a < b ? a : b;
}

int max(int a, int b) {
  return a > b ? a : b;
}

// Инициализация дерева
void t_init(int x, int l, int r) {
  a[x].l = l; a[x].r = r;
  a[x].m = a[x].s = 0;
  if (l != r) {
    t_init((x << 1) + 1, l, (l + r) >> 1);
    t_init((x << 1) + 2, ((l + r) >> 1) + 1, r);
  }
}

// Изменение интервала
void t_modify(int x, int l, int r, int k) {
  if ((l == a[x].l) && (r == a[x].r)) a[x].s += k;
  else {
    if (l <= a[(x << 1) + 1].r) t_modify((x << 1) + 1, l, min(r, a[(x << 1) + 1].r), k);
    if (r >= a[(x << 1) + 2].l) t_modify((x << 1) + 2, max(l, a[(x << 1) + 2].l), r, k);
    a[x].m = a[(x << 1) + 1].m + a[(x << 1) + 1].s;
    if (a[(x << 1) + 2].m + a[(x << 1) + 2].s> a[x].m) a[x].m = a[(x << 1) + 2].m + a[(x <<
1) + 2].s;
  }
}

// Выбор максимума
int t_max(int x, int l, int r, int s) {
  int r1, r2;
  if ((l == a[x].l) && (r == a[x].r)) return s + a[x].m;
  else {
    r1 = r2 = -2000000000;
    if (l <= a[(x << 1) + 1].r) r1 = t_max((x << 1) + 1, l, min(r, a[(x << 1) + 1].r), s +
a[(x << 1) + 1].s);
    if (r >= a[(x << 1) + 2].l) r2 = t_max((x << 1) + 2, max(l, a[(x << 1) + 2].l), r, s +
a[(x << 1) + 2].s);
    return r1 > r2 ? r1 : r2;
  }
}

int main() {
  int n, s, m, t, a, b;

  freopen("input.txt", "r", stdin);
  freopen("output.txt ", "w", stdout);
```

```
  scanf("%d %d %d", &n, &s, &m);
  t = 1;
  while (t < n) t <<= 1;
  n = t;
  t_init(0, 0, n - 1);
  while (m > 0) {
    scanf("%d %d %d", &a, &b, &t);
    a--; b -= 2;
    if (t_max(0, a, b, 0) + t <= s) {
      t_modify(0, a, b, t);
      printf("T\n");
    } else printf("N\n");
    m--;
  }
  return 0;
}
```

Задача 13. Shaping regions[http://usaco.org].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

N opaque rectangles (1 <= N <= 1000) of various colors are placed on a white sheet of paper whose size is A wide by B long. The rectangles are put with their sides parallel to the sheet's borders. All rectangles fall within the borders of the sheet so that different figures of different colors will be seen.

The coordinate system has its origin (0,0) at the sheet's lower left corner with axes parallel to the sheet's borders.

Входные данные:

The order of the input lines dictates the order of laying down the rectangles. The first input line is a rectangle "on the bottom".

Line 1: A, B, and N, space separated (1 <= A,B <= 10,000)

Lines 2-N+1: Five integers: llx, lly, urx, ury, color: the lower left coordinates and upper right coordinates of the rectangle whose color is `color' (1 <= color <= 2500) to be placed on the white sheet. The color 1 is the same color of white as the sheet upon which the rectangles are placed.

Выходные данные:

The output file should contain a list of all the colors that can be seen along with the total area of each color that can be seen (even if the regions of color are disjoint), ordered by increasing color. Do not display colors with no area.

Пример:

input.txt	output.txt
11111111111111111111 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 33333333443333333331 11222222442222222211 11222222442222222211 11222222442222222211 11222222442222222211 11222222442222222211 11222222442222222211 11111111441111111111 11111111441111111111	1 91 2 84 3 187 4 38

Комментарий.

Рассмотрим полученный рисунок по строкам. Очевидно, что строка i + 1 будет отличаться от строки i, тогда и только тогда, когда на одной из них будет начинаться или заканчиваться прямоугольник. Тогда, определив для каждой из таких строк (а их будет не больше 2N) их вид (площадь каждого цвета в них), можно вычислить и площадь каждого цвета во всем рисунке.

Для определения вида каждой строки будем использовать кучу.

Заведем два массива: Y – будет хранить координаты строк, в которых начинаются или кончаются прямоугольники, X – будет хранить абсциссы начал и концов прямоугольника (x), признак начала или конца (f) и номер прямоугольника (i). Оба массива отсортируем в порядке неубывания координат.

Организуем цикл по элементам массива Y. При этом в начале его будем обновлять площадь каждого цвета c[j], добавляя вычисленные в конце предыдущей операции площади цветов в предыдущей строке dc[j] помноженные на разность координат строк Y[i] – Y[i – 1] (где i — индекс текущей строки).

Во второй части цикла проходим по всем прямоугольникам, которые покрывают данную строку и вычисляем площадь каждого цвета.

Программа.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

const int MAXN = 2000;
const int MAXM = 2500;

// Исходные прямоугольники
struct {
    int x1, y1, x2, y2, c;
} p[MAXN];
// Абсциссы начал и концов
struct {
    int x, i;
    bool f;
} X[MAXN << 1];
// Координаты строк
int Y[MAXN << 1];
// Площади цветов (в общем и для текущей строки)
int c[MAXM], dc[MAXM];

// Индексированная куча
class Heap {
    int h[MAXN + 1];
    int hi[MAXN + 1];
    int n;
    void siftUp(int x) {
        int y;
        y = h[x];
        while ((x > 0) && (y > h[(x - 1) >> 1])) {
            h[x] = h[(x - 1) >> 1];
            hi[h[x]] = x;
            x = (x - 1) >> 1;
        }
        h[x] = y;
        hi[y] = x;
    }
    void siftDown(int x) {
        int y, c;
        y = h[x];
        while (x < n >> 1) {
            c = (x << 1) + 1;
            if ((c < n - 1) && (h[c + 1] > h[c])) ++c;
            if (y > h[c]) break;
            h[x] = h[c];
            hi[h[x]] = x;
            x = c;
        }
        h[x] = y;
        hi[y] = x;
    }
public:
    Heap() {
        n = 0;
```

```
        for (int i = 0; i <= MAXN; ++i) hi[i] = -1;
    }
    void add(int x) {
        h[n++] = x; hi[x] = n - 1;
        siftUp(n - 1);
    }
    void remove(int x) {
        int y;
        if (hi[x] == -1) return;
        y = x;
        x = hi[x];
        hi[y] = -1;
        h[x] = MAXN;
        siftUp(x);
        h[0] = h[--n];
        hi[h[0]] = 0;
        siftDown(0);
    }
    int top() {
        if (!n) return -1;
        return h[0];
    }
} heap;

// Сортировка X по полю x
void sort1(int l, int r) {
    int i, j, t, x;
    bool tt;
    x = X[l + rand() % (r - l + 1)].x;
    i = l; j = r;
    do {
        while (X[i].x < x) ++i;
        while (X[j].x > x) --j;
        if (i <= j) {
            t = X[i].x; X[i].x = X[j].x; X[j].x = t;
            t = X[i].i; X[i].i = X[j].i; X[j].i = t;
            tt = X[i].f; X[i].f = X[j].f; X[j].f = tt;
            ++i; --j;
        }
    } while (i <= j);
    if (j > l) sort1(l, j);
    if (i < r) sort1(i, r);
}

// Сортировка Y по полю y
void sort2(int l, int r) {
    int i, j, t, x;
    x = Y[l + rand() % (r - l + 1)];
    i = l; j = r;
    do {
        while (Y[i] < x) ++i;
        while (Y[j] > x) --j;
        if (i <= j) {
            t = Y[i]; Y[i] = Y[j]; Y[j] = t;
            ++i; --j;
        }
    } while (i <= j);
    if (j > l) sort2(l, j);
    if (i < r) sort2(i, r);
}

int main() {
    int a, b, n, m, xc, yc, i, j, k, x, y, px, color;

// Ввод
    srand(time(0));
    scanf("%d %d %d", &a, &b, &n);
    m = 1;
    xc = yc = 0;
    p[0].x1 = 0; p[0].y1 = 0; p[0].x2 = a; p[0].y2 = b; p[0].c = 0;
    X[xc].x = 0; X[xc].i = 0; X[xc].f = false; ++xc;
    X[xc].x = a; X[xc].i = 0; X[xc].f = true; ++xc;
    Y[yc++] = 0; Y[yc++] = b;
```

```
    for (i = 0, j = 1; i < n; ++i) {
        scanf("%d %d %d %d %d", &p[j].x1, &p[j].y1, &p[j].x2, &p[j].y2, &p[j].c);
        if (p[j].x1 < 0) p[j].x1 = 0;
        if (p[j].x2 < 0) p[j].x2 = 0;
        if (p[j].y1 < 0) p[j].y1 = 0;
        if (p[j].y2 < 0) p[j].y2 = 0;
        if (p[j].x1 > a) p[j].x1 = a;
        if (p[j].x2 > a) p[j].x2 = a;
        if (p[j].y1 > b) p[j].y1 = b;
        if (p[j].y2 > b) p[j].y2 = b;
        if (p[j].x1 > p[j].x2) {
            p[j].x1 ^= p[j].x2; p[j].x2 ^= p[j].x1; p[j].x1 ^= p[j].x2;
        }
        if (p[j].y1 > p[j].y2) {
            p[j].y1 ^= p[j].y2; p[j].y2 ^= p[j].y1; p[j].y1 ^= p[j].y2;
        }
        if ((p[j].x2 - p[j].x1) * (p[j].y2 - p[j].y1) > 0) {
            X[xc].x = p[j].x1; X[xc].i = j; X[xc].f = false; ++xc;
            X[xc].x = p[j].x2; X[xc].i = j; X[xc].f = true; ++xc;
            Y[yc++] = p[j].y1;
            Y[yc++] = p[j].y2;
            if (p[j].c > m) m = p[j].c;
            --p[j].c;
            ++j;
        }
    }
    n = j;

// Подготовка
    if (xc > 1) sort1(0, xc - 1);
    if (yc > 1) sort2(0, yc - 1);

    for (i = j = 1; i < yc; ++i) if (Y[i] != Y[j - 1]) Y[j++] = Y[i];
    yc = j;

// Проход по всем строкам
    for (i = 0; Y[i] < b; ++i) {
// Обновление площадей цветов
        if (i)
            for (j = 0; j < m; ++j) {
                c[j] += dc[j] * (Y[i] - Y[i - 1]);
                dc[j] = 0;
            }
        y = Y[i];
        px = 0;
        color = 0;
// Вычисление площади каждого цвета в текущей строке
        for (j = 0; j < xc; j = k) {
// Вычисление цвета текущего отрезка строки
            x = X[j].x;
            for (k = j; (k < xc) && (X[k].x == x); ++k)
                if ((p[X[k].i].y1 <= y) && (p[X[k].i].y2 >= y + 1)) {
                    if (X[k].f) heap.remove(X[k].i);
                    else heap.add(X[k].i);
                }
            while ((k < xc) && ((p[X[k].i].y1 > y) || (p[X[k].i].y2 < y + 1))) ++k;
            dc[color] += x - px;
            px = x;
// Обновление информации о цвете
            if (heap.top() != -1) color = p[heap.top()].c;
        }
    }
    if (i) for (j = 0; j < m; ++j) c[j] += dc[j] * (b - Y[i - 1]);

// Вывод
    for (i = 0; i < m; ++i) if (c[i]) printf("%d %d\n", i + 1, c[i]);
    return 0;
}
```

ГЛАВА 3. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ.

Задача 14. City game [SEERC 2004].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Bob is a strategy game programming specialist. In his new city building game the gaming environment is as follows: a city is built up by areas, in which there are streets, trees, factories and buildings. There is still some space in the area that is unoccupied. The strategic task of his game is to win as much rent money from these free spaces. To win rent money you must erect buildings, that can only be rectangular, as long and wide as you can. Bob is trying to find a way to build the biggest possible building in each area. But he comes across some problems – he is not allowed to destroy already existing buildings, trees, factories and streets in the area he is building in.

Each area has its width and length. The area is divided into a grid of equal square units.

The rent paid for each unit on which you're building stands is 3$.

Your task is to help Bob solve this problem. The whole city is divided into K areas. Each one of the areas is rectangular and has a different grid size with its own length M and width N. The existing occupied units are marked with the symbol R. The unoccupied units are marked with the symbol F.

Входные данные:

The first line of the input file contains an integer K – determining the number of datasets. Next lines contain the area descriptions. One description is defined in the following way: The first line contains two integers-area length M<=1000 and width N<=1000, separated by a blank space. The next M lines contain N symbols that mark the reserved or free grid units, separated by a blank space. The symbols used are:

R – reserved unit

F – free unit

In the end of each area description there is a separating line.

Выходные данные:

For each data set in the input file print on a separate line, on the standard output, the integer that represents the profit obtained by erecting the largest building in the area encoded by the data set. Sample input and output:

Пример:

input.txt	output.txt
2 5 6 R F F F F F F F F F F F R R R F F F F F F F F F F F F F F F 5 5 R R R R R R R R R R R R R R R R R R R R R R R R R	45 0

Комментарий.

Задача заключается в следующем: найти прямоугольник максимальной площади, не содержащий зданий. Основная идея решения — для каждой пустой клетки (i, j) найти прямоугольник максимальной площади такой, что он содержит эту клетку, лежит не левее j и его ширина равна количеству клеток, не содержащих зданий правее j, включая j. Перебирая все такие прямоугольники находим максимальную площадь.

Нахождение ширины прямоугольника для каждой клетки производится проходом справа налево, при этом, если текущая клетка пустая, то пишем в нее ширину прямоугольника для предыдущей клетки плюс один, если текущая клетка содержит здание, пишем в нее 0.

Нахождение нижней и верхней границ прямоугольника для каждой клетки осуществляется одним проходом с использованием стека.

Программа.

```
#include <stdio.h>

// Исходная матрица
bool a[1000][1000];
// Ширина прямоугольников
int b[1000][1000];
// Стек
int st[1000];
// Нижняя и верхняя границы прямоугольников
int r[1000];
int l[1000];

int main() {
  int t, n, m, i, j, p, h;
  char s[10];

  scanf("%d", &t);
  while (t > 0) {
// Ввод
    scanf("%d %d", &n, &m);
    for (i = 0; i < n; i++) for (j = 0; j < m; j++) {
      scanf("%s", s); a[i][j] = s[0] == 'R';
    }
// Вычисление ширины прямоугольников
    for (i = 0; i < n; i++) b[i][m - 1] = 1 - a[i][m - 1];
    for (j = m - 2; j >= 0; j--) for (i = 0; i < n; i++)
      if (a[i][j]) b[i][j] = 0; else b[i][j] = b[i][j + 1] + 1;
    p = 0;
    for (j = 0; j < m; j++) {
// Нахождение верхних границ прямоугольников, левая граница которых находится на j-м столбце
      l[0] = 0;
      st[0] = 0; h = 1;
      for (i = 1; i < n; i++) {
        l[i] = i;
        while ((h > 0) && (b[i][j] <= b[st[h - 1]][j])) l[i] = l[st[--h]];
        st[h++] = i;
      }
// Нахождение нижних границ прямоугольников, левая граница которых находится на j-м столбце
      r[n - 1] = n - 1;
      st[0] = n - 1; h = 1;
      for (i = n - 2; i >= 0; i--) {
        r[i] = i;
        while ((h > 0) && (b[i][j] <= b[st[h - 1]][j])) r[i] = r[st[--h]];
        st[h++] = i;
      }
// Выбор максимального по площади прямоугольника
      for (i = 0; i < n; i++) if ((r[i] - l[i] + 1) * b[i][j] > p) p = (r[i] - l[i] + 1) *
b[i][j];
    }
// Вывод
    printf("%d\n", p * 3);
    t--;
  }
  return 0;
}
```

Задача 15. Folding [NEERC 2002].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Bill is trying to compactly represent sequences of capital alphabetic characters from 'A' to 'Z' by folding repeating subsequences inside them. For example, one way to represent a sequence AAAAAAAAAABABABCCD is 10(A)2(BA)B2(C)D. He formally defines folded sequences of characters along with the unfolding transformation for them in the following way:

- A sequence that contains a single character from 'A' to 'Z' is considered to be a folded sequence. Unfolding of this sequence produces the same sequence of a single character itself.

- If S and Q are folded sequences, then SQ is also a folded sequence. If S unfolds to S' and Q unfolds to Q', then SQ unfolds to S'Q'.
- If S is a folded sequence, then X(S) is also a folded sequence, where X is a decimal representation of an integer number greater than 1. If S unfolds to S', then X(S) unfolds to S' repeated X times.

According to this definition it is easy to unfold any given folded sequence. However, Bill is much more interested in the reverse transformation. He wants to fold the given sequence in such a way that the resulting folded sequence contains the least possible number of characters.

Входные данные:

The input file contains a single line of characters from 'A' to 'Z' with at least 1 and at most 100 characters.

Выходные данные:

Write to the output file a single line that contains the shortest possible folded sequence that unfolds to the sequence that is given in the input file. If there are many such sequences then write any one of them.

Пример

input.txt	output.txt
AAAAAAAAAABABABCCD	9(A)3(AB)CCD
NEERCYESYESYESNEERCYESYESYES	2(NEERC3(YES))

Комментарий.

Основная идея — разбить задачу размерностью (длиной строки) n на подзадачи меньшей размерности, скомбинировав результаты решения которых, можно получить решение исходной задачи.

Программа.

```
#include <stdio.h>
#include <string.h>

// Исходная строка
char s[101];
// Временная строка
char ts[101];
// Матрица решений
struct {
  char c;
  char *s;
} a[100][100];

// Длина числа в цифрах
int numlen(int a) {
  if (a < 10) return 1;
  if (a < 100) return 2;
  if (a < 1000) return 3;
  return 4;
}

// Длина строки
int slen(int l, int r) {
  int len;
  if (a[l][r].c > 1) len = numlen(a[l][r].c) + 2;
  else len = 0;
  len += strlen(a[l][r].s);
  return len;
}

// Перевод числа в строку
void itoa(int a, char* s) {
  int i, j;
  char ts[10];
  i = 0;
  do {
    ts[i++] = a % 10 + '0';
    a /= 10;
```

```
  } while (a > 0);
  for (j = 0; j < i; j++) s[i - j - 1] = ts[j];
  s[i] = 0;
}

// Решение задачи для подстроки l..r
void test(int l, int r) {
  int i, j, k, m, mi;
  bool f;

// Если уже решали, то выход
  if (a[l][r].c != -1) return;
// Обработка тривиального случая
  if (l == r) {
    a[l][r].s[0] = s[l];
    a[l][r].s[1] = 0;
    a[l][r].c = 1;
    return;
  }
// Обработка общего случая (имеет смысл сжимать, только строки длиной не менее 5)
  if (r - l + 1 >= 5) {
    for (i = l + 1; i <= r; i++) {
      test(l, i - 1);
      test(i, r);
      if (strcmp(a[l][i - 1].s, a[i][r].s) == 0) {
        if ((a[l][r].c == -1) ||
          (strlen(a[l][r].s) > strlen(a[l][i - 1].s) + numlen(a[l][i - 1].c + a[i][r].c) +
2)) {
          a[l][r].c = a[l][i - 1].c + a[i][r].c;
          strcpy(a[l][r].s, a[l][i - 1].s);
        }
      } else if ((a[l][r].c == -1) || (strlen(a[l][r].s) > slen(l, i - 1) + slen(i, r))) {
        if (a[l][i - 1].c > 1) {
          itoa(a[l][i - 1].c, a[l][r].s);
          strcat(a[l][r].s, "(");
          strcat(a[l][r].s, a[l][i - 1].s);
          strcat(a[l][r].s, ")");
        } else strcpy(a[l][r].s, a[l][i - 1].s);
        if (a[i][r].c > 1) {
          itoa(a[i][r].c, ts);
          strcat(a[l][r].s, ts);
          strcat(a[l][r].s, "(");
          strcat(a[l][r].s, a[i][r].s);
          strcat(a[l][r].s, ")");
        } else strcat(a[l][r].s, a[i][r].s);
        a[l][r].c = 1;
      }
    }
    m = slen(l, r);
    mi = -1;
    for (i = 1; i < r - l + 1; i++)
      if ((r - l + 1) % i == 0) {
        f = true;
        for (j = l + i; (j <= r) && f; j += i)
          for (k = 0; k < i; k++) if (s[j + k] != s[l + k]) f = false;
        if (f && (i + 2 + numlen((r - l + 1) / i) < m)) {
          mi = i;
          m = i + 2 + numlen((r - l + 1) / i);
        }
      }
    if (mi != -1) {
      a[l][r].s[mi] = 0;
      a[l][r].c = (r - l + 1) / mi;
    }
  } else {
    strncpy(a[l][r].s, s + l, r - l + 1);
    a[l][r].s[r - l + 1] = 0;
    a[l][r].c = 1;
  }
}

int main() {
  int i, j, n;
```

```
// Ввод и инициализация
  scanf("%s", s);
  n = strlen(s);
  for (i = 0; i < n; i++) for (j = 0; j < n; j++) {
    a[i][j].c = -1;
    if (j >= i) a[i][j].s = new char[j - i + 2];
  }
// Решение
  test(0, n - 1);
// Вывод
  if (a[0][n - 1].c > 1) printf("%d(%s)\n", a[0][n - 1].c, a[0][n - 1].s);
  else printf("%s\n", a[0][n - 1].s);
  return 0;
}
```

Задача 16. Strategic game [SEERC 2000].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Bob enjoys playing computer games, especially strategic games, but sometimes he cannot find the solution fast enough and then he is very sad. Now he has the following problem. He must defend a medieval city, the roads of which form a tree. He has to put the minimum number of soldiers on the nodes so that they can observe all the edges. Can you help him?

Your program should find the minimum number of soldiers that Bob has to put for a given tree.

Входные данные:

The input file contains several data sets in text format. Each data set represents a tree with the following description:

- the number of nodes
- the description of each node in the following format
  node_identifier:(number_of_roads) node_identifier$_1$ node_identifier$_2$ … node_identifier$_{number_of_roads}$
  or
  node_identifier:(0)

The node identifiers are integer numbers between 0 and n-1, for n nodes (0 < n <= 1500). Every edge appears only once in the input data.

Выходные данные:

The output should be printed on the standard output. For each given input data set, print one integer number in a single line that gives the result (the minimum number of soldiers).

Пример

input.txt	output.txt
4 0:(1) 1 1:(2) 2 3 2:(0) 3:(0) 5 3:(3) 1 4 2 1:(1) 0 2:(0) 0:(0) 4:(0)	1 2

Комментарий.

Решим задачу для поддерева каждой вершины-потомка корня, в двух вариантах: включая стартовую вершину и не включая. После этого найденные решения можно скомбинировать и получить решение исходной задачи.

Программа.

```
#include <stdio.h>
#include <ctype.h>

// Структура для хранения графа
int f[1500];
struct {
  int n, t;
} e[10000];
// Матрица решений
int w[1500][2];

// Ввод целого числа
bool getint(int& a) {
  int c;
  c = getchar(); while ((c != EOF) && (!isdigit(c))) c = getchar();
  if (c == EOF) return false;
  a = 0;
  while (isdigit(c)) {
    a = a * 10 + c - '0';
    c = getchar();
  }
  return true;
}

// Решение задачи для поддерева вершины v с предком p
int dfs(int v, int p, int k) {
  int i, a, b;
// Если уже решали, то выйти
  if (w[v][k] != -1) return w[v][k];
// Вариант без включения вершины v
  if (k == 0) {
    w[v][k] = 0;
    for (i = f[v]; i != -1; i = e[i].n) if (e[i].t != p) {
      w[v][k] += dfs(e[i].t, v, 1);
    }
  } else {
// Вариант с включением вершины v
    w[v][k] = 1;
    for (i = f[v]; i != -1; i = e[i].n) if (e[i].t != p) {
      a = dfs(e[i].t, v, 0);
      b = dfs(e[i].t, v, 1);
      w[v][k] += a < b ? a : b;
    }
  }
  return w[v][k];
}

int main() {
  int n, a, b, c, i, j, k, h, t, v;
  while (getint(n)) {
// Ввод
    for (i = 0; i < n; i++) f[i] = -1;
    k = 0;
    for (i = 0; i < n; i++) {
      w[i][0] = w[i][1] = -1;
      getint(a); getint(c);
      for (j = 0; j < c; j++) {
        getint(b);
        e[k].n = f[a]; e[k].t = b; f[a] = k++;
        e[k].n = f[b]; e[k].t = a; f[b] = k++;
      }
    }
// Решение
    a = dfs(0, -1, 0); b = dfs(0, -1, 1);
// Вывод
    printf("%d\n", a < b ? a : b);
  }
  return 0;
}
```

Задача 17. Короли [http://acm.sgu.ru].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Определить количество способов, которыми можно расставить k королей (ходят во все стороны, за один ход передвигаются на одну клетку) на шахматной доске n x n так, чтобы никакие два из них не били друг друга.

Входные данные:

На первой строке входного файла записаны два целых числа: n и k ($1 \le n \le 10$; $0 \le k \le n2$).

Выходные данные:

В выходной файл выведите единственное число — количество расстановок.

Пример

input.txt	output.txt
4 4	79

Комментарий.

Данная задача решается методом «динамика по профилю». Профиль — способ расстановки королей на одной строке. Основная идея — вычислим количество расстановок k королей на доске m x n (где $m \le n$), ограниченных профилями i и j для всех пар i и j. После этого можно сделать то же самое для m + 1, а в конце нужно просуммировать найденные значения по всем i, j.

Программа

```
#include <stdio.h>
#include <string.h>

const int BITS[11] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};

// Входные данные
int n, k;
// Правильные пары смежных профилей
int valid[1024][1024];
// Правильные профили
int right[1024];
// Количество правильных профилей
int r_count;
// d[i] - количество профилей, образующих правильную пару с профилем i
int d[1024];
// c[i] - количество королей в профиле i
int c[1024];
// Результат
__int64 res;
// Матрица ответов
__int64 a[2][200][200][26];
// Текущая строка матрицы ответов
int ind;

int main() {
    int i;

// Ввод и обработка тривиальных случаев
    scanf("%d %d", &n, &k);
    if (k == 0) {
        printf("1\n");
        return 0;
    }
    if (k == 1) {
        printf("%d\n", n * n);
        return 0;
    }
    if (k > 25) {
```

```
        printf("0\n");
        return 0;
    }
// Построение правильных профилей
    r_count = 0;
    for (i = 0; i < (1 << n); i++) {
        bool b = true;
        for (int j = 1; (j < n) && b; j++)
            if ((i & BITS[j]) && (i & BITS[j - 1])) b = false;
        if (b) {
            d[r_count] = 0;
            right[r_count++] = i;
        }
    }
// Построение правильных пар профилей
    for (i = 0; i < r_count; i++) {
        for (int j = i + 1; j < r_count; j++) {
            bool v = true;
            for (int l = 0; (l < n) && v; l++) {
                if (right[i] & BITS[l]) {
                    if (l == 0) {
                        if ((right[j] & BITS[l]) || (right[j] & BITS[l + 1])) v = false;
                    }
                    else
                    if (l == n - 1) {
                        if ((right[j] & BITS[l]) || (right[j] & BITS[l - 1])) v = false;
                    }
                    else {
                        if ((right[j] & BITS[l]) || (right[j] & BITS[l + 1]) || (right[j] &
BITS[l - 1]))
                            v = false;
                    }
                }
                else
                if (right[j] & BITS[l]) {
                    if (l == 0) {
                        if ((right[i] & BITS[l]) || (right[i] & BITS[l + 1])) v = false;
                    }
                    else
                    if (l == n - 1) {
                        if ((right[i] & BITS[l]) || (right[i] & BITS[l - 1])) v = false;
                    }
                    else {
                        if ((right[i] & BITS[l]) || (right[i] & BITS[l + 1]) || (right[i] &
BITS[l - 1]))
                            v = false;
                    }
                }
            }
            if (v) {
                valid[i][d[i]++] = j;
                valid[j][d[j]++] = i;
            }
        }
    }
    valid[0][d[0]++] = 0;
// Вычисление количества фигур в каждом профиле
    for (i = 0; i < r_count; i++) {
        c[i] = 0;
        for (int j = 0; j < n; j++)
            if (right[i] & BITS[j]) c[i]++;
    }
    ind = 0;

// Решение
    for (i = 0; i < r_count; i++)
        for (int j = 0; j < r_count; j++)
            for (int l = 0; l <= k; l++)
                a[0][i][j][l] = 0;
    for (i = 0; i < r_count; i++) {
        for (int j = 0; j <= k; j++) {
            if (c[i] == j) a[ind][i][i][j] = 1;
            else a[ind][i][i][j] = 0;
```

```
        }
    }
    while (n > 1) {
        ind = 1 - ind;
        for (i = 0; i < r_count; i++)
            for (int j = 0; j < r_count; j++)
                for (int l = 0; l <= k; l++)
                    a[ind][i][j][l] = 0;
        for (i = 0; i < r_count; i++) {
            for (int j = i; j < r_count; j++) {
                for (int l = 0; l < d[j]; l++) {
                        for (int k1 = c[j], k2 = 0; k1 <= k; k1++, k2++) {
                                a[ind][i][j][k1] += a[1 - ind][i][valid[j][l]][k2];
                                a[ind][j][i][k1] = a[ind][i][j][k1];
                        }
                }
            }
        }
        n--;
    }
// Подсчет результата и вывод
    res = 0;
    for (i = 0; i < r_count; i++)
        for (int j = 0; j < r_count; j++)
            res += a[ind][i][j][k];
    printf("%I64d\n", res);
    return 0;
}
```

Задача 18. Tour [SEERC 2005].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

John Doe, a skilled pilot, enjoys traveling. While on vacation, he rents a small plane and starts visiting beautiful places. To save money, John must determine the shortest closed tour that connects his destinations. Each destination is represented by a point in the plane pi = <xi,yi>. John uses the following strategy: he starts from the leftmost point, then he goes strictly left to right to the rightmost point, and then he goes strictly right back to the starting point. It is known that the points have distinct x-coordinates.

Write a program that, given a set of n points in the plane, computes the shortest closed tour that connects the points according to John's strategy.

Входные данные:

The program input is from a text file. Each data set in the file stands for a particular set of points. For each set of points the data set contains the number of points, and the point coordinates in ascending order of the x coordinate. White spaces can occur freely in input. The input data are correct.

Выходные данные:

For each set of data, your program should print the result to the standard output from the beginning of a line.

Пример

input.txt	output.txt
3 1 1 2 3 3 1 4 1 1 2 3 3 1 4 2	6.47

Комментарий.

Будем хранить матрицу d, элемент d[i, j] которой показывает минимальную длину цепи, начинающейся в i, заканчивающейся в j и проходящей через все точки левее самой правой из i и j ровно один раз. Это значение несложно пересчитывать для последующих i и j. Ответом к задаче будет являться d[n – 1, n – 1].

Программа

```
#include <stdio.h>
#include <math.h>

const double INF = 1e20;

// Исходные точки
struct {
    int x, y;
} p[200];
// Матрица ответов
double d[200][200];

// Расстояние между точками на плоскости
double dist(int x1, int y1, int x2, int y2) {
    return sqrt((double)((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)));
}

int main() {
    int n, i, j, k;

    while (scanf("%d", &n) == 1) {
// Ввод и сортировка
        for (i = 0; i < n; ++i) scanf("%d %d", &p[i].x, &p[i].y);
        for (i = 0; i < n; ++i) for (j = i + 1; j < n; ++j)
            if (p[i].x > p[j].x) {
                p[i].x ^= p[j].x; p[j].x ^= p[i].x; p[i].x ^= p[j].x;
                p[i].y ^= p[j].y; p[j].y ^= p[i].y; p[i].y ^= p[j].y;
            }
// Решение
        for (i = 0; i < n; ++i) for (j = 0; j < n; ++j)
            if (!i && !j) d[i][j] = 0.0;
            else {
                d[i][j] = INF;
                if ((i - 1 > j) || ((i - 1 == j) && !j)) {
                    if (d[i][j] > d[i - 1][j] + dist(p[i - 1].x, p[i - 1].y,
                          p[i].x, p[i].y))
                        d[i][j] = d[i - 1][j] + dist(p[i - 1].x, p[i - 1].y,
                          p[i].x, p[i].y);
                }
                if ((j - 1 > i) || ((j - 1 == i) && !i)) {
                    if (d[i][j] = d[i][j - 1] + dist(p[j - 1].x, p[j - 1].y,
                          p[j].x, p[j].y))
                        d[i][j] = d[i][j - 1] + dist(p[j - 1].x, p[j - 1].y,
                          p[j].x, p[j].y);
                }
                if (i - 1 <= j) {
                    for (k = 0; k < i - 1; ++k)
                        if (d[i][j] > d[k][j] + dist(p[k].x, p[k].y, p[i].x, p[i].y))
                            d[i][j] = d[k][j] + dist(p[k].x, p[k].y, p[i].x, p[i].y);
                }
                if (j - 1 <= i) {
                    for (k = 0; k < j - 1; ++k)
                        if (d[i][j] > d[i][k] + dist(p[k].x, p[k].y, p[j].x, p[j].y))
                            d[i][j] = d[i][k] + dist(p[k].x, p[k].y, p[j].x, p[j].y);
                }
            }
// Вывод
        printf("%.2lf\n", d[n - 1][n - 1]);
    }
    return 0;
}
```

Задача 19. Trip [CEOI 2003].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Alice and Bob want to go on holiday. Each of them has planned a route, which is a list of cities to be visited in a given order. A route may contain a city more than once.

As they want to travel together, they have to agree on a common route. None wants to change the order of the cities on his or her route or add other cities. Therefore they have no choice but to remove some cities from the route. Of course the common route should be as long as possible.

There are exactly 26 cities in the region. Therefore they are encoded on the lists as lower case letters from 'a' to 'z'.

Входные данные:

The input file trip.in consists of two lines; the first line is Alice's list, the second line is Bob's list. Each list consists of 1 to 80 lower case letters with no spaces inbetween.

Выходные данные:

The output file trip.out should contain all routes that meet the conditions described above, but no route should be listed more than once. Each route should be printed on a separate line. There is at least one such non-empty route, but never more than 1000 different ones. The order of the routes in the output file doesn't matter.

Пример

input.txt	output.txt
Abcabcaa Acbacba	ababa abaca abcba acbca acaba acaca acbaa

Комментарий.

Задача заключается в нахождении всех возможных наибольших общих подпоследовательностей двух последовательностей.

Решим задачу нахождения наибольшей общей подпоследовательности. Для этого заполним матрицу a, элемент a[i, j] которой равен длине НОП префикса i строки 1 и префикса j строки 2. После этого все возможные НОП можно выписать поиском в глубину по этой матрице.

Программа.

```
#include <stdio.h>
#include <string.h>

// Исходные строки и выводимая строка
char s[100], t[100], p[100];
// Заполняемая матрица
int a[100][100];
// Бор (используется для устранения дубликатов)
struct {
    int e[26];
} b[200000];
int bc;
// Длины строк
int n, m;

// Создание новой вершины бора
int b_new() {
    int i;
    for (i = 0; i < 26; i++) b[bc].e[i] = -1;
    bc++;
    return bc - 1;
}

// Обход в глубину с выводом
```

```
void dfs(int v, int i, int j, int k) {
  int ti;
  ti = i;
  if ((i < 0) || (j < 0)) return;
  while (i && (a[i][j] == a[i - 1][j])) i--;
  while ((j >= 0) && (i <= ti)) {
    if (s[i] == t[j]) {
      p[k] = s[i];
      if (b[v].e[s[i] - 'a'] == -1) {
        b[v].e[s[i] - 'a'] = b_new();
        if (!k) printf("%s\n", p);
      }
      if (k) dfs(b[v].e[s[i] - 'a'], i - 1, j - 1, k - 1);
    }
    if (j && (a[i][j] == a[i][j - 1])) j--;
    else if ((i <= ti) && (a[i][j] == a[i + 1][j])) i++;
    else break;
  }
}


int main() {
    int i, j;

// Ввод
    scanf("%s %s", s, t);
    n = strlen(s); m = strlen(t);
// Заполнение матрицы a
    for (i = 0; i < n; i++) for (j = 0; j < m; j++)
        if (!i && !j) a[i][j] = s[i] == t[j];
        else {
            a[i][j] = 0;
            if (i && (a[i - 1][j] > a[i][j])) a[i][j] = a[i - 1][j];
            if (j && (a[i][j - 1] > a[i][j])) a[i][j] = a[i][j - 1];
            if (i && j && (s[i] == t[j]))
                if (a[i - 1][j - 1] + 1 > a[i][j]) a[i][j] = a[i - 1][j - 1] + 1;
            if ((!i || !j) && (s[i] == t[j]))
            if (1 > a[i][j]) a[i][j] = 1;
        }
    p[a[n - 1][m - 1]] = 0;
    bc = 0;
    b_new();
// Вывод
    dfs(0, n - 1, m - 1, a[n - 1][m - 1] - 1);
    return 0;
}
```

ГЛАВА 4. ТЕОРИЯ ИГР.

Задача 20. Two heaps.

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**64 мегабайта**

Two players play with two heaps initially composed of A and B stones. Both players have their sets of possible moves: $a_1, a_2, \dots a_k$ for the first, and $b_1, b_2, \dots b_l$ for the second. The first player can take a_i stones from any heap by its move (each time for only one i in range $1 \le i \le k$), and the second can take b_j. Players move in turn. If a player cannot move, she loses. Your task is to determine the winner!

Входные данные:

First line contains integers A and B. The second line starts with integer k, followed by k numbers a_i. The third line has form $l\ b_1\ b_2 \dots b_l$. All numbers satisfy the following requirements: $1 \le A;B \le 1000$, $1 \le k$; $l \le 10$, $1 \le a_i$; $b_j \le 1000$.

Выходные данные:

If first player wins, output First. Otherwise output Second.

Пример:

input.txt	output.txt
2 2 2 1 2 1 1	First
2 2 1 1 2 1 2	Second

Комментарий.

Эта задача — стандартная задача на теорию игр. Состояние игры полностью определяется тремя параметрами: количеством камней в первой куче, во второй и номером ходящего игрока. Всего состояний в игре может быть (A + 1) * (B + 1) * 2 = 2004002, так что вполне возможно хранить для каждого состояния какую-то информацию. Представим состояния игры в виде вершин орграфа. Проведем дугу между вершинами u и v, если из состояния u можно достичь состояния v за один ход. Заметим, что в графе не будет циклов (т.к. в камни можно только убирать, но не добавлять).

В каждой вершине будем хранить информацию, о том выигрышная она или нет. Тогда сразу можно инициализировать вершины, соответствующие концу игры, а остальные вершины инициализируются так: пусть вершина определяет состояние (a, b, p), тогда, если из него можно достичь проигрышного для игрока 1 – p состояния, то оно выигрышное, в противном случае — проигрышное.

Программа.

```
#include <stdio.h>

// Выигрышность состояний
int c[2000][2000][2];
// Исходные данные
int a[2][100];
int n[2];

int main() {
    int A, B, i, j, k;

// Ввод
    scanf("%d %d", &A, &B);
    scanf("%d", &n[0]);
    for (i = 0; i < n[0]; i++) scanf("%d", &a[0][i]);
    scanf("%d", &n[1]);
    for (i = 0; i < n[1]; i++) scanf("%d", &a[1][i]);
    for (i = 0; i < n[0]; i++) for (j = i + 1; j < n[0]; j++)
        if (a[0][i] > a[0][j]) {
            a[0][i] ^= a[0][j]; a[0][j] ^= a[0][i]; a[0][i] ^= a[0][j];
        }
    for (i = 0; i < n[1]; i++) for (j = i + 1; j < n[1]; j++)
```

```
        if (a[1][i] > a[1][j]) {
            a[1][i] ^= a[1][j]; a[1][j] ^= a[1][i]; a[1][i] ^= a[1][j];
        }
// Решение
    for (i = 0; i <= A; i++) for (j = 0; j <= B; j++) {
        if (!i && !j) c[i][j][0] = c[i][j][1] = 0; // Заведомо проигрышное состояние для
любого игрока
        else {
            c[i][j][0] = c[i][j][1] = 0;
// Игрок 0: Пытаемся взять камни из первой кучи и попасть в проигрышное состояние для
другого игрока
            for (k = 0; !c[i][j][0] && (k < n[0]) && (a[0][k] <= i); k++)
                if (!c[i - a[0][k]][j][1]) c[i][j][0] = 1;
// Игрок 0: Пытаемся взять камни из второй кучи и попасть в проигрышное состояние для
другого игрока
            for (k = 0; !c[i][j][0] && (k < n[0]) && (a[0][k] <= j); k++)
                if (!c[i][j - a[0][k]][1]) c[i][j][0] = 1;
// Игрок 1: Пытаемся взять камни из первой кучи и попасть в проигрышное состояние для
другого игрока
            for (k = 0; !c[i][j][1] && (k < n[1]) && (a[1][k] <= i); k++)
                if (!c[i - a[1][k]][j][0]) c[i][j][1] = 1;
// Игрок 1: Пытаемся взять камни из второй кучи и попасть в проигрышное состояние для
другого игрока
            for (k = 0; !c[i][j][1] && (k < n[1]) && (a[1][k] <= j); k++)
                if (!c[i][j - a[1][k]][0]) c[i][j][1] = 1;
        }
    }
// Вывод
    if (c[A][B][0]) printf("First\n");
    else printf("Second\n");
    return 0;
}
```

ГЛАВА 5. ВЫЧИСЛИТЕЛЬНАЯ ГЕОМЕТРИЯ.

Задача 21. Точка в многоугольнике. [ЛКШ'2004]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 1 секунда
Ограничение по памяти 16 мегабайт

Дан произвольный многоугольник и точка. Необходимо определить принадлежит ли точка многоугольнику.

Входные данные: В первой строке входного файла находится число N – количество точек многоугольника (3≤N≤100000). В следующих N строках записаны в порядке обхода пары чисел типа integer задающих координаты соответствующих точек многоугольника. В последней строке записана пара чисел x, y – координаты проверяемой точки.

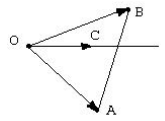
Выходные данные: В выходной файл выведите одно слово YES, если точка принадлежит многоугольнику, и NO, в противном случае.

Пример:

input.txt	output.txt
3 0 0 0 2 2 2 0 1	YES

Комментарий. Идея решения этой задачи состоит в следующем: пустим из точки луч, скажем, вправо. При этом будем считать, сколько сторон многоугольника он пересечет. Стороны многоугольника, нижний коней которых лежит на луче, учитать не будем. Тогда если это число нечетное, то точка лежит внутри многоугольника, в противном случае – нет.

Теперь технические детали. Точку обозначим O. Пусть сторона, которую мы проверяем – AB, причем A – ее нижний конец (см. рис.). Пусть C – любая точка на луче, исходящем из O вправо, отличная от O. Тогда для того чтобы сторона AB удовлетворяла нашим условиям, необходимо проверить выполняются ли следующие неравенства:



расположение стороны AB и точки O

a) [OA, OC] * [OB, OC] ≤ 0, где [x, y] – векторное произведение векторов x и y. Это неравенство означает, что точки A и B не лежат по одну сторону от прямой;
b) [OA, OB] > 0. Это неравенство означает, что точка O лежит "левее" отрезка AB, т.е. луч OC пересекает отрезок AB;
c) [OA, OC] ≠ 0. Это неравенство гарантирует, что точка A не лежит на луче OC.

Если все 3 неравенства выполняются, то будем считать, что луч OC пересекает сторону AB многоугольника.

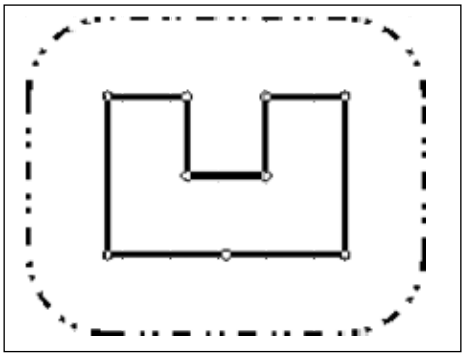
Программа. Приведем программный код, реализующий вышеописанный подход:

```
#include <stdio.h>
#define MAXN 100001

int sign(int x) { // знак числа x
	if (x < 0) return -1;
	return x > 0;
}

int x[MAXN], y[MAXN];

int main() {
	int n, i, x0, y0, c;
	scanf("%d", &n);
	for (i = 0; i < n; i++) // читаем вершины многоугольника
		scanf("%d %d", x + i, y + i);
	x[n] = x[0], y[n] = y[0];
	scanf("%d %d", &x0, &y0);
	for (c = i = 0; i < n; i++) { // идем по сторонам многоугольника
		int a, b;
		if (y[i] == y[i + 1]) continue;
		if (y[i] < y[i + 1]) a = i, b = i + 1;
```

```
            else a = i + 1, b = i;
            // условия
            if (sign(-(y[a] - y0)) * sign(-(y[b] - y0)) <= 0 && // (a)
                  sign((x[a] - x0) * (y[b] - y0) - (x[b] - x0) * (y[a] - y0)) > 0 && //(b)
                  sign(-(y[a] - y0))) // (c)
                  c ^= 1; // меняем четность
      }
      puts(c ? "YES" : "NO");
      return 0;
}
```

Задача 22. Стена. [NEERC'2001, SF]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 1 секунда
Ограничение по памяти 64 мегабайта



Давным-давно жил-был очень жадный Король, который приказал своему Архитектору построить стену вокруг королевского замка. Король был так жаден, что он не стал выслушивать предложения Архитектора и приказал построить стену, использовав минимальное количество камня и других затрат, но потребовал также, чтобы не приближалась к замку на определенное расстояние. Если приказ Короля не будет выполнен, то Архитектор лишится головы. Более того, Король потребовал от Архитектора предоставить план стены с точным количеством ресурсов, которые пойдут на постройку стены.

Ваша задача – помочь бедному Архитектору, написав программу, которая определила бы минимальную длину стены, которая бы удовлетворяла всем требованиям Короля.

Задача несколько упрощается тем фактом, что замок Короля имеет форму многоугольника и расположен на плоскости. Архитектор уже ввел декартову систему координат и измерил все координаты вершин замка в футах.

Входные данные: В первой строке входного файла содержатся целые числа N и L ($3 \le N \le 1000$, $1 \le L \le 1000$) – количество вершин королевского замка и минимальное количество футов, на которое стена может приближаться к замку, соответственно. Следующие N строк содержат координаты вершин замка, перечисленные против часовой стрелки, - два целых числа X_i и Y_i, разделенные пробелом ($-10000 \le X_i, Y_i \le 10000$). Все вершины различны, стороны замка не пересекаются ни в каких других точках, кроме вершин.

Выходные данные: Выведите длину стены, округленную таким образом, чтобы результат был точен до 8 дюймов(в футе 12 дюймов).

Пример:

input.txt	output.txt
9 100 200 400 300 400 300 300 400 300 400 400 500 400 500 200 350 200 200 200	1628

Комментарий. Сначала построим выпуклую оболочку для данного многоугольника. После этого можно счичать, что многоугольник выпуклый. Отодвинем стороны многоугольника на L от самого многоугольника, как это показано на рисунке. Остается их соединить дугами окружности радиуса L с центрами в вершинах многоугольника.

Теперь длина стены равна сумме длин сторон многоугольника + сумма длин дуг окружности радиуса L.

Заметим, что дуги окружности вкупе дают оборот 360 градусов, поэтому сумма длин дуг окружности равна просто длине окружности, т.е. $2\pi L$.

Таким образом, для того чтобы решить задачу, небоходимо вычислить длину выпуклой оболочки многоугольника и сложить ее с $2\pi L$.

Алгоритм построения выпуклой оболочки можно найти в [12].

Время работы алгоритма – *O(N log(N))*.

Задача 23. Фонтан. [Всероссийская командная олимпиада школьников'2000]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 1 секунда
Ограничение по памяти 64 мегабайта

Администрация одного института решила построить в холле фонтан. По плану администрации, фонтан должен иметь форму круга с максимально возможным радиусом. Дизайнеру сообщили, что холл института имеет вид прямоугольника, размером $X \times Y$ метров. Однако когда дизайнер стал выбирать место для фонтана, он столкнулся с серьезной проблемой: в холле института обнаружилось N круглых колонн, снести которые не представляется возможным.

Таким образом, у него появилась проблема: где следует поместить фонтан, чтобы он имел максимально возможный радиус и не имел ненулевого по площади пересечения с колоннами. Вам предстоит помочь ему в решении этой нелегкой задачи.

Входные данные.

На первой строке входного файла находятся вещественные числа X и Y, $1 \le X$, $Y \le 10^4$. Будем считать, что прямоугольник холла расположен на координатной сетке так, что его углы имеют координаты (0, 0), (0, Y), (X, 0) и (X, Y).

На второй строке находится число N ($0 \le N \le 10$) – количество колонн. Следующие N строк содержат параметры колонн - i-я строка содержит три вещественных числа X_i, Y_i и R_i - координаты центра и радиус колонны, все колонны полностью находятся внутри холла и не пересекаются, хотя могут касаться друг друга и стен холла, $0.1 \le R_i$). Все числа во входном файле разделены пробелами.

Выходные данные.

Выведите в выходной файл три вещественных числа: X_F, Y_F и Y_F - координаты центра и радиус фонтана. Фонтан должен быть полностью расположен внутри холла (допускается касание стен) и не иметь ненулевого пересечения ни с одной из колонн (допускается касание). Радиус фонтана должен быть максимален. Разделяйте числа пробелами и/или переводами строки. Если решений несколько, выведите любое из них.

Примечание о точности вычислений: При проверке результата работы Вашей программы нарушение условий оптимальности и отсутствия пересечения со стенами и колоннами менее чем на 10^{-2} будет игнорироваться.

Пример:

input.txt	output.txt
10 20 0	5.000 5.000 5.000
20 20 4 2 2 2 18 2 2 2 18 2 18 18 2	10.000 10.000 9.314
20 20 4 2 2 2 18 2 2 3 17 2 16 16 4	9.510 7.054 7.053

Комментарий. Это самая сложная задача из предлагавшихся на олимпиаде. Идея ее решения следующая: предположим, мы хотим построить фонтан радиуса R. Тогда чтобы проверить, что мы можем это сделать, проделаем следующую операцию: увеличим радиус всех колонн на R, а каждую сторону холла уменьшим на 2R, чтобы его центр остался на том же месте. Тогда, чтобы проверить, что мы можем построить фонтан, достаточно проверить, что после этой операции круги колонн полностью покрывают прямоугольник холла. Если это так, то построить фонтан данного радиуса, очевидно, не удастся.

Чтобы проверить, что множество кругов покрывает заданный прмоугольник можно использовать следующий алгоритм: достаточно проверить, что точки пересечения любых двух окружностей, лежащие внутри прямоугольника, точки пересечения окружностей с прямоугольником и углы прямоугольника покрываются каким-либо другим кругом (область, не покрываемая кругами, всегда содержит одну из таких точек).

Для нахождения точек пересечения двух окружностей необходимо решить следующую систему уравнений относиельно x и y:

После замены $x - x_1 = \bar{x}$, $y - y_1 = \bar{y}$ и обозначения $x_1 - x_2 = \Delta x$, $y_1 - y_2 = \Delta y$, система принимает вид:

$$\begin{cases} \bar{x}^2 + \bar{y}^2 = {r_1}^2 \\ \bar{x}^2 + 2\bar{x}\Delta x + \Delta x^2 + \bar{y}^2 + 2\bar{y}\Delta y + \Delta y^2 = {r_2}^2 \end{cases}$$

После вычитания первого уравнения из второго, получаем линейную зависимость между $\bar{x}$ и $\bar{y}$:

$$\begin{cases} \bar{x}^2 + \bar{y}^2 = {r_1}^2 \\ 2\bar{x}\Delta x + \Delta x^2 + 2\bar{y}\Delta y + \Delta y^2 = {r_2}^2 - {r_1}^2 \end{cases}$$

Отсюда несложно получить квадратное уравнение для $\bar{x}$ или $\bar{y}$.

Пересечение прямой с окружностью выполняется аналогично – там линейная зависимость сразу задается принадлежностью к прямой.

После этого для проверки того, что точка лежит внутри круга достаточно проверить, что расстояние от точки до его центра меньше его радиуса.

Однако этот метод сопряжен с техническими трудностями, связанными с проблемами, которые возникают при разрешении линейной зависимости относительно одной из переменных – при этом возникает опасность деления на 0, что требует рассмотрения двух различных случаев. Имеется еще один способ проверки, что набор кругов с заданной точностью покрывает прямоугольник, правда существенно использующий факт наличия требуемой точности вычислений. Предположим, что мы хотим проверить, что некоторый прямоугольник покрывается заданным множеством кругов. Если все четыре его вершины покрываются одним кругом, то, очевидно, прямоугольник полностью покрывается кругами. В противном случае разобьем прямоугольник на четыре одинаковых прямоугольника и рекурсивно проверим, что они покрываются кругами. Если в процессе рекурсии сторона прямоугольника стала меньше некторой заданой величины, разумно подозреваем, что этот прямоугольник полностью кругами не покрывается (он, вероятно, лежит около точки пересечения кругов, но покрываемой другими).

Рекурсивная функция, выполняющая соответствующую проверку приведена ниже:

```
{ Рассотяние между двумя точками }
function dist(x1, y1, x2, y2: real): real;
begin
  dist := sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
end;

{ Параметры – координаты верхнего-левого и правого-нижнего углов
  прямоугольника }
function check(x1, y1, x2, y2: real): boolean;
var
  i: longint;
begin
  if (abs(x1 - x2) < 1e-5) and (abs(y1-y2) < 1e-5) then
  begin
    { Запомнимаем точку, которая не покрывется – например центр
      прямоугольника, поскольку заданая точность позволяет отклонение }
    fx := (x1 + x2) / 2;
    fy := (y1 + y2) / 2;

    check := false;
    exit;
  end;

{ Проверяем покрытие одним кругом }
  for i := 1 to n do
  begin
    { m – текущий радиус фонтана – «прибавка» к радиусам колонн }
    if (dist(x1, y1, x[i], y[i]) <= r[i] + m) and
       (dist(x1, y2, x[i], y[i]) <= r[i] + m) and
       (dist(x2, y1, x[i], y[i]) <= r[i] + m) and
       (dist(x2, y2, x[i], y[i]) <= r[i] + m) then
    begin
      check := true;
      exit;
    end;
  end;

  { Рекурсивно вызываемся }
  check := check(x1, y1, (x1 + x2) / 2, (y1 + y2) / 2) and
           check((x1 + x2) / 2, y1, x2, (y1 + y2) / 2) and
           check(x1, (y1 + y2) / 2, (x1 + x2) / 2, y2) and
           check((x1 + x2) / 2, (y1 + y2) / 2, x2, y2);
```

```
end;
```

Теперь искомый радиус фонтана можно найти двоичным поиском:

```
lb := 0; { Левая граница }
{ xr и yr – длина и ширина холла }
if (xr < yr) then rb := xr / 2 else rb := yr / 2; { Правая граница }

{ Дихотомия }
while abs(lb - rb) > 1e-4 do
begin
  m := (lb + rb) / 2;

  if check(m, m, xr - m, yr - m) then
  begin
    rb := m;
  end
  else
  begin
    lb := m;
  end;
end;

{ Выводим ответ }
writeln(fx :0 :3, ' ', fy :0 :3, ' ', m :0 :3);
```

Задача 24. Триангуляция. [Российские зимние сборы школьников 2006]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Вася нарисовал выпуклый N-угольник и провел в нем несколько диагоналей таким образом, что никакие две диагонали не пересекаются внутри N-угольника.

Теперь он утверждает, что весь N-угольник оказался разбит на треугольники.

Напишите программу, которая проверяет истинность Васиного утверждения.

Входные данные.

Во входном файле записано сначала число N — количество вершин N-угольника ($3 \le N \le 100\,000$). Далее записано число M — количество диагоналей, проведенных Васей. Далее записано M пар чисел, задающих диагонали (каждая диагональ задается парой номеров вершин, которые она соединяет). Гарантируется, что каждая пара чисел задает диагональ (то есть две вершины различны, и не являются соседними), а также что никакие две пары не задают одну и ту же диагональ. Никакие две диагонали не пересекаются внутри N-угольника.

Вершины N-угольника нумеруются числами от 1 до N.

Выходные данные.

Если Васино утверждение верно, то выходной файл должен содержать единственное число 0.

В противном случае быть выведено сначала число K — количество вершин в какой-нибудь не треугольной части. Далее должно быть выведено K чисел — номера вершин исходного N-угольника, которые являются вершинами этой K-угольной части в порядке обхода этой части.

Пример:

input.txt	output.txt
3 0	0
4 1 3 1	0
6 2 1 3 5 3	4 1 3 5 6

Комментарий. Короче говоря, нам дан планарный граф, т.е. такой граф, который можно изобразить на плоскости так, чтобы его ребра не пересекались нигде, кроме вершин. Многоугольники, на которые разбивает плоскость планарный граф, называются гранями графа. Одна из интересных задач, связанных с планарными графами – поиск всех его граней. Любой простой цикл, не содержащий в себе других циклов, будет гранью.

Будем обходить этот граф. Для каждой вершины упорядочим все ребра, инцидентные вершине, по полярному углу. Войдя в вершину выходить будем обязательно по следующему ребру в этом упорядочении. Как

только мы достигнем вершины, из которой мы вышли, мы нашли очередной цикл. Действуя таким образом, мы обойдем все грани графа. Также мы найдем и внешнюю грань, которую следует обрабатывать особо.

Программа. Приведем программный код, реализующий вышеописанный подход:

```
#include <stdio.h>
#define MAXN 400000
#define swap(a, b) (a ^= b, b ^= a, a ^= b) // обмен чисел

int getint() { // чтение числа из стандартного входа
	int v = 0;
	char c;
	do c = getchar(); while (c < '0' || c > '9');
	do {
		v = v * 10 + c - '0';
		c = getchar();
	} while (c >= '0' && c <= '9');
	return v;
}

// a – начальная вершина ребра
// b - конечная вершина ребра
// d – начало списка упорядоченных вершин
// r – обратное ребро
// f - метка
int a[MAXN], b[MAXN], d[MAXN], r[MAXN], f[MAXN];
int n0; // n0 == n
char u[MAXN];

int cmp(const int a1, const int b1, const int a2, const int b2) { // сравнивает точки a1, b1
	register int bb1, bb2;                                        // и a2, b2
	if (a1 != a2) return a1 - a2;
	bb1 = b1 - a1;
	while (bb1 < 0) bb1 += n0;
	while (bb1 >= n0) bb1 -= n0;
	bb2 = b2 - a2;
	while (bb2 < 0) bb2 += n0;
	while (bb2 >= n0) bb2 -= n0;
	return bb1 - bb2;
}

// сортируем точки
int partition(const int le, const int ri) {
	int i = le - 1, j = ri + 1, x = a[(le + ri) >> 1], y = b[(le + ri) >> 1];
	while (1) {
		do i++; while (cmp(a[i], b[i], x, y) < 0);
		do j--; while (cmp(a[j], b[j], x, y) > 0);
		if (i < j) {
			swap(a[i], a[j]);
			swap(b[i], b[j]);
			if (r[i] != j) {
				swap(r[i], r[j]);
				r[r[i]] = i, r[r[j]] = j;
			}
		} else
			return j;
	}
}

void sort(int l, int r) {
	while (l < r) {
		int m = partition(l, r);
		sort(l, m);
		l = m + 1;
	}
}

int main() {
	int n, i, c;
	// читаем количество точек и ребер и добавляем в граф последние
	n = getint();
	i = getint();
```

```
	c = 0;
	while (i--) {
		int x = getint() - 1, y = getint() - 1;
		a[c] = x, b[c] = y, r[c] = c++ + 1;
		a[c] = y, b[c] = x, r[c] = c++ - 1;
	}
	for (i = n - 2; i >= 0; i--) {
		a[c] = i, b[c] = i + 1, r[c] = c++ + 1;
		a[c] = i + 1, b[c] = i, r[c] = c++ - 1;
	}
	a[c] = 0, b[c] = n - 1, r[c] = c++ + 1;
	a[c] = n - 1, b[c] = 0, r[c] = c++ - 1;
	// сортируем ребра
	n0 = n;
	sort(0, c - 1);
	// инициализируем d
	i = 0;
	while (i < c) {
		int x = a[i];
		d[x] = i;
		do i++; while (i < c && a[i] == x);
	}
	// обход
	memset(f, ~0, sizeof(f));
	for (i = 0; i < n; i++) { // по всем вершинам
		int j;
		for (j = d[i]; j < c && a[j] == i; j++) // по всем инцидентным ребрам
		if (!u[j]) {
			int z = i, k = j, c0 = 0; // z – текущая вершина, k – текущее ребро
			static int q[MAXN];
			do {
				int x = r[k], y = b[k];
				q[c0++] = z;
				u[k] = 1;
				if (++x >= c || a[x] != y)
				x = d[y];
				z = y, k = x;
			} while (z != i); // пока не вернулись в i
			if (c0 > 3 && (c0 < n || c == (n << 1))) { // если количество вершин > 3
				printf("%d\n", c0);              // и < n или количество ребер
				for (k = 0; k < c0; k++)         // равно 2 * n, т.е. нет
					printf("%d ", q[k] + 1);  // диагоналей
				goto _done;
			}
		}
	}
	putchar('0');
	_done:
	return 0;
}
```

Задача 25. Seeing the Boundary. [IOI 2003]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**64 мегабайта**

Farmer Don watches the fence that surrounds his N meter by N meter square, flat field ($2 \le N \le 500{,}000$). One fence corner is at the origin (0, 0) and the opposite corner is at (N, N); the sides of Farmer Don's fence are parallel to the X and Y axes.

Fence posts appear not only at all four corners but also at every meter along each side of the fence, for a total of $4 \cdot N$ fence posts. The fence posts are vertical and are considered to have no radius. Farmer Don wants to determine how many of his fence posts he can watch when he stands at a given location within his fence.

Farmer Don's field contains R ($1 \le R \le 30{,}000$) huge rocks that obscure his view of some fence posts, as he is not tall enough to look over any of these rocks. The base of each rock is a convex polygon with nonzero area whose vertices are at integer coordinates. The rocks stand completely vertical. Rocks do not overlap, do not touch other rocks, and do not touch Farmer Don or the fence. Farmer Don does not touch the fence, does not stand within a rock, and does not stand on a rock.

Given the size of Farmer Don's fence, the locations and shapes of the rocks within it, and the location where Farmer Don stands, compute the number of fence posts that Farmer Don can see. If a vertex of a rock lines up perfectly with a fence post from Farmer Don's location, he is not able to see that fence post.

Входные данные

The first line of input contains two space-separated integers: N and R.

The next line of input contains two space-separated integers that specify the X and Y coordinates of Farmer Don's location inside the fence.

The rest of the input file describes the R rocks:

- Rock i's description starts with a line containing a single integer pi ($3 \le pi \le 20$), the number of vertices in the rock's base.
- Each of the next pi lines contains a space-separated pair of integers that are the X and Y coordinates of a vertex. The vertices of a rock's base are distinct and given in counterclockwise order.

Выходные данные

The output file should contain a single line with a single integer, the number of fence posts visible to Farmer Don.

Пример

input.txt	output.txt
100 1 60 50 5 70 40 75 40 80 40 80 50 70 60	319

Комментарий:

Основная идея — будем рассматривать камни как набор отрезков. Проведем из точки, в которой находится фермер лучи через обе вершины каждого отрезка и найдем точки их пересечения с забором. Таким образом для каждого отрезка можно определить интервал забора, который он закрывает от фермера. После этого объединяем пересекающиеся и касающиеся отрезки — это будет закрытая часть забора, и считаем длину оставшейся части.

Программа

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define max(a, b) (a > b ? a : b)
#define min(a, b) (a < b ? a : b)

const double eps = 1e-10;

// Информация о текущем камне
__int64 a[21][2];
// Интервалы забора, закрываемые отрезками
double b[1000000][2];
// Объединенные закрытые интервалы
double c[1000000][2];

int cmp(const void* a, const void* b) {
    if (*(double*)a > *(double*)b) return 1;
    if (*(double*)a < *(double*)b) return -1;
    return 0;
}

// Определяет, пересекаются ли луч и отрезок и, если да, возвращает точку пересечения
bool cross(__int64 x1, __int64 y1, __int64 x2, __int64 y2,
           __int64 x3, __int64 y3, __int64 x4, __int64 y4, double& x, double& y) {
    __int64 a1, b1, c1, a2, b2, c2, d;
    a1 = y2 - y1; b1 = x1 - x2; c1 = x1 * y2 - y1 * x2;
    a2 = y4 - y3; b2 = x3 - x4; c2 = x3 * y4 - y3 * x4;
    d = (a1 * b2 - a2 * b1);
    if (d == 0) return false;
    x = (double) (c1 * b2 - c2 * b1) / (double) d;
    y = (double) (a1 * c2 - a2 * c1) / (double) d;
    if ((x < x3 - eps) || (x > x4 + eps) || (y < y3 - eps) || (y > y4 + eps)) return false;
    if ((x2 < min(x1, x) - eps) || (x2 > max(x1, x) + eps) ||
```

```
        (y2 < min(y1, y) - eps) || (y2 > max(y1, y) + eps)) return false;
    return true;
}

// Проверяет, расположены ли отрезки против часовой стрелки
bool pos(__int64 x1, __int64 y1, __int64 x2, __int64 y2, __int64 x3, __int64 y3) {
    return (x2 - x1) * (y3 - y1) >= (x3 - x1) * (y2 - y1);
}

int main() {
    __int64 n, r, x, y, p;
    int bc, cc;
    double tx, ty;
    __int64 i, j, s, tt;
    bool ch;

    scanf("%I64d %I64d", &n, &r);
    scanf("%I64d %I64d", &x, &y);
    bc = 0;
    for (i = 0; i < r; i++) {
        scanf("%I64d", &p);
        for (j = 0; j < p; j++) scanf("%I64d %I64d", &a[j][0], &a[j][1]);
        a[p][0] = a[0][0]; a[p][1] = a[0][1];
        for (j = 1; j <= p; j++) {
            if (pos(x, y, a[j][0], a[j][1], a[j - 1][0], a[j - 1][1])) {
                ch = true;
                tt = a[j][0]; a[j][0] = a[j - 1][0]; a[j - 1][0] = tt;
                tt = a[j][1]; a[j][1] = a[j - 1][1]; a[j - 1][1] = tt;
            } else ch = false;
             // Определение точек пересечения
            if (cross(x, y, a[j - 1][0], a[j - 1][1], 0, 0, n, 0, tx, ty)) {
                b[bc][0] = tx;
                if (cross(x, y, a[j][0], a[j][1], 0, 0, n, 0, tx, ty)) b[bc][1] = tx;
                else if (cross(x, y, a[j][0], a[j][1], n, 0, n, n, tx, ty))
                    b[bc][1] = n + ty;
                else if (cross(x, y, a[j][0], a[j][1], 0, n, n, n, tx, ty))
                    b[bc][1] = 3.0 * n - tx;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, 0, n, tx, ty))
                    b[bc][1] = 4.0 * n - ty;
            } else if (cross(x, y, a[j - 1][0], a[j - 1][1], n, 0, n, n, tx, ty)) {
                b[bc][0] = n + ty;
                if (cross(x, y, a[j][0], a[j][1], n, 0, n, n, tx, ty)) b[bc][1] = n + ty;
                else if (cross(x, y, a[j][0], a[j][1], 0, n, n, n, tx, ty))
                    b[bc][1] = 3.0 * n - tx;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, 0, n, tx, ty))
                    b[bc][1] = 4.0 * n - ty;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, n, 0, tx, ty))
                    b[bc][1] = tx + 4.0 * n;
            } else if (cross(x, y, a[j - 1][0], a[j - 1][1], 0, n, n, n, tx, ty)) {
                b[bc][0] = 3.0 * n - tx;
                if (cross(x, y, a[j][0], a[j][1], 0, n, n, n, tx, ty))
                    b[bc][1] = 3.0 * n - tx;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, 0, n, tx, ty))
                    b[bc][1] = 4.0 * n - ty;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, n, 0, tx, ty))
                    b[bc][1] = tx + 4.0 * n;
                else if (cross(x, y, a[j][0], a[j][1], n, 0, n, n, tx, ty))
                    b[bc][1] = n + ty + 4.0 * n;
            } else if (cross(x, y, a[j - 1][0], a[j - 1][1], 0, 0, 0, n, tx, ty)) {
                b[bc][0] = 4.0 * n - ty;
                if (cross(x, y, a[j][0], a[j][1], 0, 0, 0, n, tx, ty))
                    b[bc][1] = 4.0 * n - ty;
                else if (cross(x, y, a[j][0], a[j][1], 0, 0, n, 0, tx, ty))
                    b[bc][1] = tx + 4.0 * n;
                else if (cross(x, y, a[j][0], a[j][1], n, 0, n, n, tx, ty))
                    b[bc][1] = n + ty + 4.0 * n;
                else if (cross(x, y, a[j][0], a[j][1], 0, n, n, n, tx, ty))
                    b[bc][1] = 3.0 * n - tx + 4.0 * n;
            }
            if (ch) {
                tt = a[j][0]; a[j][0] = a[j - 1][0]; a[j - 1][0] = tt;
                tt = a[j][1]; a[j][1] = a[j - 1][1]; a[j - 1][1] = tt;
            }
```

```
            if (b[bc][1] > 4.0 * n + eps) {
                b[bc + 1][0] = 0.0;
                b[bc + 1][1] = b[bc][1] - 4.0 * n;
                if (b[bc + 1][1] >= 4.0 * n + eps) {
                    printf("0\n");
                    return 0;
                }
                b[bc][1] = 4.0 * n;
                bc++;
            }
            bc++;
        }
    }
    qsort(b, bc, sizeof(b[0]), cmp);
       // Объединение интервалов
    c[0][0] = b[0][0]; c[0][1] = b[0][1];
    cc = 0;
    for (i = 1; i < bc; i++) {
        if (b[i][0] <= c[cc][1] + eps) {
            if (b[i][1] > c[cc][1]) c[cc][1] = b[i][1];
        }
        else {
            c[++cc][0] = b[i][0];
            c[cc][1] = b[i][1];
        }
    }
    if (c[cc][1] > 4 * n) c[cc][1] = (double) 4 * n;
    cc++;
       // Подсчет видимых кольшков
    s = n * 4;
    for (i = 0; i < cc; i++)
        s -= (__int64)(floor(c[i][1] + eps) - ceil(c[i][0] - eps)) + 1;
    if ((fabs(c[0][0]) < eps) && (fabs(4 * n - c[cc - 1][1]) < eps)) s++;
    printf("%I64d\n", s);
    return 0;
}
```

Задача 26. Gunman. [NEERC semifinal 2004]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Consider a 3D scene with OXYZ coordinate system. Axis OX points to the right, axis OY points up, and axis OZ points away from you. There is a number of rectangular windows on the scene. The plane of each window is parallel to OXY , its sides are parallel to OX and OY . All windows are situated at different depths on the scene (different coordinates $z > 0$).

A gunman with a rifle moves along OX axis ($y = 0$ and $z = 0$). He can shoot a bullet in a straight line. His goal is to shoot a single bullet through all the windows. Just touching a window edge is enough. Your task is to determine how to make such shot.

Входные данные

The first line of the input file contains a single integer number n ($2 \le n \le 100$) — the number of windows on the scene. The following n lines describe the windows. Each line contains five integer numbers $x1_i$, $y1_i$, $x2_i$, $y2_i$, z_i ($0 < x1_i, y1_i, x2_i, y2_i, z_i < 1000$). Here ($x1_i$, $y1_i$, z_i) are coordinates of the bottom left corner of the window, and ($x2_i$, $y2_i$, z_i) are coordinates of the top right corner of the window ($x1_i < x2_i$, $y1_i < y2_i$). Windows are ordered by z coordinate ($z_i > z_{i-1}$ for $2 \le i \le n$).

Выходные данные

Output a single word “UNSOLVABLE” if the gunman cannot reach the goal of shooting a bullet through all the windows. Otherwise, on the first line output a word “SOLUTION”. On the next line output x coordinate of the point from which the gunman must fire a bullet. On the following n lines output x, y, z coordinates of the points where the bullet goes through the consecutive windows. All coordinates in the output file must be printed with six digits after decimal point.

Пример

input.txt	output.txt
3 1 3 5 5 3 1 2 5 7 5	SOLUTION -1.000000 2.000000 3.000000 3.000000

5 2 7 6 6	4.000000 5.000000 5.000000 5.000000 6.000000 6.000000
3 2 1 5 4 1 3 5 6 8 2 4 3 8 6 4	UNSOLVABLE

Комментарий.

Переформулируем задачу: требуется найти прямую, проходящую через ось OX и все окна.

Используем два факта из начертательной геометрии:

1. Любая прямая и перпендикулярная плоскостям XOZ и YOZ плоскость имеют две проекции: по одной на каждую из этих плоскостей. При этом, если прямая пересекает плоскость (или вообще любой объект), то их проекции также должны пересекаться.

2. Если взять любую прямую на XOZ и прямую на YOZ, их можно рассматривать как проекции некоторой прямой, которую можно однозначно по ним восстановить.

Используя первый факт мы получаем две упрощенные задачи:

1. Найти прямую, начинающуюся в начале координат и проходящую через все отрезки.

2. Найти прямую проходящую через все отрезки.

Можно доказать, что если эти подзадачи имеют решение, то значит они имеют решение в качестве прямых, проходящих через концы каких-то отрезков. Таким образом, можно просто перебрать все возможные прямые, проходящие через концы отрезков и построить по ним как по проекциям искомую прямую. А используя второй факт, можно искать эти проекции независимо друг от друга.

Программа

```
#include <stdio.h>

// Информация об окне
struct {
    int x1, y1, x2, y2, z;
} p[100];

// Возвращает, пересекает ли прямая отрезок
bool cross(int ax, int ay, int bx, int by, int x1, int y1, int x2, int y2) {
    int a, b;
    a = (bx - ax) * (y1 - ay) - (by - ay) * (x1 - ax);
    b = (bx - ax) * (y2 - ay) - (by - ay) * (x2 - ax);
    return ((a <= 0) && (b >= 0)) || ((a >= 0) && (b <= 0));
}

int main() {
    int n, i, j, k, x1, z1, x2, z2, z, y;
    double k1, k2, b;

// Ввод
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d %d %d %d %d", &p[i].x1, &p[i].y1, &p[i].x2, &p[i].y2, &p[i].z);

// Поиск проекции на XOZ
    for (i = 0; i < n; i++) {
        x1 = p[i].x1; z1 = p[i].z;
        for (j = i + 1; j < n; j++) {
            x2 = p[j].x1; z2 = p[j].z;
            for (k = 0; k < n; k++) if ((k != i) && (k != j))
                if (!cross(x1, z1, x2, z2, p[k].x1, p[k].z, p[k].x2, p[k].z)) break;
            if (k == n) break;
            x2 = p[j].x2; z2 = p[j].z;
            for (k = 0; k < n; k++) if ((k != i) && (k != j))
                if (!cross(x1, z1, x2, z2, p[k].x1, p[k].z, p[k].x2, p[k].z)) break;
            if (k == n) break;
        }
        if (j < n) break;
        x1 = p[i].x2; z1 = p[i].z;
        for (j = i + 1; j < n; j++) {
            x2 = p[j].x1; z2 = p[j].z;
            for (k = 0; k < n; k++) if ((k != i) && (k != j))
                if (!cross(x1, z1, x2, z2, p[k].x1, p[k].z, p[k].x2, p[k].z)) break;
            if (k == n) break;
```

```
            x2 = p[j].x2; z2 = p[j].z;
            for (k = 0; k < n; k++) if ((k != i) && (k != j))
                if (!cross(x1, z1, x2, z2, p[k].x1, p[k].z, p[k].x2, p[k].z)) break;
            if (k == n) break;
        }
        if (j < n) break;
    }

// Если не нашли, то решения нет
    if (i == n) {
        printf("UNSOLVABLE\n");
        return 0;
    }

// Поиск проекции на YOZ
    for (i = 0; i < n; i++) {
        z = p[i].z; y = p[i].y1;
        for (j = 0; j < n; j++) if (j != i)
            if (!cross(0, 0, z, y, p[j].z, p[j].y1, p[j].z, p[j].y2)) break;
        if (j == n) break;
        z = p[i].z; y = p[i].y2;
        for (j = 0; j < n; j++) if (j != i)
            if (!cross(0, 0, z, y, p[j].z, p[j].y1, p[j].z, p[j].y2)) break;
        if (j == n) break;
    }

// Если не нашли, то решения нет
    if (i == n) {
        printf("UNSOLVABLE\n");
        return 0;
    }

// Вычисление точек пересечения и вывод
    k1 = (double) (x2 - x1) / (double) (z2 - z1);
    b = (double) (x1 * z2 - z1 * x2) / (double) (z2 - z1);
    k2 = (double) y / (double) z;
    printf("SOLUTION\n%.6lf\n", b);
    for (i = 0; i < n; i++)
        printf("%.6lf %.6lf %.6lf\n", k1 * p[i].z + b, k2 * p[i].z, (double) p[i].z);
    return 0;
}
```

Задача 27. Многоугольник. [ЛКШ 2005]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

На плоскости задан правильный N-угольник с центром в начале координат, а также вне него — точки A и B. Представим, что между точками A и B натянута нитка, которая не может пересечь границу многоугольника и потому огибает его.

Какова минимальная длина нитки ?

Входные данные

В первой строке входного файла записано число N ($3 \le N \le 30$). Во второй строке записаны координаты одной из вершин N-угольника. В третьей и четвертой строке записаны координаты точек A и B соответственно. Все координаты — действительные числа по абсолютной величине не превосходящие 1000.

Выходные данные

Выведите в выходной файл исокомую длину нитки с точностью двух знаков после десятичной точки.

Пример

input.txt	output.txt
30 3 4 0 -4 0	9.30

Комментарий

Можно доказать, что нитка может «отрываться» от многоугольника только в его вершинах. Причем вершин, в которых это может произойти максимум 4. Это такие вершины, что прямая, проходящая через них и точку A или B не пересекает многоугольник (такие прямые называются опорными). Найти их можно, выбирая самую левую или самую правую вершину относительно точки A или B (отсюда и 4 варианта). После определения этих вершин, можно простым перебором определить те две точки, проходя через которые, нитка будет иметь наименьшую длину.

Программа

```
#include <stdio.h>
#include <math.h>

// Вершины многоугольника
struct {
  double x, y;
} p[100];
int n;
// Длина стороны многоугольника
double w;

// Расстояние между двумя точками на плоскости
double dist(double x1, double y1, double x2, double y2) {
  return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

// Расстояние между двумя вершинами многоугольника по его границе
double dist2(int a, int b) {
  int c;
  c = labs(a - b);
  if (n - c < c) c = n - c;
  return c * w;
}

double min(double a, double b) {
  return a < b ? a : b;
}

double max(double a, double b) {
  return a > b ? a : b;
}

// Определяет, пересекаются ли отрезки
bool _cross(double x1, double y1, double x2, double y2, double x3, double y3, double x4,
double y4) {
  double a, b;
  if ((min(x1, x2) > max(x3, x4)) || (min(x3, x4) > max(x1, x2)) ||
      (min(y1, y2) > max(y3, y4)) || (min(y3, y4) > max(y1, y2))) return false;
  a = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
  b = (x2 - x1) * (y4 - y1) - (y2 - y1) * (x4 - x1);
  if (a * b > 0) return false;
  a = (x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3);
  b = (x4 - x3) * (y2 - y3) - (y4 - y3) * (x2 - x3);
  if (a * b > 0) return false;
  return true;
}

// Определяет, пересекает ли отрезок многоугольник
bool cross(double x1, double y1, double x2, double y2) {
  int i;
  for (i = 0; i < n; i++)
    if (_cross(x1, y1, x2, y2, p[i].x, p[i].y, p[i + 1].x, p[i + 1].y)) return true;
  return false;
}

// Поиск левой опорной прямой из точки x, y
int findl(double x, double y) {
  int a, i;
  a = 0;
  for (i = 1; i < n; i++)
    if ((p[a].x - x) * (p[i].y - y) > (p[a].y - y) * (p[i].x - x)) a = i;
  return a;
```

```
}

// Поиск правой опорной прямой из точки x, y
int findr(double x, double y) {
  int a, i;
  a = 0;
  for (i = 1; i < n; i++)
    if ((p[a].x - x) * (p[i].y - y) < (p[a].y - y) * (p[i].x - x)) a = i;
  return a;
}

int main() {
  int i, a, b, c, d;
  double x, y, x1, y1, x2, y2, t, pi, r, dt, s;

// Ввод
  pi = acos(-1.0);
  scanf("%d %lf %lf %lf %lf %lf %lf", &n, &x, &y, &x1, &y1, &x2, &y2);

// Построение многоугольника
  r = sqrt(x * x + y * y);
  dt = 2.0 * pi / n;
  w = 2 * r * sin(dt / 2.0);
  p[0].x = x; p[0].y = y;
  if (x == 0) {
    if (y > 0) t = pi / 2; else t = -pi / 2;
  } else t = atan(y / x);
  for (i = 1; i < n; i++) {
    t += dt;
    p[i].x = r * cos(t); p[i].y = r * sin(t);
  }
  p[n] = p[0];
// Если можно протянуть нитку напрямую
  if (!cross(x1, y1, x2, y2)) {
    printf("%.2lf\n", dist(x1, y1, x2, y2));
    return 0;
  }
// Поиск опорных точек
  a = findl(x1, y1); b = findr(x1, y1);
  c = findl(x2, y2); d = findr(x2, y2);
// Определение минимальной длины нитки
  s = dist(x1, y1, p[a].x, p[a].y) + dist(p[c].x, p[c].y, x2, y2) + dist2(a, c);
  t = dist(x1, y1, p[a].x, p[a].y) + dist(p[d].x, p[d].y, x2, y2) + dist2(a, d);
  if (t < s) s = t;
  t = dist(x1, y1, p[b].x, p[b].y) + dist(p[c].x, p[c].y, x2, y2) + dist2(b, c);
  if (t < s) s = t;
  t = dist(x1, y1, p[b].x, p[b].y) + dist(p[d].x, p[d].y, x2, y2) + dist2(b, d);
  if (t < s) s = t;
// Вывод
  printf("%.2lf\n", s);
  return 0;
}
```

ГЛАВА 6. ГРАФЫ

Задача 28. Лабиринт знаний.

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**3 секунды**
Ограничение по памяти	**16 мегабайт**

В ЛКШ построили аттракцион «Лабиринт знаний». Лабиринт представляет собой N комнат, занумерованных от 1 до N, между некоторыми из которых есть двери. Когда человек проходит через дверь, показатель его знаний изменяется на определенную величину, фиксированную для данной двери. Вход в лабиринт находится в комнате 1, выход – в комнате N. Каждый ЛКШонок проходит лабиринт ровно один раз и попадает в группу в зависимости от набранных знаний (при входе в лабиринт этот показатель равен нулю). Ваша задача показать наилучший результат.

Входные данные:

Первая строка входного файла содержит целые числа N ($1 \le N \le 2000$) – количество комнат и M ($1 \le M \le 10000$) – количество дверей. В каждой из следующих M строк содержится описание двери – номера комнат, из которой она ведет и в которую она ведет, а также целое число, которое прибавляется к количеству знаний при прохождении через дверь (это число по модулю не превышает 10000). Двери могут вести из комнаты в нее саму, между двумя комнатами может быть более одной двери.

Выходные данные:

В выходной файл выведите «:)» - если можно получить неограниченно большой запас знаний, «:(» - если лабиринт пройти нельзя, и максимальное количество набранных знаний в противном случае.

Пример:

input.txt	output.txt
2 2 1 2 5 1 2 -5	5

Комментарий. Для начала формализуем задачу: лабиринт просто-напросто представляет собой граф с N вершинами и M ребрами. Каждое ребро имеет вес – это то количество знаний, которое получает человек, проходя через какую либо дверь (положительное в случае увеличения, отрицательное в случае уменьшения). Мы ищем в этом графе путь наибольшего веса из вершины 1 к вершине N.

Чтобы свести эту задачу к классической (поиск пути наименьшего веса), заменим все ребер на противоположные значения. Теперь применим алгоритм Форда-Беллмана для поиска пути наименьшего веса к полученному графу [12]. Если расстояние от вершины N равно ∞, то пути от вершины 1 до вершины N нет, и мы можем смело выводить «:(». В противном случае, когда путь до вершины N существует, нам нужно проверить нет ли какого-либо отрицательного цикла на пути от 1 к N. Сделать это можно следующим образом: необходимо попробовать выполнить еще одну релаксацию по всем ребрам; все вершины, расстояние до которых уменьшилось, принадлежат какому-либо отрицательному циклу. Поэтому нам теперь остается найти множество вершин, достижимых из вышеупомянутых. Если N окажется среди таковых, то это означает, что на пути из 1 к N мы можем зайти в этот цикл и набрать в нем бесконечное количество знаний, и можно выводить «:)». В противном случае мы просто выводим расстояние до вершины N, взятое с противоположным знаком.

Время работы алгоритма определяется врменем работы алгоритма Форда-Беллмана – O(N * M).

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXM 10000
#define MAXN 2000
#define INF -0xFFFFFFFFFFFFFFF // бесконечность

typedef struct {
	int u, v, c;
} edge; // ребро (u – начало, v - конец, c - вес)

typedef __int64 i64; // очевидно, придется использовать int64

edge e[MAXM];
i64 d[MAXN], old[MAXN]; // расстояния

int main() {
	int n, m, i;
```

```
	// читаем ввод
	scanf("%d %d", &n, &m);
	for (i = 0; i < m; i++) {
		scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].c);
		e[i].u--; e[i].v--;
	}
	for (i = 1; i < n; i++) // инициализируем расстояния(заметьте, что d[0] = 0)
		d[i] = INF;
	for (i = n - 1; i; i--) { // алгоритм Форда-Беллмана
		int j;			// n - 1 раз
		i64 t;
		for (j = 0; j < m; j++) // по всем ребрам
			if (d[e[j].u] > INF && (t = d[e[j].u] + e[j].c) > d[e[j].v])
				d[e[j].v] = t;
	}
	memcpy(old, d, sizeof(d)); // сохраняем полученные значения расстояний
	for (i = 0; i < m; i++) { // еще одна "внешняя" итерация алгоритма Форда-Беллмана
		i64 t;
		if (d[e[i].u] > INF && (t = d[e[i].u] + e[i].c) > d[e[i].v])
			d[e[i].v] = t;
	}
	if (d[n - 1] == INF) // выход недосягаем
		puts(":(");
	else {
		// здесь мы находим множество вершин, достижимых из всех вершин i, для которых
		// old[i] < d[i]. Это можно сделать с помощью стандартного метода обхода графов
		// если вершина n - 1 принадлежит этому множеству, выводим ":)",
		// в противном случае d[n - 1]
}
	return 0;
}
```

Задача 29. Кубики. [Всероссийскаяч командная олимпиада'2000]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Родители подарили Пете набор детских кубиков. Поскольку Петя скоро пойдет в школу, они купили ему кубики с буквами. На каждой из шести граней каждого кубика написана буква.

Теперь Петя хочет похвастаться перед старшей сестрой, что научился читать. Для этого он хочет сложить из кубиков ее имя. Но это оказалось довольно сложно сделать – ведь разные буквы могут находиться на одном и том же кубике и тогда Петя не сможет использовать обе буквы в слове. Правда одна и та же буква может встречаться на разных кубиках. Помогите Пете!

Дан набор кубиков и имя сестры. Выясните, можно ли выложить ее имя с помощью этих кубиков и если да, то в каком порядке следует выложить кубики.

Входные данные:

На первой строке входного файла находится число N ($1 \le N \le 100$) - количество кубиков в наборе у Пети. На второй строке записано имя Петиной сестры – слово, состоящие только из больших латинских букв, не длиннее 100 символов. Следующие N строк содержат по 6 букв (только большие латинские буквы), которые написаны на соответствующем кубике.

Выходные данные:

На первой строке выходного файла выведите `YES` если выложить имя Петиной сестры данными кубиками можно, `NO` в противном случае.

Если ответ `YES`, на второй строке выведите M различных чисел из диапазона $1..N$, где M - количество букв в имени Петиной сестры. i-е число должно быть номером кубика, который следует положить на i-е место при составлении имени Петиной сестры. Кубики нумеруются с 1, в том порядке, в котором они заданы во входном файле. Если решений несколько, выведите любое. Разделяйте числа пробелами.

Пример:

input.txt	output.txt
4 ANN ANNNNN BCDEFG HIJKLM	NO

NOPQRS	
5 HELEN ABCDEF GHIJKL MNOPQL STUVWN EIUOZK	YES 2 1 3 5 4

Комментарий. Эта задача сводится к нахождению максимального паросочетания в двудольном графе. Назовем кубики и буквы имени сестры вершинами нашего графа и соединим кубик ребром с буквой, если эта буква написана на этом кубике. Заметим что граф действительно двудольный, а выбор кубиков для выкладывания имени эквивалентен построению паросочетания. Проскольку количество ребер в паросочетании не превышает количества вершин в меньшей доле, то искомое паросочетание действительно максимально.

Для нахождения максимального паросочетания разработаны стандартные алгоритмы. Наиболее популярный алгоритм, использующий удлиняющие чередующиеся цепи описан, например в книге *Ахо А.В., Хопкрофт Дж.Э., Ульман Дж.Д. «Структуры данных и алгоритмы»*, нахождение максимального паросочетания через поиск максимального потока в сети с помощью алгоритма Форда-Фалкерсона описано, например, в [12], теоретическое исследование проблемы максимального паросочетания можно найти, например, в книге [27].

Задача 30. Яблоко от яблони... [Всероссийскаяч командная олимпиада'2001]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 1 секунда
Ограничение по памяти 16 мегабайт

У Пети в саду растет яблоня. Воодушевленный историей об Исааке Ньютоне, который, как известно, открыл закон всемирного тяготения после того, как ему на голову упало яблоко, Петя с целью повысить свою успеваемость по физике часто сидит под яблоней.

Однако, поскольку по физике у Пети твердая тройка, яблоки с его яблони падают следующим образом. В какой-то момент одно из яблок отрывается от ветки, на которой оно висит, и начинает падать строго вниз. Если в некоторый момент оно задевает другое яблоко, то то тоже отрывается от своей ветки и начинает падать вниз, при этом первое яблоко не меняет направление своего падения. Вообще, если любое падающее яблоко заденет другое яблоко на своем пути, то оно также начнет падать.

Таким образом, в любой момент каждое яблоко либо висит на ветке, либо падает строго вниз, причем все яблоки кроме первого, чтобы начать падать, должны сначала соприкоснуться с каким-либо другим падающим яблоком.

Выясните, какие яблоки упадут с Петиной яблони.

Входные данные:

Первая строка входного файла содержит N - количество яблок на петиной яблоне ($1 \le N \le 200$). Следующие N строк содержат описания яблок. Будем считать все яблоки шарами. Каждое яблоко задается координатами своей самой верхней точки (той, где оно исходно прикреплено к дереву, длиной черенка пренебрежем) x_i, y_i и z_i и радиусом r_i ($-10000 \le x_i, y_i, z_i \le 10000$, $1 \le r_i \le 10000$, все числа целые). Гарантируется, что изначально никакие яблоки не пересекаются (даже не соприкасаются). Ось OZ направлена вверх.

Выходные данные:

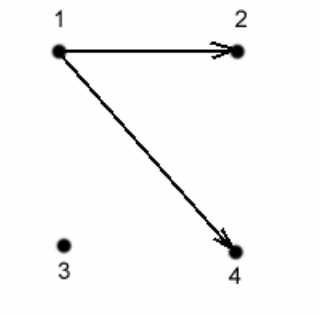
Выведите на первой строке выходного файла количество яблок, которые упадут с яблони, если начнет падать первое яблоко. На следующей строке выведите номера упавших яблок. Яблоки нумеруются, начиная с 1, в том порядке, в котором они заданы во входном файле.

Пример:

input.txt	output.txt
4 0 0 10 4 5 0 3 1 -7 4 7 1 0 1 2 6	3 1 2 4

Комментарий. Эта задача эквивалентна задаче о нахождении достижимого множества вершин в ориентированном графе. Покажем, как наша задача сводится к ней.

Рассмотрим граф, в котором вершинами будут яблоки, из i-й вершины в j-ю имеется ребро, если i-е яблоко задевает j-е при своем падении. Тогда множество вершин, достижимых из первой и будет множеством упавших яблок. Граф, соответствующий примеру приведен на рисунке.



- $Z_i > Z_j$
- проекции яблок на плоскость, параллельную OXY,
  пересекаются, чтобы проверить, что два круга пересекаются, достаточно проверить, что
  $\sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2} \le R_i + R_j$.

Приведем фрагмент программы, который вводит данные и преобразует из в матрицу смежности графа:

```
read(n); // читаем ввод
for i := 1 to n do
begin
    read(x[i], y[i], z[i], r[i]);
    z[i] := z[i] - r[i];
end;

for j := 1 to n do
    for k := 1 to n do
    begin
        if (sqr(x[j] - x[k]) + sqr(y[j] - y[k]) <=
                sqr(r[j] + r[k])) and (z[k] < z[j]) then
            a[j][k] := true;
    end;
```

Задача 31. Don't go left. [Andrew Stankevich contest 5]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**16 мегабайт**

Turing Machine (TM) is a well known abstraction, used in theoretical computer science. TM has a tape alphabet Σ and a finite set U of states. TM has an access to the tape, infinite in both directions. Tape is accessed with the head that can move along it, read and write characters to it. The tape is initially filled with blanks (B from Σ) and contains the input word ω from $(\Sigma \setminus \{B\})^*$.

The transition function of TM is the function $\varphi : U \times \Sigma \to U \times \Sigma \times \{\leftarrow, \rightarrow, \downarrow\}$. TM acts in the following way. Let its states be numbered from 1 to u where u = |U|. Let 1 be the initial and u be the terminal state. Initially TM is in the initial state and its head points to the first character of the input word ω.

Let TM be in a state q and its head point to the character c. If q is the terminal state, it stops and is said to accept the word ω. Let q be non-terminal state. If $\varphi(q, c) = \langle r, d, \alpha \rangle$, TM changes its state to r, writes d on the input tape instead of c and moves its head in the direction α. That is, if α is ←, it moves its head to the previous character of the tape, if it is →, it moves its head to the next character and if it is ↓, it does not move its head.

TM is called consecutive if whatever the input word ω is, it never moves its head to the left, that is, it never moves it to the previous character on the tape.

In this problem you have to determine whether the given TM is consecutive.

Входные данные:

The first line of the input file contains u — the number of states of the given TM and s = |Σ| — the number of characters in its tape alphabet ($2 \le u \le 100$, $2 \le s \le 100$). Denote the characters from Σ by $c_1, c_2, \ldots, c_s$. We will consider that $B = c_s$ that is — blank is the last character of the alphabet.

Next $(u - 1) \cdot s$ lines describe φ. The line of the input file with the number $1 + (i - 1) \cdot s + j$ describes $\varphi(i, c_j)$. Each line contains three integer numbers: k ($1 \le k \le u$) — the new state of TM, l ($1 \le l \le s$) — the character c_l is written on the tape and α ($-1 \le \alpha \le 1$) — the direction of the move (−1 stands for ←, 0 for ↓, and 1 for →). The last state is accepting, so the transitions from it are of no matter, therefore they are not described.

Remember, that the tape is initially filled with blanks and blank never occurs inside the input word. However, the input word may be empty, this is the only case when the head of TM initially points to a blank.

Выходные данные:

Print "YES" if the TM given is consecutive and "NO" if it is not.

Пример:

input.txt	output.txt
4 3 1 1 1 2 2 0 3 3 1 2 1 -1 2 2 0 2 3 0 3 1 1 4 3 1 2 3 0	YES

Комментарий. Эта задача сводится к нахождению обходу в ширину на графе. Граф строится следующим образом: вершина графа представляет собой тройку (state, symbol, outsideWord), где state – состояние МТ, symbol – текущий символ на ленте, на который указывает головка, outsideWord – признак того, что головка находится вне слова. Нам необходимо найти множество вершин, достижимых из вершин вида (1, x, 0), где x может быть любым символом кроме пробела, и (1, ‘ ‘, 1), которая соответствует пустому слову.

Программа.

```
#include <stdio.h>
#define MAXN 102
#define S 7
#define M 127
#define pack(x, y, z) (((x) << (S + 1)) | ((y) << 1) | (z))
#define getx(v) ((v) >> (S + 1))
#define gety(v) (((v) >> 1) & M)
#define getz(v) ((v) & 1)

int getint() { // чтение чисел
	int v = 0;
	char c, s;
	do c = getchar(); while (c != '-' && (c < '0' || c > '9'));
	if (c == '-') {
		do c = getchar(); while (c < '0' || c > '9');
		s = 1;
	} else
		s = 0;
	do {
		v = v * 10 + c - '0';
		c = getchar();
	} while (c >= '0' && c <= '9');
	return s ? -v : v;
}

int as[MAXN][MAXN], ac[MAXN][MAXN], dx[MAXN][MAXN], q[(MAXN * MAXN) << 2];
char u[MAXN][MAXN][2];

int main() {
	int n, a, i, h, t;
	char ok;
	// читаем входные данные
	n = getint(); a = getint();
	for (i = 0; i + 1 < n; i++) {
		int j;
		for (j = 0; j < a; j++)
			as[i][j] = getint() - 1, ac[i][j] = getint() - 1, dx[i][j] = getint();
	}
	// инициализируем обход в ширину
	q[0] = pack(0, a - 1, 1); u[0][a - 1][1] = 1;
	for (t = 1, h = 0, i = a - 2; i >= 0; i--) {
		q[t++] = pack(0, i, 0); u[0][i][0] = 1;
	}
	ok = 1;
	while (h < t) {
		int s = getx(q[h]), w = gety(q[h]), f = getz(q[h]), ss; h++;
		if (s + 1 == n) continue; // это конечное состояние
		ss = as[s][w];
		switch (dx[s][w]) {
```

```
                        case -1:
                               ok = 0; // достигли ←
                               goto _done; // выходим
                        case  0:
                               if (!u[ss][ac[s][w]][f]) {
                                      q[t++] = pack(ss, ac[s][w], f);
                                      u[ss][ac[s][w]][f] = 1;
                               }
                               break;
                        case  1:
                               if (!f) {// внутри слова
                                      for (i = a - 2; i >= 0; i--) // перебираем все
                                             if (!u[ss][i][0]) {  // возможные символы,
                                                    q[t++] = pack(ss, i, 0); // которые
                                                    u[ss][i][0] = 1; // могут быть справа
                                             }
                               }
                               if (!u[ss][a - 1][1]) { // если это пробел
                                      q[t++] = pack(ss, a - 1, 1);
                                      u[ss][a - 1][1] = 1;
                               }
                               break;
                }
        }
_done:
        puts(ok ? "YES" : "NO");
        return 0;
}
```

Задача 32. Агенты [ЛКШ 2004].

Входной файл	**age.in**
Выходной файл	**age.out**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**16 мегабайт**

После нескольких неудач со своими агентами, Центральное Интеллектуальное Агенство Байтлэнда решило улучшить свою деятельность. Обычно самой большой проблемой была подготовка безопасных встреч агентов. Ваша программа поможет в решении этой проблемы. Для данного описания сети дорог Байтлэнда и начальных позиций двух агентов, она должна отвечать, возможна ли их безопасная встреча.

Чтобы встреча была безопасной, агенты должны выполнять следующие предписания:

- агенты двигаются в течение дня и их встречи происходят вечером;
- агент должен менять свое местоположение каждый день;
- агенты могут двигаться только по дорогам, соединяющим города (к сожалению еще одна проблема заключается в том, что в Байтлэнде односторонние дороги);
- агент не может двигаться слишком быстро (это может выглядеть подозрительно) - за один день он может двигаться не далее, чем в соседний город;
- расстояние между двумя городами, соединенными дорогами, так мало, что агент, вышедший из одного города утром, успеет придти во второй до наступления вечера;
- говорят, что встреча состоялась, если два агента оказались в одном и том же городе в один и тот же вечер.

Напишите программу, которая:

- читает количество городов, описание сети дорог и начальные позиции агентов из текстового файла age.in;
- проверяет, возможна ли безопасная встреча, и, если это так, то сколько дней необходимо для ее организации;
- пишет результат в текстовый файл age.out.

Входные данные:

В первой строке текстового файла age.in находятся два целых числа n и m, разделенные пробелом, причем 1 <= n <= 250, 0 <= m <= n(n-1).

Во второй строке находятся два целых числа a1 и a2, разделенные одним пробелом, 1 <= a1, a2 <= n, a1 <> a2, обозначающие соответственно начальные позиции первого и второго агентов.

В следующих m строках находятся пары натуральных чисел a и b, разделенные пробелом, 1 <= a, b <= n, a <> b, обозначающие наличие дороги из города a в город b.

Выходные данные:

В текстовом файле age.out должна быть ровно одна строка, содержащая:

- ровно одно положительное целое число, равное минимальному времени (в днях), необходимому для организации безопасной встречи двух агентов - если такая встреча возможна;
- слово NIE - если такая встреча невозможна.

Пример:

age.in	age.out
6 7 1 5 1 2 4 5 2 3 3 4 4 1 5 4 5 6	3

Комментарий. Эта задача решается с помощью обхода в ширину графа, вершины которого являются парами (x, y), где x соответствует положению первого агента, y – положению второго. Но в данном случае для перехода из одного состояния в другие нам потребуется время порядка $O(n^2)$ в худшем случае. Поэтому поступим следующим образом: будем сначала двигать первого агента, а затем второго. То есть время перехода из одного состояния в другие будет равно O(n) в худшем случае.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 250
#define MAXM (MAXN * MAXN)

typedef struct {          // x – положение первого агента, y – положение второго агента
      int x, y, z, d;     // z = 0 соответствует состоянию, когда оба агента сделали ход
} vert;                   // d – расстояние до данной вершины

typedef struct {
      int v, n;
} edge;

int h[MAXN]; // заголовки списков смежности вершин
vert q[MAXM << 1]; // очередь
char u[MAXN][MAXN][2]; // метки
edge e[MAXM];// списки смежности вершин

int main() {
      int n, m, i;
      int hh, tt;
      freopen("age.in", "r", stdin);
      freopen("age.out", "w", stdout);
      // читаем входные данные
      scanf("%d %d %d %d", &n, &m, &hh, &tt); hh--; tt--;
      memset(h, ~0, sizeof(h));
      for (i = 0; i < m; i++) {
            int a, b;
            scanf("%d %d", &a, &b); a--; b--;
            e[i].v = b; e[i].n = h[a]; h[a] = i;
      }
      u[hh][tt][0] = 1;
      q[0].x = hh; q[0].y = tt; q[0].d = 0;
      hh = tt = 0;
      // обход в ширину
      while (hh <= tt) {
            int x = q[hh].x, y = q[hh].y = q[hh].y, dd = q[hh].d, z = q[hh].z;
            if (x == y && z == 0) { // если к вечеру оба в одном месте
                  printf("%d\n", q[hh].d);
                  return 0;
            }
            hh++;
            if (!z) { // если должен ходить первый
                  for (i = h[x]; i != -1; i = e[i].n) // перебираем вершины, смежные x
                        if (!u[e[i].v][y][1]) {
                              u[e[i].v][y][1] = 1;
                              q[++tt].x = e[i].v;
                              q[tt].y = y;
                              q[tt].d = dd;
                              q[tt].z = 1;
```

```
                        }
                } else { // если второй
                        dd++;
                        for (i = h[y]; i != -1; i = e[i].n) // перебираем вершины, смежные y
                                if (!u[x][e[i].v][0]) {
                                        u[x][e[i].v][0] = 1;
                                        q[++tt].d = dd;
                                        q[tt].x = x;
                                        q[tt].y = e[i].v;
                                        q[tt].z = 0;
                                }
                }
        }
        puts("NIE");
        return 0;
}
```

Задача 33. День рождения [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**16 мегабайт**

Митя знаком с M юношами и N девушками и хочет пригласить часть из них на свой день рождения. Ему известно, с какими девушками знаком каждый юноша, и с какими юношами знакома каждая девушка. Он хочет добиться того, чтобы каждый приглашённый был знаком со всеми приглашёнными противоположного пола, пригласив при этом максимально возможное число своих знакомых. Помогите ему это сделать!

Входные данные:

Входной файл состоит из одного или нескольких наборов входных данных. В первой строке входного файла записано число наборов K ($1 \le K \le 20$). В последующих строках записаны сами наборы входных данных.

В первой строке каждого набора задаются числа $0 \le M \le 150$ и $0 \le N \le 150$. Далее следуют M строк, в каждой из которых записано одно или несколько чисел - номера девушек, с которыми знаком i-й юноша (каждый номер встречается не более одного раза). Строка завершается числом 0.

Выходные данные:

Для каждого набора выведите четыре строки. В первой из них выведите максимальное число знакомых, которых сможет пригласить Митя. В следующей строке выведите количество юношей и количество девушек в максимальном наборе знакомых, разделённые одним пробелом. Следующие две строки должны содержать номера приглашённых юношей и приглашённых девушек соответственно. Числа в каждой из этих двух строк разделяются ровно одним пробелом и выводятся
в порядке возрастания. Если максимальных наборов несколько, то выведите любой из них.

Разделяйте вывод для разных наборов входных данных одной пустой строкой.

Пример:

input.txt	output.txt
2 2 2 1 2 0 1 2 0 3 2 1 2 0 2 0 1 2 0	4 2 2 1 2 1 2 4 2 2 1 3 1 2

Комментарий. Эта задача решается с помощью построения максимального независимого множества на дополнении к исходному двудольному графу. Этот граф, очевидно, также будет двудольным. Максимальное независимое множество строится с помощью максимального паросочетания в двудольном графе.

Можно доказать, что оно состоит из всех достижимых при обходе графа в ширину или в глубину вершин левой доли и недостижимых вершин правой доли [The 7th Baltic Olympiad in Informatics BOI 2001, Warsaw, 2001].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 152
```

```
char g[MAXN][MAXN]; // матрица смежности графа
char ux[MAXN],uy[MAXN]; // метки правой и левой долей
int mxy[MAXN], myx[MAXN], q[MAXN << 1]; // mxy – пары вершин левой доли
                                        // myx – пары вершин правой доли
                                        // q - очередь
int search(int v, int n) { // поиск в глубину
	int i;
	if (ux[v]) return 0;
	ux[v] = 1;
	for (i = 0; i < n; i++)
		if (g[v][i] && !uy[i] && myx[i] != v) {
			uy[i] = 1;
			if (myx[i] == -1 || search(myx[i], n)) {
				mxy[v] = i;
				myx[i] = v;
				return 1;
			}
		}
	return 0;
}

int main() {
	int tc;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	scanf("%d", &tc);
	while (tc--) {
		int n, m, i, cx, cy;
		scanf("%d %d", &m, &n);
		memset(g, 1, sizeof(g)); // изначально полный двудольный граф
		for (i = 0; i < m; i++) {
			int v;
			scanf("%d", &v);
			while (v--) {
				g[i][v] = 0; // исключаем ребро (i, v)
				scanf("%d", &v);
			}
		}
		// строим максимальное паросочетание
		memset(mxy, ~0, sizeof(mxy));
		memset(myx, ~0, sizeof(myx));
		for (i = 0; i < m; i++) {
			memset(ux, 0, sizeof(ux));
			memset(uy, 0, sizeof(uy));
			search(i, n);
		}
		// обходим в ширину получившийся граф
		memset(ux, 0, sizeof(ux));
		memset(uy, 0, sizeof(uy));
		for (i = 0; i < m; i++)
			if (mxy[i] == -1 && !ux[i]) {
				int h, t;
				ux[q[h = t = 0] = i] = 1;
				while (h <= t) {
					int v = q[h++], j;
					for (j = 0; j < n; j++)
						if (g[v][j] && !uy[j]) {
							uy[j] = 1;
							ux[q[++t] = myx[j]] = 1;
						}
				}
			}
		// считаем посещенные вершины левой доли
		for (cx = i = 0; i < m; i++)
			if (ux[i]) cx++;
		// считаем посещенные вершины правой доли
		for (cy = i = 0; i < n; i++)
			if (!uy[i]) cy++;
		printf("%d\n%d %d\n", cx + cy, cx, cy);
		// выводим посещенные вершины левой доли
		for (i = 0; i < m; i++)
			if (ux[i]) printf("%d ", i + 1);
```

```
            putchar('\n');
            // выводим посещенные вершины правой доли
            for (i = 0; i < n; i++)
                  if (!uy[i]) printf("%d ", i + 1);
            putchar('\n');
            putchar('\n');
      }
      return 0;
}
```

Задача 34. Мосты и точки сочленения [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Задан связный неориентированный граф G. В графе возможно наличие нескольких ребер между одной и той же парой вершин. Найдите все мосты и все точки сочленения в графе G.

Входные данные:

Первая строка входного файла содержит целое число N ($1 \le N \le 10000$) – количество вершин графа. Вторая строка входного файла содержит целое число M ($1 \le M \le 100000$) – количество ребер графа. В каждой из следующих M строк содержатся ровно два числа a и b ($1 \le a, b \le N$). Эти числа описывают ребро, соединяющее вершины с номерами a и b. Вершины нумеруются последовательными натуральными числами от 1 до N.

Выходные данные:

Первая строка выходного файла должна содержать число X - количество мостов в графе G. Следующие X строк должны содержать по одному числу – номеру ребра, являющегося мостом. Ребра нумеруются последовательными натуральными числами от 1 до M в порядке их появления во входном файле. Следующая строка выходного файла должна содержать число Y – количество точек сочленения в графе G. Следующие Y строк должны содержать по одному числу – номеру вершины, являющейся точкой сочленения. Выводите номера мостов и точек сочленения в порядке возрастания.

Пример:

input.txt	output.txt
6 7 1 2 2 3 1 3 3 4 4 5 5 6 4 6	1 4 2 3 4

Комментарий. Классическая задача поиска на графе. Приведем здесь детальный разбор решения этой задачи [Поиск в глубину и его применение, А. Лахно, Москва]:

Мосты

Мостом неориентированного графа *G* называется ребро, при удалении которого увеличивается количество компонент связности графа. Соответственно для связного графа мостом называется ребро, при удалении которого граф перестает быть связным.

При удалении всех мостов граф распадается на компоненты связности, которые называются *компонентами реберной двусвязности*.

Между любыми двумя вершинами одной компоненты реберной двусвязности существуют, по крайней мере, два пути, не пересекающиеся по ребрам. Верно и обратное утверждение: любые две вершины, между которыми существуют два пути, не пересекающиеся по ребрам, принадлежат одной компоненте реберной двусвязности.

Между двумя компонентами реберной двусвязности не может быть более одного ребра — в противном случае они образовывали бы одну компоненту реберной двусвязности. Если две различные компоненты реберной двусвязности соединены ребром, то это ребро — мост.

Рис. 6

Так, например, для графа на *рис*. 6 мостами будут ребра: (1, 4) и (2, 5), — а компонентами реберной двусвязности соответственно наборы вершин: (1, 2, 3), (4), (5, 6, 7, 8).

Рассмотрим задачу нахождения мостов для заданного неориентированного графа *G*.

Эта задача решается с помощью двух поисков в глубину.

При первом поиске, проходя по ребру (*v*, *u*) в направлении от вершины *v* к вершине *u* “ориентируем” его против направления движения, т.е. запрещаем прохождение по ребру (*v*, *u*) в направлении от *v* к *u*. При обнаружении новой вершины заносим ее в конец списка *list*.

Рис. 7

Применим описанный алгоритм к графу с *рис*. 6. Начиная из вершины 1, проходим по ребрам (1, 2) и (2, 3), ориентируя их против направления движения. Из вершины 3 нет ребер, ведущих в непосещенные вершины. Возвращаемся в вершину 2, проходим по ребрам (2, 5), (5, 6), (6, 7) и (7, 8). Возвращаемся в вершину 1, проходим по ребру (1, 4). Возвращаемся в вершину 1 и завершаем поиск в глубину.

На *рис*. 7 представлено то, как будет выглядеть граф с *рис*. 6 по окончании первого поиска в глубину. При этом по “ориентированным” ребрам можно ходить только в направлении, указанном стрелками, а по “неориентированным” ребрам в обоих направлениях. Список *list* выглядит следующим образом: 1, 2, 3, 5, 6, 7, 8, 4.

Второй поиск в глубину осуществляется с учетом “ориентированных” ребер. Внешний цикл по вершинам должен идти в том порядке, в каком они записаны в списке *list*. Получающиеся деревья поиска красим каждое в свой цвет. Эти деревья являются компонентами реберной двусвязности нашего графа. Для графа с *рис*. 7 получим следующие три дерева поиска: (1, 3, 2), (5, 8, 7, 6) и (4).

Рис. 8

Теперь остается только выбрать ребра исходного графа *G*, соединяющие вершины разного цвета (*рис*. 8): (1, 4) и (2, 5) — именно они и будут мостами.

Поясним, почему деревья поиска в глубину, получающиеся во время второго поиска являются компонентами реберной двусвязности нашего графа.

Пусть очередное дерево поиска получилось в результате запуска из некоторой вершины *x* компоненты реберной двусвязности *A* . Докажем, что полученное дерево совпадает с *A*, в предположении, что все деревья поиска, полученные ранее, действительно образуют компоненты реберной двусвязности.

Рис. 9

Во-первых, покажем, что дерево поиска в глубину не может содержать вершин из какой-то другой компоненты реберной двусвязности *B* (*рис*. 9).

Если между компонентами *A* и *B* нет ребер, то утверждение очевидно. Пусть между *A* и *B* есть некоторое ребро (*v*, *u*) (более одного ребра между A и B быть не может, поскольку в противном случае они образовывали бы одну компоненту реберной двусвязности).

Если вершина *u* уже обработана, то пройти по ребру (*v*, *u*), а, значит, и попасть в компоненту *B*, нельзя.

Если же вершина *u* еще не обработана, то она стоит в списке *list* заведомо позже вершины *v*. Это значит, что при первом поиске в глубину мы попали в *v*, когда *u* была еще белой. Не пройдя по ребру (*v*, *u*), из вершины *v* в вершину *u* попасть невозможно (иначе *A* и *B* образовывали бы одну компоненту реберной двусвязности). Поэтому при первом поиске в глубину “сориентируем” ребро (*v*, *u*) в направлении от *u* к *v*, а, значит, при втором поиске в глубину попасть из *A* в *B* будет уже невозможно.

Рис. 10

Покажем теперь, почему дерево поиска, начатого из некоторой вершины *x* компоненты реберной двусвязности *A*, будет содержать все вершины из *A* (*рис*. 10). Будем рассуждать от противного. Предположим, что некоторая вершина *y*, лежащая в этой же компоненте, не войдет в дерево поиска. Поскольку, по предположению все деревья поиска, полученные ранее, действительно образуют компоненты реберной двусвязности, то *y* может не попасть в дерево второго поиска в глубину только в том случае, когда она не достижима из *x* в графе с учетом ориентации ребер. Обозначим за *X* множество вершин компоненты *A*, достижимых из вершины *x* при втором поиске в глубину, а за *Y* множество вершин компоненты *A*, из которых при втором поиске в глубину достижима вершина *y*. Поскольку *X* и *Y* лежат внутри одной компоненты реберной двусвязности, то между ними есть, по крайней мере, два ребра. А так как вершина *y* недостижима из вершины *x*, то все ребра между *X* и *Y* ориентированы от *Y* к *X*. Все ориентированные ребра принадлежат деревьям первого поиска в глубину, причем ребра между *X* и *Y* принадлежат одному дереву (*X* и *Y* подмножества компоненты реберной двусвязности *A*, а компонента реберной двусвязности всегда связна). Но тогда получается, что при построении одного дерева поиска в глубину мы из *X* в *Y* попадали минимум два раза, а из *Y* в *X* ни одного, чего быть не может — противоречие. Следовательно, все вершины *A* достижимы из *x*, а, значит, войдут в дерево поиска.

Таким образом, деревья поиска в глубину, получающиеся при втором поиске в глубину, являются компонентами реберной двусвязности исходного графа *G*.

Для того чтобы запрещать прохождение по ребру в заданном направлении, а также проверять возможность прохождения, полезно вместе с каждым ребром хранить два флажка, отвечающих за возможность прохождения по ребру в каждом из направлений.

По окончанию работы программы переменная *num* будет содержать количество компонент реберной двусвязности заданного графа, а в массиве *color* хранятся номера компонент реберной двусвязности, к которым принадлежат соответствующие вершины.

Точки сочленения

Точкой сочленения неориентированного графа *G* называется вершина, при удалении которой вместе со всеми смежными ребрами увеличивается количество компонент связности графа. Соответственно для связного графа точкой сочленения называется вершина, при удалении которой граф перестает быть связным.

Так, например, для графа на *рис*. 11 точками сочленения будут вершины 1 и 2.

Рис. 11

Рассмотрим задачу нахождения точек сочленения для заданного неориентированного графа *G*.

Эта задача решается с помощью поиска в глубину следующим образом.

Применив к графу, изображенному на *рис*. 11, алгоритм поиска в глубину, получим некоторый лес поиска в глубину (*рис*. 12).

Рис. 12

Корень дерева (начальная вершина) поиска в глубину является точкой сочленения тогда и только тогда, когда если у него более одного сына в дереве поиска в глубину. Сыновьями вершины *v* являются вершины, впервые обнаруженные из *v* при поиске в глубину. Для графа, изображенного на *рис*. 12, вершина 1 имеет двух сыновей: 4 и 2, — и поэтому она является точкой сочленения.

Действительно, если корень дерева имеет только одного сына или же вообще не имеет сыновей, то он, очевидно, не является точкой сочленения. Если же у корня более одного сына, то к нему “подвешены” несколько поддеревьев, между которыми нет ребер (при поиске в глубину в неориентированном графе перекрестных ребер быть не может). Таким образом, при удалении корня количество компонент связности увеличится, а, значит, он является точкой сочленения.

Так для графа, изображенного на *рис*. 12, к вершине 1 “подвешены” два поддерева. Первое состоит из одной вершины 4, второе из вершин 2, 3, 5, 6, 7. При удалении вершины 1 граф распадается на две компоненты связности: (4) и (2, 3,5 ,6, 7).

Для того чтобы понять, является ли промежуточная вершина *v* дерева поиска в глубину точкой сочленения, надо узнать, существует ли поддерево с корнем в одном из сыновей *u* вершины *v*, которое “отсоединится” от дерева поиска при удалении самой вершины *v*. Поддерево может “отсоединиться” от дерева поиска в том и только том случае, когда из вершин этого поддерева нет обратных рёбер, ведущих в вершины, обнаруженные до вершины *v*.

Пусть $up[u]$ — минимальное время обнаружения среди всех вершин, которые могут быть достигнуты из поддерева с корнем в вершине *u* при прохождении ровно по одному обратному ребру.

Тогда промежуточная вершина дерева поиска в глубину *v* является точкой сочленения тогда и только тогда, когда у нее существует сын *u* такой, что $up[u] \geq d[v]$, где $d[v]$ — время обнаружения вершины *v*. В соответствии с этим признаком, вершина 2 графа (*рис*. 12) является точкой сочленения, поскольку у вершины 2 есть сын 5 такой, что $up[5]=d[2]$ (в поддереве с корнем в вершине 5 есть только одно обратное ребро (7, 2)).

Действительно, если существует такой сын *u* вершины *v*, что $up[u] \geq d[v]$, то при удалении вершины *v* поддерево с корнем *u* отделится от дерева, образовав новую компоненту связности, а, значит, *v* — точка сочленения. Если же такого сына не существует, то при удалении *v* количество компонент связности не увеличится, т.к. все поддеревья с корнями в сыновьях вершины *v* соединены с “верхней” частью исходного дерева обратными ребрами.

Таким образом, для нахождения точек сочленения в процессе поиска в глубину необходимо отслеживать количество сыновей $ch[v]$ у корней деревьев поиска в глубину, а также вычислять величину $up[v]$ для всех вершин графа.

Величина $up[v]$ вычисляется как минимум из величин $up[u]$ для всех вершин *u*, являющихся сыновьями *v* в дереве поиска в глубину, и времён обнаружения $d[w]$ всех вершин *w*, достижимых непосредственно из *v* по обратному ребру. Изначально $up[v]$ присваивается значение $n+1$, заведомо большее времени обнаружения любой вершины.

Поиск мостов и точек сочленения представляет интерес, например, для анализа надежности компьютерных сетей.

Задача о поиске мостов соответствует вопросу нахождения соединительных линий, при поломке одной из которых, сеть перестает быть связной, а задача о поиске точек сочленения позволяет разрешить вопрос нахождения компьютеров, при поломке одного из которых сеть перестает быть связной.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 10000
#define MAXM 100000

typedef struct {
       int v, n, id;
       char u;
} edge;

edge e[MAXM << 1];
int h[MAXN], d[MAXN], l[MAXN], tm;
char u[MAXM];

void search1(int v) { // первый поиск в глбину (мосты)
```

```
	int i;
	u[d[tm++] = v] = 1;
	for (i = h[v]; i != -1; i = e[i].n)
		if (!u[e[i].v]) {
			search1(e[i].v);
			e[i].u = 1;
		 }
}

void search2(int v, int x) { // второй поиск в глубине (мосты)
	int i;
	l[v] = x;
	for (i = h[v]; i != -1; i = e[i].n)
		if (!e[i].u && l[e[i].v] == -1)
			search2(e[i].v, x);
}

void search3(int v) { // поиск в глубину (точки сочленения)
	int i, c;
	char ok;
	d[v] = l[v] = ++tm;
	for (ok = c = 0, i = h[v]; i != -1; i = e[i].n)
		if (d[e[i].v]) {
			if (d[e[i].v] < l[v]) l[v] = d[e[i].v];
		} else {
			search3(e[i].v);
			if (l[e[i].v] < l[v]) l[v] = l[e[i].v];
			if (d[v] <= l[e[i].v])
				ok = 1;
			c++;
		}
	u[v] = (!v && c > 1) || (v && ok);
}

int main() {
	int n, m, c, i;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d %d", &n, &m);
	memset(h, ~0, sizeof(h));
	for (i = 0; i < m; i++) {
		int u, v, j = i << 1;
		scanf("%d %d", &u, &v); u--; v--;
		e[j].v = v; e[j].n = h[u]; e[j].id = i; h[u] = j++;
		e[j].v = u; e[j].n = h[v]; e[j].id = i; h[v] = j;
	}
	// мосты
	search1(0);
	memset(l, ~0, sizeof(l));
	for (i = 0; i < n; i++)
		if (l[d[i]] == -1) search2(d[i], d[i]);
	memset(u, 0, sizeof(u));
	for (c = i = 0; i < n; i++) {
		int j;
		for (j = h[i]; j != -1; j = e[j].n)
			if (l[i] != l[e[j].v] && !u[e[j].id]) {
				u[e[j].id] = 1;
				c++;
			}
	}
	printf("%d\n", c);
	for (i = 0; i < m; i++)
		if (u[i]) printf("%d\n", i + 1);
	// точки сочленения
	memset(d, 0, sizeof(d));
	memset(u, 0, sizeof(u));
	memset(l, 0, sizeof(l));
	tm = 0;
	search3(0);
	for (c = i = 0; i < n; i++)
		if (u[i]) c++;
	printf("%d\n", c);
```

```
	for (c = i = 0; i < n; i++)
		if (u[i]) printf("%d\n", i + 1);
	return 0;
}
```

Задача 35. Минимальное остовное дерево [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Задан связный неориентированный взвешенный граф G. В графе возможно наличие нескольких ребер между одной и той же парой вершин. Найдите вес минимального остовного дерева в графе G.

Входные данные:

Первая строка входного файла содержит целое число N ($1 \le N \le 10000$) – количество вершин графа. Вторая строка входного файла содержит целое число M ($1 \le M \le 100000$) – количество ребер графа. В каждой из следующих M строк содержатся ровно три числа a, b, c ($1 \le a, b \le N$, $1 \le c \le 100000$). Эти числа описывают ребро, соединяющее вершины с номерами a и b и имеющее вес c. Вершины нумеруются последовательными натуральными числами от 1 до N.

Выходные данные:

Единственная строка выходного файла должна содержать одно число, равное весу минимального остовного дерева в графе G.

Пример:

input.txt	output.txt
4 4 1 2 1 2 3 2 3 4 3 1 4 4	6

Комментарий. Классическая задача поиска на графе. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 10000
#define MAXM 100000

typedef struct {    // u, v – вершины, инцидентные ребру
	int u, v, c; // c – вес ребра
} edge;

int p[MAXN], r[MAXN], idx[MAXM];
edge e[MAXM];

// сортировка ребер по весу
int partition(int l, int r) {
	int i = l - 1, j = r + 1, x = e[idx[l + rand() % (r - l + 1)]].c;
	while (1) {
		do i++; while (e[idx[i]].c < x);
		do j--; while (e[idx[j]].c > x);
		if (i < j) {
			idx[i] ^= idx[j]; idx[j] ^= idx[i]; idx[i] ^= idx[j];
		} else
			return j;
	}
}

void sort(int l, int r) {
	while (l < r) {
		int m = partition(l, r);
		sort(l, m);
		l = m + 1;
```

```
	}
}

int find(int x) { // поиск множества, к которому принадлежит вершина x
	if (p[x] == x) return x;
	else return p[x] = find(p[x]);
}

void join(int x, int y) { // слияние множеств x и y
	if (r[x] < r[y])
		p[x] = y;
	else {
		p[y] = x;
		if (r[x] == r[y])
			r[y]++;
	}
}

int main() {
	int n, m, c, i;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d %d", &n, &m);
	for (i = 0; i < m; i++) {
		scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].c);
		e[i].u--; e[i].v--;
		idx[i] = i;
	}
	for (i = 0; i < n; i++)
		p[i] = i;
	// сортируем ребра
	sort(0, m - 1);
	// строим дерево
	for (c = i = 0; i < m; i++) {
		int ri = find(e[idx[i]].u), rj = find(e[idx[i]].v);
		if (ri != rj) {
			c += e[idx[i]].c;
			join(ri, rj);
		}
	}
	printf("%d\n", c);
	return 0;
}
```

Задача 36. Кратчайший путь [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Задан связный неориентированный взвешенный граф G. В графе возможно наличие нескольких ребер между одной и той же парой вершин. Найдите вес кратчайшего пути между двумя заданными вершинами A и B.

Входные данные:

Первая строка входного файла содержит целое число N ($1 \le N \le 10000$) – количество вершин графа. Вторая строка входного файла содержит целое число M ($1 \le M \le 100000$) – количество ребер графа. В каждой из следующих M строк содержатся ровно три числа a, b, c ($1 \le a, b \le N$, $1 \le c \le 100000$). Эти числа описывают ребро, соединяющее вершины с номерами a и b и имеющее вес c. Последние две строки содержат целые числа A и B ($1 \le A, B \le N$) – начальную и конечную вершины пути. Вершины нумеруются последовательными натуральными числами от 1 до N.

Выходные данные:

Единственная строка выходного файла должна содержать одно целое число, равное весу кратчайшего пути между вершинами A и B в графе G.

Пример:

input.txt	output.txt
3 3	2

1 2 3 1 3 1 2 3 1 1 2	

Комментарий. Классическая задача поиска на графе на алгоритм Дейкстры. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ».* Однако алгоритм Дейкстры необходимо реализовывать с использованием кучи[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 10000
#define MAXM 100000
#define INF 2000000000

typedef struct {    // v – вершина, в которую ведет ребро
	int v, c, n; // c – вес ребра
} edge;             // n – следующее ребро

int d[MAXN], h[MAXN]; // d – текущее расстояние до вершин, h – заголовки списков смежности
edge e[MAXM << 1];    // e – списки смежности вершин
int heap[MAXN + 1], idx[MAXN + 1]; // heap - куча, idx – позиция элемента в куче

void siftup(int i) {  // проталкивание по куче вверх
	while (i > 1) {
		int p = i >> 1;
		if (d[heap[p]] <= d[heap[i]]) break;
		heap[i] ^= heap[p]; heap[p] ^= heap[i]; heap[i] ^= heap[p];
		idx[heap[i]] = i;
		idx[heap[p]] = p;
		i = p;
	}
}

void siftdown(int hs) {  // проталкивание по куче вниз
	int i = 1;
	do {
		int m = d[heap[i]], mi = i, j;
		if ((j = i << 1) <= hs)
			if (d[heap[j]] < m) {
				m = d[heap[j]];
				mi = j;
			}
		if (++j <= hs)
			if (d[heap[j]] < m) {
				m = d[heap[j]];
				mi = j;
			}
		if (i != mi) {
			heap[i] ^= heap[mi]; heap[mi] ^= heap[i]; heap[i] ^= heap[mi];
			idx[heap[i]] = i;
			i = idx[heap[mi]] = mi;
		} else
			break;
	} while (1);
}

int main() {
	int n, m, i, sr, ds;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d %d", &n, &m);
	memset(h, ~0, sizeof(h));
	for (i = 0; i < m; i++) {
		int u, v, c, j = i << 1;
		scanf("%d %d %d", &u, &v, &c); u--; v--;
		e[j].v = v; e[j].c = c; e[j].n = h[u]; h[u] = j++;
```

```
            e[j].v = u; e[j].c = c; e[j].n = h[v]; h[v] = j;
        }
        scanf("%d %d", &sr, &ds); sr--; ds--;
        // заносим в кучу все вершины
        for (i = 0; i < n; i++) {
            heap[i + 1] = i; idx[i] = i + 1;
            d[i] = INF;
        }
        heap[idx[0] = sr + 1] = 0;
        d[heap[1] = sr] = 0; idx[sr] = 1;
        m = n;
        // пока в куче что-либо есть
        while (m) {
            int t;
            // оставшиеся в куче вершины недостижимы
            if (d[t = heap[1]] == INF) break;
        // извлекаем самую верхнюю вершину
            idx[heap[1]] = -1;
            heap[1] = heap[m--]; idx[heap[1]] = 1;
            siftdown(m);
            // пробегаем по смежным вершинам и обновляем расстояния до них
            for (i = h[t]; i != -1; i = e[i].n) {
                int av = d[t] + e[i].c;
                if (av < d[e[i].v] && idx[e[i].v] >= 0) {
                    d[e[i].v] = av;
                    siftup(idx[e[i].v]);
                }
            }
        }
        printf("%d", (d[ds] < INF)?d[ds]:-1);
        return 0;
}
```

Задача 37. Байтесар-коммивояжер [ЛКШ 2004].

Входной файл	**kom.in**
Выходной файл	**kom.out**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Байтесарский коммивояжёр путешествует вокруг Байтотии. Раньше торговцы могли сами выбирать себе маршрут и посещать те города, которые считали нужными и прибыльными. Но эти времена ушли безвозвратно.

Теперь существует Центральный Контролирующий Офис для коммивояжеров. Для каждого торговца теперь был создан список городов, которые необходимо было посетить именно в данном порядке. Это было сделано для улучшения и систематизации торговли. Но как это обычно и случается, маршруты в большинстве случаев далеко не оптимальные.

Один из коммивояжеров обратился к вам с просьбой: выяснить сколько времени займет путешествие.

Города в Байтории названы числами от 1 до n. Столица Байтории имеет номер 1. Именно от туда и начинают свое путешествие коммивояжеры. Поездка между любыми двумя городами соединенными дорогами занимает 1 единицу времени. Дороги были построенны очень аккуратно, поэтому не было ни одного цикла.

Входные данные:

В первой строке текстового файла kom.in записано одно число n - количество городов в Байтории, 1<=n<=30000. В следующих n-1 строках, описана сеть дорог. На каждой из этих строк записаны два целых a и b (1<=a, b<=n; a<>b), означающие, что города a и b связаны дорогой. В строке n+1 есть одно число m равное количеству городов, которые коммивояжер должен посетить, 1<=m<=5 000. В следующих m строках записаны города в маршруте Байтесара - одно число в строке.

Выходные данные:

В первой строке текстового файла kom.out должно быть одно целое число, равное полному времени поездки Байтесара.

Пример:

kom.in	kom.out
5 1 2 1 5 3 5 4 5	7

4 1 3 2 5	

Комментарий. Здесь надо научиться быстро отвечать на запросы типа: «Каково расстояние между вершинами x и y?» В этой задаче можно использовать тот факт, что Байторий представляет собой дерево. Действительно, если мы знаем минимальное расстояние от вершины 1 до всех остальных вершин, то для того, чтобы найти расстояние между вершинами x и y, мы просто складываем расстояния от 1 до x и y и вычитаем из результата удвоенное расстояние от 1 до z, где z – наименьший общий предшественник x и y при обходе дерева от вершины 1. Поскольку у нас только фиксированное количество таких запросов, здесь мы можем решать задачу о наименьшем общем предшественнике в offline. Алгоритм приведен в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 30000
#define MAXM 5002

// система непересекающихся множеств
int p[MAXN], r[MAXN];

int find(int x) {
	if (p[x] == x) return x;
	else return p[x] = find(p[x]);
}

void join(int i, int j) {
	if (r[i] < r[j])
		p[i] = j;
	else {
		p[j] = i;
		if (r[i] == r[j])
			r[i]++;
	}
}

typedef struct {
	int v, n;
} edge;

typedef struct {
	int v, n, idx;
} nei;

edge e[MAXN << 1]; // списки смежных вершин
nei ne[(MAXM << 1) + 2]; // списки смежности для вершин, для которых необходимо считать lca

int he[MAXN], hn[MAXN]; // заголовки списков смежности


int d[MAXN], a[MAXM], ans[MAXN], la[MAXM]; // d – расстояния от корня до вершин
char u[MAXN], up[MAXM << 1];

void lca(int v) { // вычисление lca
	int i;
	u[v] = 1;
	ans[find(v)] = v;
	for (i = he[v]; i != -1; i = e[i].n)
		if (!u[e[i].v]) {
			lca(e[i].v);
			join(find(v), find(e[i].v));
			ans[find(v)] = v;
		}
	for (i = hn[v]; i != -1; i = ne[i].n)
		if (u[ne[i].v])
			la[ne[i].idx] = ans[find(ne[i].v)];
}
```

```
void walk(int v, int di) { // вычисление расстояний от корня до вершин
	int i;
	u[v] = 1;
	d[v] = di;
	for (i = he[v]; i != -1; i = e[i].n)
		if (!u[e[i].v])
			walk(e[i].v, di + 1);
}

int main() {
	int n, m, i, c;
	freopen("kom.in", "r", stdin);
	freopen("kom.out", "w", stdout);
	// читаем входные данные
	scanf("%d", &n);
	memset(he, ~0, sizeof(he));
	for (i = 1; i < n; i++) {
		int j = (i - 1) << 1, a, b;
		scanf("%d %d", &a, &b); a--; b--;
		e[j].v = b; e[j].n = he[a]; he[a] = j++;
		e[j].v = a; e[j].n = he[b]; he[b] = j;
	}
	scanf("%d", &m);
	memset(hn, ~0, sizeof(hn));
	for (a[0] = 0, i = 1; i <= m; i++) {
		int j = (i - 1) << 1;
		scanf("%d", a + i); a[i]--;
		ne[j].v = a[i]; ne[j].n = hn[a[i - 1]]; ne[j].idx = i - 1; hn[a[i - 1]] = j++;
		ne[j].v = a[i - 1]; ne[j].n = hn[a[i]]; ne[j].idx = i - 1; hn[a[i]] = j;
	}
	for (i = 0; i < n; i++)
		p[i] = i;
	memset(ans, ~0, sizeof(ans));
	memset(la, ~0, sizeof(la));
	// вычисляем lca
	lca(0);
	memset(u, 0, sizeof(u));
	// вычисляем расстояния до вершин
	walk(0, 0);
	for (c = i = 0; i < m; i++) // пробегаем по маршруту коммивояжера
		c += d[a[i]] + d[a[i + 1]] - (d[la[i]] << 1);
	printf("%d\n", c);
	return 0;
}
```

Задача 38. Компоненты связности [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Вам задан неориентированный граф с N вершинами и M ребрами ($1 \le N \le 20000$, $1 \le M \le 200000$). В графе отсутствуют петли и кратные ребра. Определите компоненты связности заданного графа.

Входные данные:

Граф задан во входном файле следующим образом: первая строка содержит числа N и M. Каждая из следующих M строк содержит описание ребра – два целых числа из диапазона от 1 до N – номера концов ребра.

Выходные данные:

На первой строке выходного файла выведите число L – количество компонент связности заданного графа. На следующей строке выведите N чисел из диапазона от 1 до L – номера компонент связности, которым принадлежат соответствующие вершины. Компоненты связности следует занумеровать от 1 до L произвольным образом.

Пример:

input.txt	output.txt
4 2 1 2 3 4	2 1 1 2 2

Комментарий. Классическая задача поиска на графе. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXE 200000
#define MAXN 20000

typedef struct {
	int v, n;
} edge;

int readint() {
	char c;
	int v = 0;
	do c = getchar(); while (c < '0' || c > '9');
	do {
		v = v * 10 + c - '0';
		c = getchar();
	} while (c >= '0' && c <= '9');
	return v;
}

edge e[MAXE << 1]; // списки смежности вершин
int head[MAXN], c[MAXN], q[MAXN]; // head – заголовки списков смежности, c –
                                  // c – метки компонент связности, q - очередь
int main() {
	int n, m, i, l;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем вход
	n = readint(); m = readint();
	memset(head, ~0, sizeof(head));
	for (i = 0; i < m; i++) {
		int u, v, j = i << 1;
		u = readint() - 1;
		v = readint() - 1;
		e[j].v = v; e[j].n = head[u]; head[u] = j++;
		e[j].v = u; e[j].n = head[v]; head[v] = j;
	}
	// обходим граф в ширину
	memset(c, ~0, sizeof(c));
	for (l = i = 0; i < n; i++)
		if (c[i] == -1) {
			int h, t;
			c[q[h = t = 0] = i] = l;
			while (h <= t) {
				int v = q[h++], j;
				for (j = head[v]; j != -1; j = e[j].n)
					if (c[e[j].v] == -1)
						c[q[++t] = e[j].v] = l;
			}
			l++;
		}
	// выводим количество и сами компоненты
	printf("%d\n", l);
	for (i = 0; i < n; i++)
		printf("%d ", c[i] + 1);
	return 0;
}
```

Задача 39. Максимальный поток[ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Найдите величину максимального потока в сети.

Входные данные:

Первая строка входного файла содержит целое число N ($2 \le N \le 800$) – количество вершин сети. Вторая строка входного файла содержит целое число M ($1 \le M \le 10000$) – количество дуг сети. В каждой из следующих M строк содержится ровно три числа a, b, c ($1 \le a, b \le N$, $1 \le c \le 100000$). Эти числа описывают дугу, идущую из вершины с номером a в вершину с номером b и имеющую пропускную способность c. Последние две строки содержат целые числа s и t ($1 \le s, t \le N$, $s \ne t$) – номера вершин, являющихся истоком и стоком, соответственно. Вершины нумеруются последовательными натуральными числами от 1 до N. В сети между фиксированной парой вершин в фиксированном направлении может быть не более одной дуги.

Выходные данные:

Единственная строка выходного файла должна содержать одно целое число, равное величине максимального потока в сети, описанной во входном файле.

Пример:

input.txt	output.txt
4 4 1 2 1 2 3 2 1 4 2 4 3 1 1 3	2

Комментарий. Классическая задача максимального поиска в сети. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12]. Ниже приведены реализации трех методов поиска потоко в сети.

Программа.

```
// алгоритм Эдмондса-Карпа
#include <stdio.h>
#include <string.h>
#define MAXN 800
#define INF 0xFFFFFFF
#define MAXM 10000
#define min(a, b) ( ((a) < (b)) ? (a):(b) )

typedef struct {
	int v, n;
} edge;

int c[MAXN][MAXN]; // пропускные способности
int f[MAXN][MAXN], q[MAXN], p[MAXN];   // f – поток, q - очередь, p – предыдущая вершина
                                       // при поиске
char u[MAXN];       // метки
int af[MAXN];       // поток, «донесенный» до вершин
edge e[MAXM << 1];  // списки смежности вершин
int head[MAXN];     // заголовки списков смежности

int main() {
	int n, m, i;
	int so, ds;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d %d" , &n, &m);
	memset(head, ~0, sizeof(head));
	for (i = 0; i < m; i++) {
		int a, b, d, j = i << 1;
		scanf("%d %d %d", &a, &b, &d); b--; a--;
		c[a][b] = d;
		e[j].v = b; e[j].n = head[a]; head[a] = j++;
		e[j].v = a; e[j].n = head[b]; head[b] = j;
	}
	scanf("%d %d", &so, &ds); so--; ds--;
	m = 0;
	do {
	// ищем путь из so в ds
		int h, t;
```

```
			memset(u, 0, sizeof(u));
			memset(af, 0, sizeof(af));
			memset(p, ~0, sizeof(p));
			// поиск в ширину из so
			af[so] = INF;
			u[q[h = t = 0] = so] = 1;
			while (h <= t) {
				int v = q[h++], a = af[v], j;
				u[v] = 2;
				for (j = head[v]; j != -1; j = e[j].n) { // пробегаем по всем ребрам
					i = e[j].v;
					if (c[v][i] > f[v][i]) { // если ребро не насыщено
						int at = min(c[v][i] - f[v][i], a);
						if (!u[i]) {
							u[q[++t] = i] = 1;
							af[i] = at;
							p[i] = v;
						} else
						if (u[i] == 1 && af[i] < t) {
							af[i] = at;
							p[i] = v;
						}
					}
				}
			}
			if (u[ds]) { "откат" назад
				int j = p[i = ds], delta = af[ds];
				while (j != -1) {
					f[i][j] = -(f[j][i] += delta);
					j = p[i = j];
				}
				m += delta;
			}
		} while (u[ds]); // пока путь существует
		printf("%d\n", m);
		return 0;
}

// алгоритм Generic preflow-push
#include <stdio.h>
#include <string.h>
#define MAXN 800
#define MAXM 10000

typedef struct {
		int v, n;
} edge;

int min(int a, int b) {
		return (a < b)? (a) : (b);
}

int f[MAXN][MAXN], c[MAXN][MAXN], h[MAXN], he[MAXN], ex[MAXN];
// f - поток, c - пропускные способности, h - заголовки списков смежности, he - высота,
// ex - избытки потока

edge e[MAXM << 1]; // списки смежности вершин

int main() {
		int n, m, i;
		int so, ds;
		char ok;
		freopen("input.txt", "r", stdin);
		freopen("output.txt", "w", stdout);
		// читаем входные данные
		scanf("%d %d", &n, &m);
		memset(h, ~0, sizeof(h));
		for (i = 0; i < m; i++) {
			int a, b, d, j = i << 1;
			scanf("%d %d %d", &a, &b, &d); a--; b--;
			c[a][b] = d;
			e[j].v = b; e[j].n = h[a]; h[a] = j++;
			e[j].v = a; e[j].n = h[b]; h[b] = j;
```

```
	}
	scanf("%d %d", &so, &ds); so--; ds--;
	// инициализируем алгоритм preflow push
	he[so] = n;
	for (i = h[so]; i != -1; i = e[i].n) {
		int ne = e[i].v;
		f[ne][so] = -(ex[ne] = f[so][ne] = c[so][ne]);
	}
	do { // пытаемся выполнить операцию push или lift
		int j;
		ok = 0;
		for (i = 0; i < n; i++)
			if (ex[i] && i != ds && i != so) { // есть избыток
				int nextp = -1, nextl = -1;
				char found = 0;
				for (j = h[i]; j != -1; j = e[j].n) { // ищем вершину, где можно
					int ne = e[j].v;		// выполнить lift
					if (c[i][ne] > f[i][ne])
						if (he[i] == he[ne] + 1) {
							nextp = ne;
							break;
						} else {
							found = 1;
							if (nextl == -1 || he[ne] < he[nextl])
								nextl = ne;
						}
				}
				if (nextp == -1)
					if (found)
						he[i] = 1 + he[nextp = nextl];
				if (nextp != -1) { // выполняем push
					int z = min(ex[i], c[i][nextp] - f[i][nextp]);
					f[nextp][i] = -(f[i][nextp] += z);
					ex[i] -= z; ex[nextp] += z;
					ok = 1; // это возможно
				}
				break;
			}
	} while (ok); // пока возможно
	for (m = i = 0; i < n; i++)
		m += f[so][i];
	printf("%d\n", m);
	return 0;
}

// алгоритм lift-to-front
#include <stdio.h>
#include <string.h>
#define MAXN 800
#define MAXM 10000

typedef struct {
	int v, n;
} edge;

// f - поток, c - пропускные способности
// h - заголовки списков смежности, he - высота,
// ex - избытки потока, curr - обрабатываемые вершины, next - следующие в списке, l - список
int c[MAXN][MAXN], f[MAXN][MAXN]; //
int h[MAXN], he[MAXN], ex[MAXN], curr[MAXN], l[MAXN], next[MAXN];
edge e[MAXM << 1]; // списки смежных вершин

void lift(int u) { // операция lift, вершина u
	int i, mi;
	for (mi = -1, i = h[u]; i != -1; i = e[i].n) {
		int v = e[i].v;
		if (c[u][v] > f[u][v])
			if (mi == -1 || (he[v] < he[mi]))
				mi = v;
	}
	he[u] = 1 + he[mi];
}
```

```
void push(int u, int v) { // операция push из u в v
	int z = c[u][v] - f[u][v];
	if (z > ex[u]) z = ex[u];
	f[v][u] = -(f[u][v] += z);
	ex[u] -= z; ex[v] += z;
}

void discharge(int u) { // разгрузить вершину u
	int i = curr[u];
	while (ex[u]) // пока есть избыток
		if (i == -1) {
			lift(u);
			i = h[u];
		} else {
			int v = e[i].v;
			if (c[u][v] > f[u][v] && he[u] == he[v] + 1)
				push(u, v);
			else
				i = e[i].n;
		}
	curr[u] = i;
}

int main() {
	int n, m, i, so, ds, prev;
	// читаем входные данные
	scanf("%d %d", &n, &m);
	memset(h, ~0, sizeof(h));
	for (i = 0; i < m; i++) {
		int a, b, d, j = i << 1;
		scanf("%d %d %d", &a, &b, &d); a--; b--;
		c[a][b] = d;
		e[j].v = b; e[j].n = h[a]; h[a] = j++;
		e[j].v = a; e[j].n = h[b]; h[b] = j;
	}
	scanf("%d %d", &so, &ds); so--; ds--;
	memcpy(curr, h, sizeof(h));
	// инициализируем алгоритм
	he[so] = n;
	for (i = h[so]; i != -1; i = e[i].n) {
		int v = e[i].v;
		f[v][so] = -(ex[v] = f[so][v] = c[so][v]);
	}
	for (m = i = 0; i < n; i++)
		if (i != so && i != ds) {
			l[m] = i;
			next[m] = m - 1;
			m++;
		}
	i = --m; prev = -1;
	// пока есть что разгружать
	while (i != -1) {
		int u = l[i], oh = he[u];
		discharge(u); // разгружаем вершину
		if (he[u] > oh && prev != -1) {
			// записываем ее в начало списка
			next[prev] = next[i];
			next[i] = m;
			m = i;
		}
		i = next[prev = i];
	}
	for (m = i = 0; i < n; i++)
		m += f[so][i];
	printf("%d\n", m);
	return 0;
}
```

Задача 40. Цикл [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**

Ограничение по времени 1 секунда
Ограничение по памяти 16 мегабайт

Дан граф. Определить, есть ли в нем цикл отрицательного веса, и если да, то вывести его.

Входные данные:

Во входном файле в первой строке число N (1<=N<=100) – количество вершин графа. В следующих N строках находится по N чисел – матрица смежности графа. Все веса ребер не превышают по модулю 10000. Если ребра нет, то соответствующее число равно 100000.

Выходные данные:

В первой строке выходного файла выведите “YES”, если цикл существует или “NO” в противном случае. При его наличии выведите во второй строке количество вершин в искомом цикле (считая одинаковые первую и последнюю) и в третьей строке – вершины, входящие в этот цикл в порядке обхода.

Пример:

input.txt	output.txt
2 0 –1 -1 0	YES 3 1 2 1

Комментарий. Классическая задача поиска на графе на алгоритм Флойда. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 100
#define INF 100000

int g[MAXN][MAXN], p[MAXN][MAXN]; // g – текущие расстояния между вершинами, p –
int path[MAXN << 1], sp;          // «родительские» ссылки, path - путь

void search(int i, int j) { // восстанавливаем путь
	if (p[i][j] == i || p[i][j] == j || p[i][j] == -1)
		return ;
	search(i, p[i][j]);
	path[sp++] = p[i][j];
	search(p[i][j], j);
}

int main() {
	int n, i, j, k;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d", &n);
	for (i = 0; i < n; i++)
		for (j = 0; j < n; j++)
			scanf("%d", g[i] + j);
	memset(p, ~0, sizeof(p));
	// «прогоняем» алгоритм Флойда
	for (k = 0; k < n; k++)
		for (i = 0; i < n; i++)
			for (j = 0; j < n; j++) {
				int t;
				if (g[i][k] < INF && g[k][j] < INF &&
					(t = g[i][k] + g[k][j]) < g[i][j]) {
					g[i][j] = t;
					p[i][j] = k;
					if (i == j && g[i][j] < 0) // если появился отрицательный
						goto done;          // вес, выходим
				}
			}
done:
	if (k < n) { // цикл есть
		k = i;
		puts("YES");
		search(i, i);
```

```
			printf("%d\n%d ", sp + 2, k + 1);
			for (i = 0; i < sp; i++)
				printf("%d ", path[i] + 1);
			printf("%d", k + 1);
		} else
			puts("NO");
		return 0;
}
```

Задача 41. В поисках невест[ЛКШ 2004].

Входной файл brides.in
Выходной файл brides.out
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Однажды король Флатландии решил отправить k своих сыновей на поиски невест. Всем известно, что во Флатландии n городов, некоторые из которых соединены дорогами. Король живет в столице, которая имеет номер 1 а город с номером n знаменит своими невестами. Итак, король повелел, чтобы каждый из его сыновей добрался по дорогам из города 1 в город n. Поскольку, несмотря на обилие невест в городе n, красивых среди них не так много, сыновья опасаются друг друга. Поэтому они хотят добраться до цели таким образом, чтобы никакие два сына не проходили по одной и той же дороге (даже в разное время). Так как король любит своих сыновей, он хочет, чтобы среднее время сына в пути до города назначения было минимально.

Входные данные:

В первой строке входного файла находятся числа n, m и k - количество городов и дорог во Флатландии и сыновей короля, соответственно ($2 \le n \le 200$, $1 \le m \le 2000$, $1 \le k \le 100$). Следующие строки содержат 3*m целых положительных числа. каждая тройка чисел - города, которые соединяет соответствующая дорога и время, которое требуется для ее прохождения (время не превышает 10^6). По дороге можно перемещаться в любом из двух направлений, два города могут быть соединены несколькими дорогами.

Выходные данные:

Если выполнить повеление короля невозможно, выведите на первой строкечисло -1. В противном случае выведите на первой строке минимальное возможное среднее время (с точностью 5 знаков после десятичной точки), которое требуется сыновьям, чтобы добраться до города назначения, не менее чем с пятью знаками после десятичной точки. В следующих k строках выведите пути сыновей, сначала число дорог в пути и затем номера дорог в пути в том порядке, в котором их следует проходить. Дороги нумеруются, начиная с единицы, в том порядке, в котором они заданы во входном файле.

Пример:

brides.in	brides.out
5 8 2 1 2 1 1 3 1 1 4 3 2 5 5 2 3 1 3 5 1 3 4 1 5 4 1	3.00000 3 1 5 6 3 2 7 8

Комментарий. Задача на поиск максимального потока минимальной стоимости. Описание можно найти в [Кристофидес Н. «Теория графов. Алгоритмический подход.»]

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 200
#define MAXM 2000
#define INF 0xFFFFFFF

typedef struct {  // s – начало ребра, v – конец ребра, n – следующее ребро, c – пропускная
	int s, v, n, c, f, r; // способность ребра, f – поток, r – обратное ребро
} edge;

edge e[MAXM << 1]; // ребра (списки смежных вершин)
int he[MAXN], d[MAXN], p[MAXN], phi[MAXN], st[MAXM]; // he – заголовки списков смежных
char u[MAXN];		// вершин, d – расстояния до вершин, p – предыдущие вершины при
			// поиске, phi – потенциалы, st - путь

int main() {
	int n, m, k, i;
	freopen("brides.in", "r", stdin);
```

```
	freopen("brides.out", "w", stdout);
	// читаем входные данные
	scanf("%d %d %d", &n, &m, &k);
	memset(he, ~0,sizeof(he));
	for (i = 0; i < m; i++) {
		int u, v, c, j = i << 1;
		scanf("%d %d %d", &u, &v, &c); u--; v--;
		e[j].s = u; e[j].v = v; e[j].c = c; e[j].n = he[u]; he[u] = j; e[j].r = j + 1;
j++;
		e[j].s = v; e[j].v = u; e[j].c = c; e[j].n = he[v]; he[v] = j; e[j].r = j - 1;
	}
	// для каждого сына ищем минимальный по стоимости путь из 0 в n - 1
	for (i = 0; i < k; i++) {
		int j, am = 0;
	// инициализируем алгоритм Дейкстры
		memset(p, ~0, sizeof(p));
		memset(u, 0, sizeof(u));
		for (j = 0; j < n; j++)
			d[j] = INF;
		d[0] = 0;
		while (1) { // алгоритм Дейкстры находит минимальные расстояния d до вершин
			int mi;
			for (mi = -1, j = 0; j < n; j++)
				if (!u[j] && d[j] < INF && (mi == -1 || d[j] < d[mi]))
					mi = j;
			if (mi == -1) break;
			u[mi] = 1;
			am = d[mi];
			for (j = he[mi]; j != -1; j = e[j].n)
				if (e[j].f < 1) {
					int z = e[j].v, ad = d[mi] + phi[mi] - phi[z] + ( (e[j].f <
0) ? (- e[j].c) : (e[j].c) );
					if (d[z] > ad) {
						d[z] = ad;
						p[z] = j;
					}
				}
		}
	if (d[n - 1] == INF) break; // если n - 1 не достижима, выходим
	// обновляем значения потенциалов и потока
		for (j = 0; j < n; j++)
			if (d[j] == INF) d[j] = am;
		for (j = 0; j < n; j++)
			phi[j] += d[j];
		j = n - 1;
		while (j != 0) {
			int pe = p[j];
			e[pe].f++;
			e[e[pe].r].f--;
		j = e[pe].s;
		}
	}
	if (i == k) { // если все прошло нормально для всех сыновей
		int s;
		// считаем стоимость потока
		for (m <<= 1, s = i = 0; i < m; i++)
			if (e[i].f > 0) s += e[i].c;
		printf("%.5lf\n", (double)s / k);
		// жадно конструируем пути
		for (i = 0; i < k; i++) {
			int j = 0, c = 0;
			while (j + 1 != n) { // пока мы не дошли до n - 1
				int q;
				for (q = he[j]; q != -1; q = e[q].n) // выбираем первое насыщенное
					if (e[q].f > 0) {   // ребро
						st[c++] = (q >> 1) + 1;
						e[q].f = 0;
						j = e[q].v;
						break;
					}
				if (q == -1) puts("oblom");
			}
			// выводим путь
```

```
                printf("%d ", c);
                for (j = 0; j < c; j++)
                        printf("%d ", st[j]);
                putchar('\n');
        }
    } else // в противном случае
          puts("-1");
    return 0;
}
```

Задача 42. Это не баг - это фича! [ЛКШ 2004].

Входной файл	**bugs.in**
Выходной файл	**bugs.out**
Ограничение по времени	**5 секунд**
Ограничение по памяти	**128 мегабайт**

У вам имеется очень старая программа, которая настолько проста, что допустить в ней более N ($N \le 20$) просто невозможно. Вам попалась как раз версия, где есть все баги. Вы скачали из Интернета K ($K \le 100$) патчей к этой программе, которые устраняют баги. Но в этих патчах тоже есть ошибки, поэтому они не только устраняют, но иногда и прибавляют баги. Каждая программа устанавливается некоторое время, поэтому хотелось бы затратить наименьшее количество времени на исправление всех ошибок. Про каждую программу известно, какие баги должны присутствовать при установки данного патча, какие отсутствовать и на какие баги эта программа не обращает внимание. Если программу возможно установить, то она исправит некоторые баги, а некоторые добавит. Какие именно записано во входном файле.

Входные данные:

Во входном файле может быть несколько тестов. Каждый описывается следующим образом:

В первой строке блока записано N и M соответственно. Далее следует описание скаченных вами патчей. Каждый патч описывается временем установки и двумя таблицами. Первая таблица описывает, что необходимо для становки программы. Таблица записана N символами из набора (0,+,-). 0 означает что эта ошибка не важна для становки данной программы. + означает, что это баг должен присутствовать при установки патча, - означает, что та ошибка должна отсутствовать при установки данной программы. Следующая таблица записана в том же формате. Здесь 0 означает, что этот баг не будет исправлен, - означает, что этот баг будет исправлен патчем, + же означает, что этот баг появится после установки патча.

Файл заканчивается двумя нулями.

Выходные данные:

Для I – го теста во входном файле выведите Product I, а затем фразу «Fastest sequence takes Q seconds.» Q здесь – это минимальное время, за которое можно исправить все баги. Если все ошибки исправить невозможно выведите «Bugs cannot be fixed.»

Между тестами отступайте строчку.

Пример:

bugs.in	bugs.out
3 3 1 000 00- 1 00- 0-+ 2 0-- -++ 4 1 7 0-0+ ---- 0 0	Product 1 Fastest sequence takes 8 seconds. Product 1 Bugs cannot be fixed.

Комментарий. Задача на алгоритма Дейкстры по графу, вершины которого являюся двоичными векторами: i-й элемент равен 1, если соответствующий баг присутствует в состоянии, 0 - отсутствует.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 20
#define MAXS (1 << MAXN)

typedef struct {
       int d;
       int nessp, nessa;// nessp – какие баги должны быть,nessa - каких багов не должны быть
       int np, na;// np – какие баги появятся, na – какие исчезнут
} edge;
```

```
int heap[MAXS + 1]; // куча
int idx[MAXS + 1], d[MAXS + 1]; // idx – позиции вершин в куче, d - расстояния до вершин

void swaph(int i, int j) { // обмен элементов кучи
	if (i != j) {
		heap[i] ^= heap[j]; heap[j] ^= heap[i]; heap[i] ^= heap[j];
		idx[heap[i]] = i; idx[heap[j]] = j;
	}
}

void siftdown(int hs) { // просеивание по куче вниз
	int i = 0;
	while (1) {
		int mi = i, j;
		if ((j = (i << 1) + 1) < hs && d[heap[j]] < d[heap[mi]])
			mi = j;
		if (++j < hs && d[heap[j]] < d[heap[mi]])
			mi = j;
		if (mi != i) {
			swaph(mi, i);
			i = mi;
		} else
			break;
	}
}

void siftup(int i) { // просеивание по куче вверх
	int p = (i - 1) >> 1;
	while (i > 0 && d[heap[i]] < d[heap[p]]) {
		swaph(p, i);
		i = p;
	  p = (i - 1) >> 1;
	}
}

char s1[MAXN + 1], s2[MAXN + 1];
char u[MAXS + 1];   // метки
edge e[MAXN << 4]; // списки смежности вершин

int main() {
	int n, m, tc = 1;
	freopen("bugs.in", "r", stdin);
	freopen("bugs.out", "w", stdout);
	// читаем входные данные
	scanf("%d %d", &n, &m);
	while (n > 0 || m > 0) {
		int i, hs;
		for (i = 0; i < m; i++) {
			int j;
			scanf("%d %s %s\n", &e[i].d, s1, s2);
			e[i].nessa = e[i].nessp = e[i].na = e[i].np = 0;
			for (j = 0; j < n; j++)
				switch (s1[j]) {
					case '+': e[i].nessp |= 1 << j; break;
					case '-': e[i].nessa |= 1 << j; break;
				}
			for (j = 0; j < n; j++)
				switch (s2[j]) {
					case '+': e[i].np |= 1 << j; break;
					case '-': e[i].na |= 1 << j; break;
				}
		}
		memset(d, ~0, sizeof(d));
	memset(idx, ~0, sizeof(idx));
		memset(u, 0, sizeof(u));
		// инициализируем алгоритм Дейкстры
		d[(1 << n) - 1] = 0; heap[0] = (1 << n) - 1; hs = 1;
		while (hs) { // пока в куче что-либо есть
			int j;
			u[i = heap[0]] = 1; // извлекаем вершину из корня кучи
			hs--;
			if (hs > 0) {
```

```
                        swaph(0, hs);
                        siftdown(hs);
                    }
                    idx[i] = -1;
                    for (j = m - 1; j >= 0; j--) // пробегаем по списку смежных вершин
                        if ((i & e[j].nessp) == e[j].nessp &&
                            ((~i) & e[j].nessa) == e[j].nessa) {
                            int z = d[i] + e[j].d, f = (i | e[j].np)  & (~e[j].na);
                            if (!u[f])
                            if (d[f] == -1) {
                                d[f] = z;
                                heap[idx[f] = hs++] = f;
                                siftup(idx[f]);
                            } else
                            if (d[f] > z) {
                                d[f] = z;
                                if (idx[f] == -1) puts("oblom");
                                siftup(idx[f]);
                            }
                        }
                }
                // выводим результат
                printf("Product %d\n", tc++);
                if (d[0] >= 0)
                    printf("Fastest sequence takes %d seconds.\n", d[0]);
                else
                    puts("Bugs cannot be fixed.");
                putchar('\n');
                scanf("%d %d", &n, &m);
        }
        return 0;
}
```

Задача 43. Цикл [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**4 секунды**
Ограничение по памяти	**1 мегабайт**

Найдите в заданном графе цикл отрицательного веса.

Входные данные:

В первой строке входного файла записаны 2 числа N и M - количество вершин и количество ребер соответственно. Количество вершин не превосходит 5000, а количество ребер 10000. Далее описаны все ребра тремя числами x_i, y_i, z_i. Это означает, что из вершины x_i в вершину y_i ведет ребро весом z_i. Возможны петли и кратные ребра.

Выходные данные:

Выведите количество вершин в найденном цикле. Далее выведите сами эти вершин в порядке обхода. Если не существует цикла отрицательного веса, то выведите 0.

Пример:

input.txt	output.txt
4 4 4 2 -100 2 3 10 3 4 10 1 4 –100	3 3 4 2

Комментарий. Классическая задача поиска на графе на алгоритм Форда-Беллмана. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 5005
#define MAXM 10005


typedef struct {
```

```
	int u, v, c;
} edge;

edge e[MAXM]; // списки смежных вершин
int d[MAXN], p[MAXN], s[MAXN << 1]; // d – расстояния до вершин, p – предыдущие вершины,
char u[MAXN];	// s – сам цикл, u - метки

int main() {
	register int i;
	int n, m;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d %d", &n, &m);
	for (i = 0; i < m; i++) {
		scanf("%d %d %d", &e[i].u, &e[i].v, &e[i].c);
		e[i].u--; e[i].v--;
	}
	// «прогоняем» алгоритм Форда-Беллмана
	memset(p, ~0, sizeof(p));
	for (i = n - 1; i > 0; i--) {
		int j;
		for (j = m - 1; j >= 0; j--) {
			int t = d[e[j].u] + e[j].c;
			if (d[e[j].v] > t) {
				d[e[j].v] = t;
				p[e[j].v] = e[j].u;
			}
		}
	}
	// проверяем, можем ли мы сделать еще одну релаксацию
	for (i = 0; i < m; i++)
		if (d[e[i].v] > d[e[i].u] + e[i].c) break;
	if (i < m) { // да, можем
		for (m = 0, i = e[i].u; i != -1 && !u[i]; i = p[i]) { // идем по обратным
			u[i] = 1;	// ссылках из этой вершины
			s[m++] = i;
		}
		// восстанавливаем цикл
		if (i >= 0)
			s[m++] = i;
		i = --m;
		do i--; while (i >= 0 && s[i] != s[m]);
		// выводим результат
		printf("%d\n", m - i);
		for (n = m; n > i; n--)
			printf("%d ", s[n] + 1);
	} else // нет, не можем
		putchar('0');
	return 0;
}
```

Задача 44. Строительство дорог [ЛКШ 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**1 мегабайт**

Недавно образовавшаяся страна построила на своей территории N городов. На это были потрачены почти все средства, накопленные ее жителями тяжелым трудом в шахтах и подземельях. Для дальнейшего существования данной страны необходимо построить дороги между городами так, чтобы из каждого города можно было добраться до любого другого. Ваше задача сделать это так, чтобы минимизировать общую длину дорог. Стоит заметить, что каждая дорога, соединяющая два города должна быть минимальной. Кроме этого прежде, чем к вам обратились, одна фирма обманула руководство страны и построила некоторые дороги, которые далеко не все являются нужными. Но все же разрушать их не имеет смысла, да и платить за них нужно. Эти хулиганы (если не сказать хуже) строили не только по несколько дорог между двумя дорогами, но и даже дороги ведущие из города в него самого.

Входные данные:

В первой строке входного файла записаны 2 числа N и M - количество городов и количество уже построенных дорого соответственно. Количество городов не превосходит 4000, а количество построенных дорого 10000. Далее описаны все все построенные дороги двумя числами x_i, y_i. Это означает, что из города x_i в город y_i ведет дорога. Далее следуют координаты городов в выбранной вами декартовой системе координат. Координаты – это целые числа, по модулю не превосходящие 10^5.

Выходные данные:

Выведите наименьшую суммарную длину вновь построенных дорог, приводящих к желаемому результату, с точностью до 8 знаков.

Пример:

input.txt	output.txt
4 4 4 2 -100 2 3 10 3 4 10 1 4 –100	3 3 4 2

Комментарий. Классическая задача поиска на графе на алгоритм Прима. Ссылки могут быть найдены в *Кормен Т., Лейзерстон Ч., Ривест Р. «Алгоритмы: построение и анализ»*[12].

Программа.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#define MAXN 4000

// система непересекающихся множеств
int p[MAXN], r[MAXN];

int find(int x) {
	if (p[x] == x) return x;
	else return p[x] = find(p[x]);
}

void join(int i, int j) {
	if (r[i] < r[j])
		p[i] = j;
	else {
		p[j] = i;
		if (r[i] == r[j])
			r[i]++;
	}
}

double d[MAXN]; // расстояния до вершин
int heap[MAXN], idx[MAXN]; // heap – куча, idx – позиции вершин в куче

void swaph(int i, int j) { // обмен элементов кучи
	if (i != j) {
		heap[i] ^= heap[j]; heap[j] ^= heap[i]; heap[i] ^= heap[j];
		idx[heap[i]] = i; idx[heap[j]] = j;
	}
}

void siftup(int i) { // просеивание по куче вверх
	int p = (i - 1) >> 1;
	while (i > 0 && d[heap[i]] < d[heap[p]]) {
		swaph(p, i);
		i = p;
	  p = (i - 1) >> 1;
	}
}

void siftdown(int hs) { // просеивание по куче вниз
	int i = 0;
	while (1) {
		int mi = i, j;
```

```
                if ((j = (i << 1) + 1) < hs && d[heap[j]] < d[heap[mi]])
                        mi = j;
                if (++j < hs && d[heap[j]] < d[heap[mi]])
                        mi = j;
                if (mi != i) {
                        swaph(mi, i);
                        i = mi;
                } else
                        break;
        }
}

int x[MAXN], y[MAXN];

int main() {
        int n, m, i, hs;
        double ans = 0;
        freopen("input.txt", "r", stdin);
        freopen("output.txt", "w", stdout);
        // читаем входные данные
        scanf("%d %d", &n, &m);
        for (i = 0; i < n; i++)
                p[i] = i;
        for (i = 0; i < m; i++) {
                int a, b;
                scanf("%d %d", &a, &b); a--; b--;
                a = find(a); b = find(b);
                if ((a = find(a)) != (b = find(b))) join(a, b);
        }
        for (i = 0; i < n; i++)
                scanf("%d %d", x + i, y + i);
        memset(idx, ~0, sizeof(idx));
        for (i = 0; i < n; i++) {
                idx[heap[i] = i] = i;
                d[i] = 1e30;
        }
        // инициализируем алгоритм Прима
        d[0] = 0;
        hs = n;
        while (hs) { // пока в куче что-либо есть
                int j, r;
                i = heap[0]; idx[i] = -1; r = find(i); // извлекаем корневую вершину
                if (--hs) {
                        idx[heap[0] = heap[hs]] = 0;
                        siftdown(hs);
                }
                ans += sqrt((double)d[i]);
                for (j = 0; j < n; j++) // пробегаем по смежным вершинам и обновляем d
                        if (idx[j] >= 0) {
                                double w = (find(j) == r) ? 0 : ((double)(x[i] - x[j]) * (x[i] -
x[j]) + (double)(y[i] - y[j]) * (y[i] - y[j]));
                                if (d[j] > w) {
                                        d[j] = w;
                                        siftup(idx[j]);
                                }
                        }
        }
        printf("%.10lf\n", ans); // выводим ответ
        return 0;
}
```

Задача 45. Cable TV Network [SEERC 2004].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

The interconnection of the relays in a cable TV network is bi-directional. The network is connected if there is at least one interconnection path between each pair of relays present in the network. Otherwise the network is disconnected. An empty network or a network with a single relay is considered connected. The safety factor f of a network with n relays is:

1. n, if the net remains connected regardless the number of relays removed from the net.
2. The minimal number of relays that disconnect the network when removed.

Write a program that reads several data sets from a text file and computes the safety factor for the cable networks encoded by the data sets.

Входные данные:

Each data set starts with two integers: $0 \le n \le 50$, the number of relays in the net, and m, the number of cables in the net. Follow m data pairs (u,v), u < v, where u and v are relay identifiers (integers in the range 0..n-1). The pair (u,v) designates the cable that interconnects the relays u and v. The pairs may occur in any order. Except the (u,v) pairs, which do not contain white spaces, white spaces can occur freely in input. Input data terminate with an end of file and are correct.

Выходные данные:

For each data set, the program prints on the standard output, from the beginning of a line, the safety factor of the encoded net.

Пример:

input.txt	output.txt
0 0 1 0 3 3 (0,1) (0,2) (1,2) 2 0 5 7 (0,1) (0,2) (1,3) (1,2) (1,4) (2,3) (3,4)	0 1 3 0 2

Комментарий.

Переформулировать эту задачу на языке теории графов можно так: найти минимальный вершинный разрез в графе. Для решения этой задачи можно использовать алгоритм нахождения максимального потока (например, Форда-Фалкерсона, см. [27).

Ключевым моментом является замена всех вершин на ребра пропускной способностью равной единице. Пропускная способность остальных ребер также предполагается равной 1.

После преобразования графа для каждой пары вершин i, j найдем максимальный поток F_{ij}. Ответом к задаче будет $\min(F_{ij})$.

Программа.

```
#include <stdio.h>
#include <ctype.h>

const int INF = 2000000000;
// Граф
int n, m;
int f[100];
struct {
  int s, t, c, f, r, n;
} e[100000];
// Очередь
int q[100];
// Пометки для нахождения потока
int h[100];
// Пометки для восстановления пути
int p[100];

int getint() {
  int c, a;
  c = getchar(); while (!isdigit(c)) c = getchar();
  a = 0;
  while (isdigit(c)) {
    a = a * 10 + c - '0';
    c = getchar();
  }
  return a;
}

// Находит максимальный поток от s к t методом Форда-Фалкерсона
int findflow(int s, int t) {
  int _h, _t, i, v, F;
  F = 0;
  for (i = 0; i < m; i++) e[i].f = 0;
  do {
    for (i = 0; i < n << 1; i++) h[i] = INF;
    q[0] = s; _h = 1; _t =0;
    while ((_t < _h) && (h[t] == INF)) {
```

```
      v = q[_t++];
      for (i = f[v]; i != -1; i = e[i].n) if ((h[e[i].t] == INF) && (e[i].f < e[i].c)) {
        h[e[i].t] = e[i].c - e[i].f;
        if (h[e[i].t] > h[v]) h[e[i].t] = h[v];
        p[e[i].t] = i;
        q[_h++] = e[i].t;
      }
    }
    if (h[t] != INF) {
      F++;
      v = t;
      while (v != s) {
        i = p[v];
        e[i].f++; e[e[i].r].f--;
        v = e[i].s;
      }
    }
  } while (h[t] != INF);
  return F;
}

void adde(int s, int t, int c, int f, int r, int i) {
  e[i].n = ::f[s]; e[i].s = s; e[i].t = t; e[i].c = c; e[i].f = f; e[i].r = r; ::f[s] = i;
}

int main() {
  int i, j, a, b, r, t;
// Ввод и построение графа
  while (scanf("%d %d", &n, &m) == 2) {
    for (i = 0; i < n << 1; i++) f[i] = -1;
    j = 0;
    for (i = 0; i < m; i++) {
      a = getint(); b = getint();
      adde(a, b + n, 1, 0, j + 1, j); j++;
      adde(b + n, a, 0, 0, j - 1, j); j++;
      adde(b, a + n, 1, 0, j + 1, j); j++;
      adde(a + n, b, 0, 0, j - 1, j); j++;
    }
    for (i = 0; i < n; i++) {
      adde(i, n + i, 1, 0, j + 1, j); j++;
      adde(n + i, i, 1, 0, j - 1, j); j++;
    }
    if (m == n * (n - 1) / 2) printf("%d\n", n);
    else if (m == 0) printf("0\n");
    else {
// Перебор
      m = j;
      r = -1;
      for (i = 0; i < n; i++) for (j = i + 1; j < n; j++) {
        t = findflow(i, j + n);
        if ((r == -1) || (t < r)) r = t;
      }
      printf("%d\n", r);
    }
  }
  return 0;
}
```

Задача 46. Evacuation Plan [NEERC 2002].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

The City has a number of municipal buildings and a number of fallout shelters that were built specially to hide municipal workers in case of a nuclear war. Each fallout shelter has a limited capacity in terms of a number of people it can accommodate, and there's almost no excess capacity in The City's fallout shelters. Ideally, all workers from a given municipal building shall run to the nearest fallout shelter. However, this will lead to overcrowding of some fallout shelters, while others will be half-empty at the same time.

To address this problem, The City Council has developed a special evacuation plan. Instead of assigning every worker to a fallout shelter individually (which will be a huge amount of information to keep), they allocated fallout shelters to municipal buildings, listing the number of workers from every building that shall use a given fallout shelter, and left the

task of individual assignments to the buildings' management. The plan takes into account a number of workers in every building - all of them are assigned to fallout shelters, and a limited capacity of each fallout shelter - every fallout shelter is assigned to no more workers then it can accommodate, though some fallout shelters may be not used completely.

The City Council claims that their evacuation plan is optimal, in the sense that it minimizes the total time to reach fallout shelters for all workers in The City, which is the sum for all workers of the time to go from the worker's municipal building to the fallout shelter assigned to this worker.

The City Mayor, well known for his constant confrontation with The City Council, does not buy their claim and hires you as an independent consultant to verify the evacuation plan. Your task is to either ensure that the evacuation plan is indeed optimal, or to prove otherwise by presenting another evacuation plan with the smaller total time to reach fallout shelters, thus clearly exposing The City Council's incompetence.

During initial requirements gathering phase of your project, you have found that The City is represented by a rectangular grid. The location of municipal buildings and fallout shelters is specified by two integer numbers and the time to go between municipal building at the location (Xi, Yi) and the fallout shelter at the location (Pj, Qj) is $D_{i,j} = |Xi - Pj| + |Yi - Qj| + 1$ minutes.

Входные данные:

The input file consists of The City description and the evacuation plan description. The first line of the input file consists of two numbers N and M separated by a space. N ($1 \le N \le 100$) is a number of municipal buildings in The City (all municipal buildings are numbered from 1 to N). M ($1 \le M \le 100$) is a number of fallout shelters in The City (all fallout shelters are numbered from 1 to M).

The following N lines describe municipal buildings. Each line contains there integer numbers Xi, Yi, and Bi separated by spaces, where Xi, Yi ($-1000 \le Xi, Yi \le 1000$) are the coordinates of the building, and Bi ($1 \le Bi \le 1000$) is the number of workers in this building.

The description of municipal buildings is followed by M lines that describe fallout shelters. Each line contains three integer numbers Pj, Qj, and Cj separated by spaces, where Pi, Qi ($-1000 \le Pj, Qj \le 1000$) are the coordinates of the fallout shelter, and Cj ($1 \le Cj \le 1000$) is the capacity of this shelter.

The description of The City Council's evacuation plan follows on the next N lines. Each line represents an evacuation plan for a single building (in the order they are given in The City description). The evacuation plan of ith municipal building consists of M integer numbers $E_{i,j}$ separated by spaces. $E_{i,j}$ ($0 \le E_{i,j} \le 1000$) is a number of workers that shall evacuate from the ith municipal building to the jth fallout shelter.

The plan in the input file is guaranteed to be valid. Namely, it calls for an evacuation of the exact number of workers that are actually working in any given municipal building according to The City description and does not exceed the capacity of any given fallout shelter.

Выходные данные:

If The City Council's plan is optimal, then write to the output file the single word OPTIMAL. Otherwise, write the word SUBOPTIMAL on the first line, followed by N lines that describe your plan in the same format as in the input file. Your plan need not be optimal itself, but must be valid and better than The City Council's one.

Пример:

input.txt	output.txt
3 4 -3 3 5 -2 -2 6 2 2 5 -1 1 3 1 1 4 -2 -2 7 0 -1 3 3 1 1 0 0 0 6 0 0 3 0 2	SUBOPTIMAL 3 0 1 1 0 0 6 0 0 4 0 1
3 4 -3 3 5 -2 -2 6 2 2 5 -1 1 3 1 1 4 -2 -2 7 0 -1 3 3 0 1 1 0 0 6 0 0 4 0 1	OPTIMAL

Комментарий.

Составим «4-дольный» граф. Первая доля состоит из одной вершины S, вторая состоит из N зданеий, третья — их M убежищ, четвертая — из одной вершины T. Построим ребра из 1-й доли во 2-ю, из 2-й в 3-ю, из 3-й в 4-ю. Припишем ребрам пропускные способности: из доли 1 в долю 2 — количеству рабочих в соответствующем здании, из 2 в 3 — равные бесконечности, из 3 в 4 — вместимости убежища. Обозначим стоимости ребер из 2 в 3 равными расстоянию от здания до убежища, а всем остальным — 0. Пустим по графу их S к T поток, такой, что поток по ребрам из S и по ребрам в T занимает всю их пропускную способность, а из 2 в 3 — равен количеству рабочих из соответствующего здания, эвакуируемых в соответствующее убежище. Очевидно, что такой поток будет максимальным. Нам требуется определить, является ли он также минимальным по стоимости. Это можно сделать, используя часть алгоритма построения минимального по стоимости потока, основанного на выявлении циклов отрицательного веса (см «Теория графов. Алгоритмический подход.» (Кристофидес)). Если выяснится, что поток не оптимальный, можно один раз уменьшить его стоимость, получив поток с меньшей стоимостью (но, возможно, еще не оптимальный — этого не требуется по условию задачи).

Программа.

```
#include <stdio.h>
#include <stdlib.h>

const int INF = 1000000;
struct {
    int x, y, c;
} Building[100], Shelter[100];
int c[100][100];
int f[202][202];
int g[202][202];
int p[202][202];
bool u[202];

int main() {
    int n, m, i, j, k, S, T, N;

// Ввод и построение графа и потока

    scanf("%d %d", &n, &m);
    for (i = 0; i < n; ++i) scanf("%d %d %d", &Building[i].x, &Building[i].y,
&Building[i].c);
    for (i = 0; i < m; ++i) scanf("%d %d %d", &Shelter[i].x, &Shelter[i].y, &Shelter[i].c);

    N = (T = (S = n + m) + 1) + 1;

    for (i = 0; i < n; ++i) for (j = 0; j < m; ++j) {
        scanf("%d", &f[i][n + j]);
        f[n + j][T] += f[i][n + j];
        c[i][j] = 1 + labs(Building[i].x - Shelter[j].x) + labs(Building[i].y -
Shelter[j].y);
    }
    for (i = 0; i < N; ++i) for (j = 0; j < N; ++j) {
        g[i][j] = INF; p[i][j] = i;
    }
    for (i = 0; i < n; ++i) for (j = 0; j < m; ++j) {
        if (f[i][n + j] > 0) g[n + j][i] = -c[i][j];
        g[i][n + j] = c[i][j];
    }
    for (i = 0; i < n; ++i) g[i][S] = 0;
    for (i = 0; i < m; ++i) {
        if (f[n + i][T] < Shelter[i].c) g[n + i][T] = 0;
        if (f[n + i][T] > 0) g[T][n + i] = 0;
    }

    for (i = 0; i < N; ++i) g[i][i] = 0;

// Поиск отрицательных циклов
    for (k = 0; k < N; ++k)
        for (i = 0; i < N; ++i) if (g[i][k] < INF)
            for (j = 0; j < N; ++j) if (g[k][j] < INF)
                if (g[i][j] > g[i][k] + g[k][j]) {
                    g[i][j] = g[i][k] + g[k][j];
                    p[i][j] = p[k][j];
                }
```

```
    for (i = 0; i < N; ++i) if (g[i][i] < 0) break;
    if (i < N) {
// Если нашли, то уменьшаем стоимость
        printf("SUBOPTIMAL\n");
        j = i;
        while (!u[j]) {
            u[j] = true;
            j = p[i][j];
        }
        i = j;
        do {
            ++f[p[i][j]][j];
            --f[j][p[i][j]];
            j = p[i][j];
        } while (j != i);
        for (i = 0; i < n; ++i, putchar('\n')) for (j = 0; j < m; ++j) printf("%d ", f[i][n
+ j]);
    } else printf("OPTIMAL\n");
    return 0;
}
```

Задача 47. Team them up! [NEERC 2001].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Your task is to divide a number of persons into two teams, in such a way, that:

- everyone belongs to one of the teams;
- every team has at least one member;
- every person in the team knows every other person in his team;
- teams are as close in their sizes as possible.

This task may have many solutions. You are to find and output any solution, or to report that the solution does not exist.

Входные данные:

For simplicity, all persons are assigned a unique integer identifier from 1 to N.

The first line in the input file contains a single integer number N ($2 \le N \le 100$) - the total number of persons to divide into teams, followed by N lines - one line per person in ascending order of their identifiers. Each line contains the list of distinct numbers Aij ($1 \le A_{ij} \le N$, $A_{ij} \ne i$) separated by spaces. The list represents identifiers of persons that ith person knows. The list is terminated by 0.

Выходные данные:

If the solution to the problem does not exist, then write a single message "No solution" (without quotes) to the output file. Otherwise write a solution on two lines. On the first line of the output file write the number of persons in the first team, followed by the identifiers of persons in the first team, placing one space before each identifier. On the second line describe the second team in the same way. You may write teams and identifiers of persons in a team in any order.

Пример:

input.txt	output.txt
5 3 4 5 0 1 3 5 0 2 1 4 5 0 2 3 5 0 1 2 3 4 0	No solution
5 2 3 5 0 1 4 5 3 0 1 2 5 0 1 2 3 0 4 3 2 1 0	3 1 3 5 2 2 4

Комментарий.

Составим граф из N вершин, в котором вершины i и j соединены, если i знает j и j знает i. Нам нужно разбить этот граф на две доли так, чтобы в каждой доле была клика (полный подграф). Инвертирум граф по ребрам: если существовало ребро (i, j) удалим его, если нет — создадим. Теперь исходная задача формулируется так: разбить

граф на две доли так, чтобы все ребра шли только из одной доли в другую, то есть сделать двудольный граф. Получили стандартную задачу, решаемую обходом в ширину. После ее решения может получиться несколько компонент связности. Их нужно будет оптимально расположить по долям.

Программа.

```
#include <stdio.h>

bool g[100][100];
int a[100];
int q[100];
int c[100];
int d[100][2];
int e[100][2][100];
int u[100];

int main() {
    int n, i, j, k, v, h, t, c1, c2, dc;

// Ввод и построение графа
    scanf("%d", &n);
    for (i = 0; i < n; i++) for (j = 0; j < n; j++) g[i][j] = false;
    for (i = 0; i < n; i++) {
        c[i] = 0;
        scanf("%d", &j);
        while (j != 0) {
            c[i]++;
            g[i][j - 1] = true;
            scanf("%d", &j);
        }
    }
    for (i = 0; i < n; i++) a[i] = -1;
    dc = 0;

// Рабиение на две доли
    for (j = 0; j < n; j++) if ((a[j] == -1) && (c[j] < n - 1)) {
        d[dc][0] = 1; d[dc][1] = 0;
        e[dc][0][0] = j;
        q[0] = j; h = 1; t = 0; a[j] = 0;
        while (t < h) {
            v = q[t++];
            for (i = 0; i < n; i++) if ((i != v) && (!g[v][i] || !g[i][v])) {
                if (a[i] == -1) {
                    a[i] = 1 - a[v];
                    e[dc][a[i]][d[dc][a[i]]++] = i;
                    q[h++] = i;
                } else if (a[i] != 1 - a[v]) {
                    printf("No solution\n");
                    return 0;
                }
            }
        }
        dc++;
    }

// Распределение компонент связанности
    c1 = c2 = 0;
    for (i = 0; i < dc; i++) {
        if (c1 < c2) {
            if (d[i][0] < d[i][1]) {
                c1 += d[i][1];
                c2 += d[i][0];
                u[i] = 1;
            } else {
                c1 += d[i][0];
                c2 += d[i][1];
                u[i] = 0;
            }
        } else {
            if (d[i][0] < d[i][1]) {
                c1 += d[i][0];
                c2 += d[i][1];
```

```
                u[i] = 0;
            } else {
                c1 += d[i][1];
                c2 += d[i][0];
                u[i] = 1;
            }
        }
    }
// Определение, кто из людей в какой доле
    for (i = 0; i < dc; i++)
        if (u[i] == 0) {
            for (j = 0; j < d[i][0]; j++) a[e[i][0][j]] = 0;
            for (j = 0; j < d[i][1]; j++) a[e[i][1][j]] = 1;
        } else {
            for (j = 0; j < d[i][1]; j++) a[e[i][1][j]] = 0;
            for (j = 0; j < d[i][0]; j++) a[e[i][0][j]] = 1;
        }
    for (i = 0; (i < n) && (c1 < (n + 1) / 2); i++) if (a[i] == -1) {
        a[i] = 0;
        c1++;
    }
    for (; i < n; i++) if (a[i] == -1) {
        a[i] = 1;
        c2++;
    }
// Вывод
    printf("%d", c1); for (i = 0; i < n; i++) if (a[i] == 0) printf(" %d", i + 1);
    putchar('\n');
    printf("%d", c2); for (i = 0; i < n; i++) if (a[i] == 1) printf(" %d", i + 1);
    putchar('\n');
    return 0;
}
```

Задача 48. Game [NEERC 1997].

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

A child has drawn N (N<=100) numbered circles of different colors. He has connected some of the circles by colored oriented line segments. Every pair of circles may have any number of segments, with any colors, connecting them. Each color (either circle color or segment color) is assigned its own unique positive integer number not greater than 100.

Starting the game the child first of all chooses three different integers L, K and Q within the range between 1 and N. Then he places one pawn into the circle number L and another one into the circle number K, whereupon he begins to move them using the following rules:

- a pawn can be moved along the line segment, if this segment has the same color with the circle where another pawn is placed,
- the pawn can be moved only in the direction of the segment (all segments are oriented),
- two pawns can never be placed in the same circle at the same time,
- the move order is free (i.e. it is not necessary to move pawns alternately),
- the game ends, when one of the pawns (any of the two) reaches the last circle number Q.

You are to write a program to find out the shortest (i.e. containing a minimal number of moves) solution for this game, if it exists.

Входные данные:

The first line of the input file contains integers N, L, K, Q separated by spaces. The second line consists of N integers c_1, c_2, ... , c_n, separated by spaces, in the given order, where ci is the color of the circle number i. The third line consists of a single integer M (0<=M<=10000) denoting the total number of segments. Then follow M lines, each containing a description of one oriented segment. Each segment is described by three integer numbers A_j, B_j, C_j, separated by spaces, where A and B are the numbers of the circles connected by the j-th segment with direction from A_j to B_j, and Cj represents the color of this segment.

Выходные данные:

The first line of the output file should contain the word "YES", if the game can come to the end, and "NO" otherwise (without quotes). If the answer is "YES", the second line of the output should contain just a single integer - the minimum number of the moves the child should make to finish the game.

Пример:

input.txt	output.txt
5 3 4 1 2 3 2 1 4 8 2 1 2 4 1 5 4 5 2 5 1 3 3 2 2 3 2 4 5 3 1 3 5 1	YES 3

Комментарий.

Из исходного графа сделаем граф состояний – каждая вершина нового графа будет соответствовать некоторому состоянию игры на старом графе (то есть положению фигур). В построенном графе можно отметить конечные состояния (когда одна из фигур на Q) и начальное состояние. После этого нужно сделать обход в ширину из начального состояния и при достижении любого конечного состояния следует остановиться и вывести результат.

Программа.

```
#include <stdio.h>
#include <stdlib.h>

// Цвета кругов
int c[200];
// Исходный граф
int e[10000][3];
// Граф состояний
int g[2000000][2];
// Конечные состояния
bool f[10000];
// Расстояние до вершины при обходе в ширину
int a[10000];
// Очередь для обхода в ширину
int Q[10000];

int cmp(const void* a, const void* b) {
    return ((int*)a)[2] - ((int*)b)[2];
}

int cmp2(const void* a, const void* b) {
    if (*(int*)a == *(int*)b) return ((int*)a)[1] - ((int*)b)[1];
    return *(int*)a - *(int*)b;
}

int main() {
    int n, m, l, k, q;
    int i, j, h, t, d, v;
    int r, s;
    int fin;

// Ввод и построение исходного графа
    scanf("%d %d %d %d", &n, &l, &k, &q);
    l--; k--; q--;
    s = l * 100 + k;
    for (i = 0; i < n; i++) scanf("%d", &c[i]);
    scanf("%d", &m);
    for (i = 0; i < m; i++) {
        scanf("%d %d %d", &e[i][0], &e[i][1], &e[i][2]);
        e[i][0]--; e[i][1]--;
    }
    qsort(e, m, sizeof(e[0]), cmp);
    for (i = 0; i < n * n; i++) f[i] = false;
    j = d = 0;
// Построение графа состояний
    for (i = 0; i < n; i++) {
        l = 0; r = m;
        while (l < r) {
            t = (l + r) >> 1;
```

```
            if (c[i] < e[t][2]) r = t;
            else if (c[i] > e[t][2]) l = t + 1;
            else l = r = t;
        }
        if (l == m) continue;
        while ((t >= 0) && (e[t][2] == c[i])) t--;
        t++;
        while ((t < m) && (e[t][2] == c[i])) {
            if ((i != e[t][0]) && (i != e[t][1])) {
                g[d][0] = i * 100 + e[t][0];
                g[d++][1] = i * 100 + e[t][1];
                g[d][0] = e[t][0] * 100 + i;
                g[d++][1] = e[t][1] * 100 + i;
                if (e[t][1] == q) f[i * 100 + e[t][1]] = f[e[t][1] * 100 + i] = true;
            }
            t++;
        }
    }
    qsort(g, d, sizeof(g[0]), cmp2);
    for (i = 0; i < 10000; i++) a[i] = -1;
    Q[0] = s;
    a[s] = 0;
    t = 0; h = 1;
    fin = -1;
// Обход в ширину в графе состояний
    while ((t < h) && (fin == -1)) {
        v = Q[t++];
        l = 0; r = d;
        while (l < r) {
            k = (l + r) >> 1;
            if (v < g[k][0]) r = k;
            else if (v > g[k][0]) l = k + 1;
            else l = r = k;
        }
        if (l == d) continue;
        i = l;
        while ((i >= 0) && (g[i][0] == v)) i--;
        i++;
        while ((i < d) && (g[i][0] == v)) {
            if (a[g[i][1]] == -1) {
                if (f[g[i][1]]) {
                    fin = a[v] + 1;
                    break;
                }
                a[g[i][1]] = a[v] + 1;
                Q[h++] = g[i][1];
            }
            i++;
        }
    }
// Вывод
    if (fin != -1) printf("YES\n%d\n", fin); else printf("NO\n");
    return 0;
}
```

Задача 49. The dog task [NEERC 1998].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Hunter Bob often walks with his dog Ralph. Bob walks with a constant speed and his route is a polygonal line (possibly self-intersecting) whose vertices are specified by N pairs of integers (X_i, Y_i) – their Cartesian coordinates.
Ralph walks on his own way but always meets his master at the specified N points. The dog starts his journey simultaneously with Bob at the point (X_1, Y_1) and finishes it also simultaneously with Bob at the point (X_N, Y_N).
Ralph can travel at a speed that is up to two times greater than his master's speed. While Bob travels in a straight line from one point to another the cheerful dog seeks trees, bushes, hummocks and all other kinds of interesting places of the local landscape which are specified by M pairs of integers (X_i', Y_i'). However, after leaving his master at the point (X_i, Y_i) (where $1 \le i < N$) the dog visits at most one interesting place before meeting his master again at the point (X_{i+1}, Y_{i+1}).

Your task is to find the dog's route, which meets the above requirements and allows him to visit the maximal possible number of interesting places. The answer should be presented as a polygonal line that represents Ralph's route. The vertices

of this route should be all points (X_i, Y_i) and the maximal number of interesting places (X_i', Y_i'). The latter should be visited (i.e. listed in the route description) at most once.

Входные данные:

The first line of the input file contains two integers N and M, separated by a space ($2 \le N \le 100$, $0 \le M \le 100$). The second line contains N pairs of integers X_1, Y_1, ..., X_N, Y_N, separated by spaces, that represent Bob's route. The third line contains M pairs of integers X_1', Y_1', ..., X_M', Y_M', separated by spaces, that represent interesting places.

All points in the input file are different and their coordinates are integers not greater than 1000 by the absolute value.

Выходные данные:

The first line of the output file should contain the single integer K – the number of vertices of the best dog's route. The second line should contain K pairs of coordinates X_1'', Y_1'', ..., X_K'', Y_K'', separated by spaces, that represent this route. If there are several such routes, then you may write any of them.

Пример:

input.txt	output.txt
4 5 1 4 5 7 5 2 -2 4 -4 -2 3 9 1 2 -1 3 8 –3	6 1 4 3 9 5 7 5 2 1 2 -2 4

Комментарий.

Составим двудольный граф: в левой доле будут отрезки пути, а в правой — интересные места. Для каждого отрезка определим все интересные места, которые могут быть из него достигнуты и проведем ребра от вершины, соответствующей отрезку к вершинам, соответствующим этим интересным местам. Построив в данном графе максимальное паросочетание, мы определим максимальное количество интерсных мест, которое может быть посещено. Остается только выписать путь, включив их.

Программа.

```
#include <stdio.h>
#include <math.h>

const int MAXN = 1000;
const int MAXM = 100000;
// Граф
int f[MAXN];
struct {
    int n, t;
} e[MAXM];
// Очередь и отметки о прохождении
int q[MAXN], u[MAXN];
// Вершина, соответствующая вершине из левой доли и из правой доли в паросочетании
int mx[MAXN], my[MAXN];
// Для восстановления пути
int p[MAXN];
// Исходные данные
struct {
    int x, y;
} pa[MAXN], pb[MAXN];

// Проверка, можно ли из заданного отрезка пути посетить заданную точку
bool check(__int64 x1, __int64 y1, __int64 x2, __int64 y2, __int64 x3, __int64 y3) {
    __int64 a, b, c;
    a = (x3 - x1) * (x3 - x1) + (y3 - y1) * (y3 - y1);
    b = (x3 - x2) * (x3 - x2) + (y3 - y2) * (y3 - y2);
    c = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);
    return (a + b - 4 * c) * (a + b - 4 * c) >= 4 * a * b;
}

int main() {
    int n, m, i, j, k, v, h, t;

// Ввод
    scanf("%d %d", &n, &m);
    for (i = 0; i < n; ++i) scanf("%d %d", &pa[i].x, &pa[i].y);
    for (i = 0; i < m; ++i) scanf("%d %d", &pb[i].x, &pb[i].y);
// Построение графа
    for (i = 0; i < n; ++i) f[i] = -1;
    k = 0;
```

```
    for (i = 0; i < n - 1; ++i) for (j = 0; j < m; ++j)
        if (check(pa[i].x, pa[i].y, pa[i + 1].x, pa[i + 1].y, pb[j].x, pb[j].y)) {
            e[k].n = f[i]; e[k].t = j; f[i] = k++;
        }
// Построение паросочетания
    for (i = 0; i < n - 1; ++i) mx[i] = -1;
    for (i = 0; i < m; ++i) my[i] = -1;
    for (i = 0; i < m; ++i) u[i] = -1;
    for (i = 0; i < n - 1; ++i) if (mx[i] == -1) {
        q[0] = i; h = 1; t = 0;
        while (t < h) {
            v = q[t++];
            for (j = f[v]; j != -1; j = e[j].n) if (u[e[j].t] != i) {
                u[e[j].t] = i;
                p[e[j].t] = v;
                if (my[e[j].t] == -1) {
                    j = e[j].t;
                    break;
                } else q[h++] = my[e[j].t];
            }
            if (j != -1) break;
        }
        while (j != -1) {
            v = mx[p[j]];
            mx[p[j]] = j;
            my[j] = p[j];
            j = v;
        }
    }
// Вывод
    j = n;
    for (i = 0; i < n - 1; ++i) if (mx[i] != -1) ++j;
    printf("%d\n", j);
    for (i = 0; i < n - 1; ++i) {
        printf("%d %d ", pa[i].x, pa[i].y);
        if (mx[i] != -1) printf("%d %d ", pb[mx[i]].x, pb[mx[i]].y);
    }
    printf("%d %d", pa[i].x, pa[i].y);
    return 0;
}
```

Задача 50. Highways [NEERC 1999].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

The island nation of Flatopia is perfectly flat. Unfortunately, Flatopia has a very poor system of public highways. The Flatopian government is aware of this problem and has already constructed a number of highways connecting some of the most important towns. However, there are still some towns that you can't reach via a highway. It is necessary to build more highways so that it will be possible to drive between any pair of towns without leaving the highway system.

Flatopian towns are numbered from 1 to N and town i has a position given by the Cartesian coordinates (x_i, y_i). Each highway connects exaclty two towns. All highways (both the original ones and the ones that are to be built) follow straight lines, and thus their length is equal to Cartesian distance between towns. All highways can be used in both directions. Highways can freely cross each other, but a driver can only switch between highways at a town that is located at the end of both highways.

The Flatopian government wants to minimize the cost of building new highways. However, they want to guarantee that every town is highway-reachable from every other town. Since Flatopia is so flat, the cost of a highway is always proportional to its length. Thus, the least expensive highway system will be the one that minimizes the total highways length.

Входные данные:

The input file consists of two parts. The first part describes all towns in the country, and the second part describes all of the highways that have already been built.

The first line of the input file contains a single integer N ($1 \le N \le 750$), representing the number of towns. The next N lines each contain two integers, x_i and y_i separated by a space. These values give the coordinates of i^{th} town (for i from 1 to N). Coordinates will have an absolute value no greater than 10000. Every town has a unique location.

The next line contains a single integer *M* ($0 \le M \le 1000$), representing the number of existing highways. The next *M* lines each contain a pair of integers separated by a space. These two integers give a pair of town numbers which are already connected by a highway. Each pair of towns is connected by at most one highway.

Выходные данные:

Write to the output file a single line for each new highway that should be built in order to connect all towns with minimal possible total length of new highways. Each highway should be presented by printing town numbers that this highway connects, separated by a space.

If no new highways need to be built (all towns are already connected), then the output file should be created but it should be empty.

Пример:

input.txt	output.txt
9 1 5 0 0 3 2 4 5 5 1 0 4 5 2 1 2 5 3 3 1 3 9 7 1 2 2 3	1 6 3 7 4 9 5 7 8 3

Комментарий.

Задача сводится к «достраиванию» минимального остовного дерева. Проще всего сделать это алгоритмом Прима.

Программа.

```
#include <stdio.h>
#include <stdlib.h>

const int INF = 2000000000;
// Координаты городов
struct {
    int x, y;
} t[750];
// Граф
bool g[750][750];
// Расстояния между городами
int d[750][750];
// Добавленные ребра
int p[750];
// Расстояние от уже построенного дерева до вершины
int w[750];
// Отметка о прохождении
bool u[750];
// Очередь
int q[750];

int main() {
    int n, m, k, ec;
    int i, j;
    int a, b, c;
    int h, t, v;

// Ввод и инициализация
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d %d", &::t[i].x, &::t[i].y);
    for (i = 0; i < n; i++) for (j = 0; j < n; j++) g[i][j] = false;
    for (i = 0; i < n; i++) {
        d[i][i] = 0;
```

```
        for (j = i + 1; j < n; j++)
            d[i][j] = d[j][i] = (::t[i].x - ::t[j].x) * (::t[i].x - ::t[j].x) +
                                (::t[i].y - ::t[j].y) * (::t[i].y - ::t[j].y);
    }
    scanf("%d", &m);
    k = 0;
    for (i = 0; i < m; i++) {
        scanf("%d %d", &a, &b);
        a--; b--;
        g[a][b] = g[b][a] = true;
    }
    for (i = 0; i < n; ++i) w[i] = d[0][i];
    for (i = 0; i < n; ++i) u[i] = false;
    q[0] = 0; h = 1; t = 0; u[0] = true;
    while (t < h) {
        v = q[t++];
        for (i = 0; i < n; ++i) if (!u[i] && g[v][i]) {
            u[i] = true;
            q[h++] = i;
        }
        for (i = 0; i < n; ++i) if (!u[i])
            if (w[i] > d[v][i]) {
                w[i] = d[v][i];
                p[i] = v;
            }
    }
// «Достраивание» остовного дерева и вывод
    while (true) {
        j = -1;
        for (i = 0; i < n; ++i) if (!u[i] && ((j == -1) || (w[i] < w[j]))) j = i;
        if (j == -1) break;
        printf("%d %d\n", p[j] + 1, j + 1);
        q[0] = j; h = 1; t = 0; u[j] = true;
        while (t < h) {
            v = q[t++];
            for (i = 0; i < n; ++i) if (!u[i] && g[v][i]) {
                u[i] = true;
                q[h++] = i;
            }
            for (i = 0; i < n; ++i) if (!u[i])
                if (w[i] > d[v][i]) {
                    w[i] = d[v][i];
                    p[i] = v;
                }
        }
    }
    return 0;
}
```

Задача 51. Flip game [NEERC 2000].

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Flip game is played on a rectangular 4x4 field with two-sided pieces placed on each of its 16 squares. One side of each piece is white and the other one is black and each piece is lying either it's black or white side up. Each round you flip 3 to 5 pieces, thus changing the color of their upper side from black to white and vice versa. The pieces to be flipped are chosen every round according to the following rules:

1. Choose any one of the 16 pieces.

2. Flip the chosen piece and also all adjacent pieces to the left, to the right, to the top, and to the bottom of the chosen piece (if there are any).

The goal of the game is to flip either all pieces white side up or all pieces black side up. You are to write a program that will search for the minimum number of rounds needed to achieve this goal.

Входные данные:

The input file consists of 4 lines with 4 characters "w" or "b" each that denote game field position.

Выходные данные:

Write to the output file a single integer number - the minimum number of rounds needed to achieve the goal of the game from the given position. If the goal is initially achieved, then write 0. If it's impossible to achieve the goal, then write the word "Impossible" (without quotes).

Пример:

input.txt	output.txt
bwbw wwww bbwb bwwb	Impossible
bwwb bbwb bwwb bwww	4

Комментарий.

Количество всевозможных состояний в игре равно $2^{16} = 65536$. Из каждого состояния можно перейти в 16 других. Видно, что полученные числа небольшие, следовательно можно составить граф состояний, в котором каждая вершина соответствует состоянию в игре, а ребро между двумя вершинами существует, если из одного состояния можно перейти в другое за один ход. В данном графе можно отметить начальную и конечную вершины и найти кратчайший путь между ними, длина которого и будет являться ответом. Так как веса всех ребер равны 1, то путь можно находить поиском в ширину.

Программа.

```
#include <stdio.h>

// Расстояние до вершин
int d[1 << 16];
// Очередь для обхода в ширину
int queue[1 << 16];

// Перейти из состояния a по ребру i
int flip(int a, int i) {
  int b = a;
  if (b & (1 << i)) b = b & ~(1 << i);
  else b = b | (1 << i);
  if (i > 3) {
    if (b & (1 << (i - 4))) b = b & ~(1 << (i - 4));
    else b = b | (1 << (i - 4));
  }
  if (i < 12) {
    if (b & (1 << (i + 4))) b = b & ~(1 << (i + 4));
    else b = b | (1 << (i + 4));
  }
  if (i % 4 > 0) {
    if (b & (1 << (i - 1))) b = b & ~(1 << (i - 1));
    else b = b | (1 << (i - 1));
  }
  if (i % 4 < 3) {
    if (b & (1 << (i + 1))) b = b & ~(1 << (i + 1));
    else b = b | (1 << (i + 1));
  }
  return b;
}

int main() {
  int a, i, j, head, tail, c, curr, b;
  char s[10];

// Ввод
  a = 0;
  for (i = 0; i < 4; i++) {
    scanf("%s", s);
    for (j = 0; j < 4; j++)
      a = (a << 1) + (s[j] == 'b');
  }
// Обход в ширину
  for (i = 0; i < (1 << 16); i++) d[i] = -1;
```

```
  d[a] = 0;
  queue[0] = a;
  head = 1;
  tail = 0;
  while ((tail < head) && (d[0] == -1) && (d[(1 << 16) - 1] == -1)) {
    curr = queue[tail++];
    for (i = 0; i < 16; i++) {
      b = flip(curr, i);
      if (d[b] == -1) {
        queue[head++] = b;
        d[b] = d[curr] + 1;
      }
    }
  }
// Вывод
  if (d[0] != -1) printf("%d\n", d[0]);
  else
  if (d[(1 << 16) - 1] != -1) printf("%d\n", d[(1 << 16) - 1]);
  else printf("Impossible\n");
  return 0;
}
```

ГЛАВА 7. КОМБИНАТОРИКА, ТЕОРИЯ ЧИСЕЛ

Задача 52. Одноцветные треугольники [ЛКШ'2004]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

В пространстве расположено N попарно различных точек. Никакие три из этих точек не лежат на одной прямой. Каждая пара точек соединена отрезком. Каждая тройка точек вместе с отрезками, их соединяющими, образует треугольник.

Вам необходимо раскрасить каждый из этих отрезков в один из двух цветов – черный или белый. При этом для каждой точки известно, какое количество черных отрезков из нее должно выходить. Ваша задача – максимизировать количество получившихся одноцветных треугольников. Треугольник называется одноцветным, если все три отрезка в нем окрашены в один и тот же цвет.

Задание. Напишите программу, которая:

- вводит из входного файла количество точек N и количество черных отрезков, выходящих из каждой точки;
- определяет максимальное количество одноцветных треугольников, которое можно получить;
- выводит результат в выходной файл.

Входные данные. Первая строка входного файла содержит целое число N ($1 \le N \le 32768$) – количество точек. В каждой из следующих N строк содержится ровно одно число D_i ($0 \le D_i \le N-1$) – количество черных отрезков, которые должны выходить из i-й точки. Гарантируется, что существует хотя бы одна раскраска отрезков, удовлетворяющая указанным требованиям.

Выходные данные. Единственная строка выходного файла должна содержать одно число, равное максимальному количеству одноцветных треугольников.

Пример.

input.txt	output.txt
6 3 3 3 3 3 3	2

Комментарий. Будем считать количество неодноцветных треугольников. Заметим, что всего у нас существует $N * (N-1) * (N-2) / 6$ различных треугольников с вершинами в заданных точках. Поэтому, посчитав количество неодносветных треугольников, будет достаточно вычесть это количество из вышеприведенной формулы.

Зафиксируем любую точку. Пусть из нее выходят k_i отрезков. Тогда, очевидно, существуют ровно $k_i * (N - 1 - k_i)$ неодноцветных треугольников, с вершиной в этой точке. Однако если просто просуммировать данное выражение по всем точкам, то некоторые треугольники будут посчитаны несколько раз.

Мы утверждаем, что любой неодноцветный треугольник будет посчитан ровно 2 раза.

Действительно, у любого неодноцветного треугольника ровно 2 вершины, в которых сходятся отрезки разных цветов. Поэтому любой неодноцветный треугольник будет посчитан ровно 2 раза.

Таким образом, просуммировав $k_i * (N - 1 - k_i)$, мы получим удвоенное количество неодноцветных треугольников. То есть для того чтобы получить ответ к задаче, небоходимо вычесть полученное значение, деленное на два, из общего еоличество треугольников.

Как видно, время работы алгоритма составляет $O(N)$.

Программа. // не требует комментариев

```
#include <stdio.h>

typedef __int64 i64;

int main() {
	int n, i;
	i64 s;
	scanf("%d", &n);
	for (s = i = 0; i < n; i++) {
		int z;
```

```
            scanf("%d", &z);
            s += (n - 1 - z) * z;
        }
        s = (i64)n * (i64)(n - 1) * (i64)(n - 2) / 6 - (s >> 1);
        printf("%I64d\n", s);
        return 0;
}
```

Задача 53. Скобки [ЛКШ'2004]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**16 мегабайт**

Найдите количество правильных скобочных последовательностей заданной длины.

Входные данные. В первой строке записано число N – длина скобочной последовательности, не превосходящее 20.

Выходные данные. Выведите искомое число.

Пример.

input.txt	output.txt
2	1

Комментарий. Для начала заметим, что если N нечетное, то правильных скобочных последовательностей длины N не существует. Для четных N эта задача предполагает использование формулы для чисел Каталана [12]: T(2N) = C(2N, N) / (N+1), где C(N, K) – количество сочетаний из N элементов по K.

Программа.

```
#include <stdio.h>

typedef __int64 i64;

int main() {
        int n, i;
        i64 r;
        scanf("%d", &n); // читаем ввод
        if (n & 1) { // n может ненарочно оказаться нечетным - учтем это
                putchar('0');
                return 0;
        }
        n >>= 1; // делим n пополам
        for (r = 1, i = 1; i < n; i++) { // домножаем и делим
                                         // на очередные множители числителя и знаменателя
                r *= 2 * (2 * i + 1);
                r /= (i + 2);
        }
        printf("%I64d\n", r);
        return 0;
}
```

Задача 54. Нулевые степени [Trivial testing system – http://acm.math.spbu.ru/tts]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**64 мегабайта**

Коля любит теорию чисел. Особенно ему нравится изучать свойства операций, производимых по различным модулям. Он знает, что числа по модулю можно складывать, вычитать, умножать и иногда даже делить.Последняя операция не всегда возможна. А именно – если какие-либо числа, отличные от нулей, по модулю m в различных степенях дают ноль, то деление может быть невозможным!

Колю заинтересовал этот вопрос, и он хочет выяснить, сколько существует различных вычетов по модулю m (т.е. различных чисел в пределах от 1 до m), которые при возведении в степень k дают ноль по модулю m.

Входные данные. Во входном файле содержатся два целых числа m и k, удовлетворяющие ограничениям $1 \le m,k \le 10^9$.

Выходные данные. В выходной файл необходимо вывести количество таких целых i в пределах от 1 до m, что $(i^k) \bmod m = 0$.

Пример.

input.txt	output.txt
4 2	2

Комментарий. Проще говоря, число i^k должно делиться на m. Поэтому число i должно содержать все простые сомножители, которые содержит m. Найдем минимальное такое i_0: очевидно, оно будет состоять только из тех простых сомножителей, входящих в m. Для этого разложим m на простые множители и для каждого сомножителя p, входящего в m в степени d, вычислим минимальную степень d_0, в которой этот множитель должен входить в i_0, чтобы выполнялось $d_0 * k \ge d$: $d_0 = ceil(d / k)$, где ceil(x) – наименьшее целое число, не меньшее x. То есть i_0 будет произведением p^{d0}.

Далее ясно, чтолюбое i, удовлетворяющее условиям задачи делится на i_0. Поэтому нам остается найти количество чисел между i_0 и m, которые делятся на i_0. Это количество равно $[m / i_0]$, где [x] – целая часть числа x.

Программа.

```
#include <stdio.h>
#define MAXC 100000
#define MAXP 50000
#define MAXD 35

char cc[MAXC];
int p[MAXP], a[MAXD], b[MAXD];

int main() {
	int m, k, c, i, x, c0;
	// решето Эратосфена
	// находим все простые числа от 2 до MAXC - 1
	p[0] = 2; c = 1;
	for (i = 3; i < MAXC; i += 2)
		if (!cc[i]) {
			int j;
			for (j = i << 1; j < MAXC; j += i)
				cc[j] = 1;
			p[c++] = i;
		}
	scanf("%d %d", &m, &k); // читаем m и k
	// разлагаем m на множители
	// c0 – количество различных сомножителей в разложении
	// a – сами сомножители
	// b – степени, в которых сомножители входят в разложение
	for (x = m, c0 = i = 0; i < c && p[i] * p[i] <= x; i++)
		if (!(x % p[i])) { // x делится на p[i]
			int k = 0;
			do k++, x /= p[i]; while (!(x % p[i]));
			a[c0] = p[i]; b[c0++] = k;
		}
	if (x > 1)
		a[c0] = x, b[c0++] = 1;
	// по всем сомножителям
	for (x = 1, i = 0; i < c0; i++) {
		int h = b[i] / k + (b[i] % k != 0); // k = ceil(b[i] / k)
		while (h--)
			x *= a[i]; // добавляем нужное количество сомножителей в x
	}
	printf("%d\n", m / x); // выводим ответ
	return 0;
}
```

Задача 55. Марсианские факториалы [Всероссийская командная олимпиада школьников'2000]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 1 секунда
Ограничение по памяти 64 мегабайта

В 3141 году очередная экспедиция на Марс обнаружила в одной из пещер таинственные знаки. Они однозначно доказывали существование на Марсе разумных существ. Однако смысл этих таинственных знаков долгое время оставался неизвестным. Недавно один из ученых, профессор Очень-Умный, заметил один интересный факт: всего в надписях, составленных из этих знаков, встречается ровно K различных символов. Более того, все надписи заканчиваются на длинную последовательность одних и тех же символов.

Вывод, который сделал из своих наблюдений профессор, потряс всех ученых Земли. Он предположил, что эти надписи являются записями факториалов различных натуральных чисел в системе счисления с основанием K. А символы в конце – это конечно же нули – ведь, как известно, факториалы больших чисел заканчиваются большим количеством нулей. Например, в нашей десятичной системе счисления факториалы заканчиваются на нули начиная с 5!=1×2×3×4×5=120. А у числа 100! в конце следует 24 нуля в десятичной системе счисления и 48 нулей в системе счисления с основанием 6 – так что у предположения профессора есть разумные основания!

Теперь ученым срочно нужна программа, которая по заданным числам N и K найдет количество нулей в конце записи в системе счисления с основанием K числа N!, чтобы они могли проверить свою гипотезу. Вам придется написать им такую программу!

Входные данные. На первой строке входного файла находятся числа *N* и *K*, разделенные пробелом. ($1 \le N \le 10^9$, $2 \le K \le 1000$).

Выходные данные. Выведите в выходной файл число *X* - количество нулей в конце записи числа *N*! в системе счисления с основанием K.

Пример.

input.txt	output.txt
5 10	1
100 10	24
100 6	48
3 10	0

Комментарий[Разбор задач Всероссийской командной олимпиады школьников'2000]. Для решения этой задачи достаточно заметить, что для того, чтобы число A заканчивалось на X нулей в системе счисления с основанием K необходимо и достаточно, чтобы оно делилось без остатка на KX. Действительно:
$A = A_P K^P + A_{P-1} K^{P-1} + \ldots + A_0 K^0$, если оно заканчивается на X нулей, то $A_i = 0$ для . Значит, $A = A_P K^P + A_{P-1} K^{P-1} +$

Значит задача свелась к тому, чтобы определить, на какую максимальную степень K делится число N!. Поскольку N! может быть очень велико, непосредственное его вычисление с целью такой проверки невозможно.

[illegible] $P_i^{b_i}$ [illegible]

Для этого применим следующие соображения: количество чисел, кратных P и не превышающих N равно $\left\lfloor \frac{N}{P} \right\rfloor$. Каждое из этих чисел даст по одному простому множетелю P в N!. Но кроме того, $\left\lfloor \frac{N}{P^2} \right\rfloor$ чисел дадут еще по одному P, $\left\lfloor \frac{N}{P^3} \right\rfloor$ - еще по одному и т. д. Значит, $b_i = \left\lfloor \frac{N}{P_i} \right\rfloor + \left\lfloor \frac{N}{P_i^2} \right\rfloor + \left\lfloor \frac{N}{P_i^3} \right\rfloor + \cdots$ Заметим, что суммирование можно остановить, когда очередное слагаемое равно 0.

Фрагмент программы, который находит простые числа, не превышающие *K*:

```
p[1] := 2; { Первое простое число }
pn := 1;   { Количество уже найденных простых чисел }
i := 3;    { Текущий кандидат на то, чтобы стать простым числом }
while i <= k do
begin
  j := 1;
  f := true;
```

```
  { Для всех простых чисел, не превышающих √i , проверим, что i на них
    не делится }
  while (p[j] * p[j] <= i) do
  begin
    if i mod p[j] = 0 then
    begin
      f := false;
      break;
    end;
    inc(j);
  end;
  if f then
  begin
    { Добавляем очередное простое число }
    inc(pn);
    p[pn] := i;
  end;
  { Следующий кандидат }
  inc(i, 2);
end;
```

Функция, которая находит максимальную степень простого числа *M*, на которую делится *N!*:

```
function find(n, m: longint): longint;
var
  r: longint;
begin
  r := 0;
  while n > 0 do
  begin
    r := r + n div m;
    n := n div m;
  end;
  find := r;
end;
```

Фрагмент программы, который находит искомый минимум:

```
{ Заносим в массив b значения b_i }
for i := 1 to pn do
  b[i] := find(n, p[i]);

min := maxlongint;
{ Перебираем все простые числа }
for j := 1 to pn do
begin
  { l – количество вхождений p[i] в разложение k }
  l := 0;
  while k mod p[j] = 0 do
  begin
    k := k div p[j];
    inc(l);
  end;
  { Если p[i] присутствует в разложении, обновим по необходимости
    минимум }
  if l > 0 then
  begin
    if b[j] div l < min then
      min := b[j] div l;
  end;
end;
```

Задача 56. Контрольный блок [Всероссийская командная олимпиада школьников'2001]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**64 мегабайта**

Фирма *Macrohard* разработала новый протокол обмена данными по сети. Каждый блок данных при этом обмене состоит из *N* чисел в диапазоне от 0 до *M*-1 включительно. Чтобы повысить надежность передачи, вместе с блоком данных пересылается контрольный блок такой же длины.

Предположим, что исходный блок состоит из чисел *a*1, *a*2, ..., *a*N. Тогда, контрольный блок состоит из чисел *b*1, *b*2, ..., *b*N, из диапазона от 0 до *M*-1 включительно, таких что выполняются следующие равенства: *b*1 = (*aN* + *bN*) mod *M*, *b*2 = (*a*1 + *b*1) mod *M*, *b*3 = (*a*2 + *b*2) mod *M*, ..., *bN* = (*aN*-1 + *bN*-1) mod *M*. (обозначение *X* mod *M* означает остаток от деления *X* на *M*, например, 7 mod 4 = 3, 6 mod 2 = 0).

Блоки данных, для которых нельзя построить контрольный блок, удовлетворяющий указанному свойству, считаются подозрительными и их передача по сети не разрешается.

Ваня хочет поступить на работу программистом в фирму *Macrohard*, и в качестве вступительного задания ему поручили написать процедуру построения контрольного блока для заданного блока данных. Помогите ему!

Входные данные. Первая строка входного файла содержит числа N и M ($1 \le N \le 1000$, $2 \le M \le 10^9$). Следующая строка содержит блок данных, для которого следует построить контрольный блок, числа разделены пробелами.

Выходные данные. На первой строке выходного файла выведите YES, если для данного блока данных можно построить контрольный блок и NO, если нельзя. В случае, если контрольный блок построить можно, на второй строке выведите контрольный блок. Числа разделяйте пробелами. Если решений несколько, можно выдать любое из них.

Пример.

input.txt	output.txt
`4 3` `1 0 0 2`	`YES` `1 2 2 2`
`4 3` `1 1 1 1`	`NO`

Комментарий[Разбор задач Всероссийской командной олимпиады школьников'2000]. Прежде чем приступать к решению этой задачи, отметим одно свойство сложения по модулю: если $A = B \bmod M$, то $(A + C) \bmod M = (B + C) \bmod M$ (*).

Предположим, что для данного блока данных $a_1, a_2, \dots, a_N$ существует какой-либо контрольный блок $b_1, b_2, \dots, b_N$. Прибавим к каждому b_i число $(M - b_1)$ по модулю M. Получится новый блок данных той же длины , , ..., $c_N = (b_N + (M - b_1)) \bmod M$. Отметим, что $c_1 = 0$.

Благодаря свойству (*) набор $c_1, c_2, \dots, c_N$ также оказывается контрольным блоком для исходного блока данных $a_1, a_2, \dots, a_N$. Таким образом, получаем следующее утверждение: для того чтобы для некоторого блока данных $a_1, a_2, \dots, a_N$ существовал контрольный блок данных $b_1, b_2, \dots, b_N$ необходимо и достаточно, чтобы существовал контрольный блок с $b_1 = 0$.

Теперь решение задачи не представляет никакого труда: зная b_1, мы без труда из равенства находим b_2, зная b_2 — b_3, и т.д. Остается только после нахождения b_N проверить, что выполнено последнее равенство.

Отметим также факт, что для того, чтобы существовал контрольный блок, необходимо и достаточно, чтобы, в чем легко убедиться, сложив равенства ,,..., по модулю M.

Программа.

```
program control_block;
var
    a: array [0..1001] of longint;
    i, r, n, m: longint;
begin
    assign(input, 'input.txt'); reset(input);
    assign(output, 'output.txt'); rewrite(output);

    read(n, m);
    r := 0;
    for i := 1 to n do
    begin
        read(a[i]);
        r := (r + a[i]) mod m;
    end;

    if r = 0 then
    begin
```

```
        writeln('YES');
        for i := 1 to n do
        begin
            write(r, ' ');
            r := (r + a[i]) mod m;
        end;
    end else begin
        writeln('NO');
    end;

    close(input);
    close(output);
end.
```

Задача 57. Yellow code [Andrew Stankevich contest 5]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

Inspired by Gray code, professor John Yellow has invented his own code. Unlike in Gray code, in Yellow code the adjacent words have many different bits.

More precisely, for $s = 2^n$ we call the permutation $a_1, a_2, \ldots, a_s$ of all n-bit binary words the Yellow code if for all $1 \le k < s$ words a_k and a_{k+1} have at least [n/2] different bits and a_1 and a_s also have at least [n/2] different bits.

Given n you have to find the n-bit Yellow code or detect that there is none.

Входные данные. Input file contains the number n ($2 \le n \le 12$).

Выходные данные. Output 2n n-bit binary vectors in the order they appear in some n-bit Yellow code, one on a line. If there is no n-bit Yellow code, output “none” on the first line of the output file.

Пример.

input.txt	output.txt
4	0000 1111 0001 1110 0010 1101 0011 1100 0101 1011 0100 1010 0110 1000 0111 1001

Комментарий. Это задача на творческий поиск. Один из вариантов решения приведен ниже.

Программа.

```
#include <stdio.h>
#define MAXN 12

char u[1 << MAXN];

void outit(int a, int n) {
     int i;
     for (i = n - 1; i >= 0; i--)
          putchar('0' + ((a >> i) & 1));
     putchar('\n');
}

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

```
int i, c;
for (c = 0, i = n - 1; i >= 0; i--)
          if ((a >> i) & 1) c++;
return c;
}

int main() {
int n, i, w, p;
char f;
scanf("%d", &n);
w = (unsigned)~0 >> (32 - n); // маска с установленными младшими n битами
outit(0, n);
outit(p = w, n);
u[0] = u[w] = 1;
for (f = 0, i = 1; i <= w; i++)
          if (!u[i]) { // если мы это число еще не выводили
               int j = i ^ w; // берем его инверсию
               u[i] = u[j] = 1;
               if (get(p ^ (f ? j : i), n) < (n >> 1)) // выбираем то, которое согласуется с
                    f ^= 1;                            // предыдущим
               if (!f)
                    outit(i, n), outit(p = j, n);
               else
                    outit(j, n), outit(p = i, n);
          }
return 0;
}
```

Задача 58. Yet another digit [Andrew Stankevich contest 5]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**2 секунды**
Ограничение по памяти	**64 мегабайта**

When making different calculations we usually use decimal notation. On the other side, computers usually store and process information in binary form. Both decimal and binary notations are special cases of so called base notations. Base notation is the way to represent integer number as the sum of powers of some integer number B called the base of the notation, taken with some coefficients called digits. Thus the weight of the i-th digit is proportional to B^i (digits being numbered from zero).

Most base notations that we use have a nice property — for any particular number the value of each digit is defined unambiguously. For example, consider the number 5. In binary notation the rightmost digit is 1, the next one is 0, the second digit is 1 and all the others are 0.

However, this property just follows from the fact that the number of different values for digits is equal to the base of the notation. If we allow more different values for each digit, the uniqueness of representation would be lost. Consider redundant binary notation, where three digits are allowed: 0, 1 and 2. In this notation some particular number may have several representations. For example, 5 can be represented as both 101 and 21.

Find out the number of different representations of the given number R in redundant binary notation.

Входные данные. Input file contains one integer number R ($0 \le R \le 10^{100}$)..

Выходные данные. Output one integer number—the number of ways R can can be represented in redundant binary notation.

Пример.

input.txt	output.txt
5	2
7	1

Комментарий. Это задача на элементы комбинаторики с использованием «длинной» арифметики. Разбор деталей оставляется читателю.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXD 105
#define MAXN 400
#define B10 10
```

```
#define swap(a, b) (a ^= b, b ^= a, a ^= b)
#define B 10000
#define LF "%04d"
#define imax(a, b) (((a) >= (b)) ? (a) : (b))
#define pswap(a, b) (tp = a, a = b, b = tp)

typedef struct { // длинное число
     int c, d[MAXD];
} tlong;

void lget(tlong *a) { // чтение длинного числа
     int i, j;
     char c;
     do c = getchar(); while (c < '0' || c > '9');
     i = 0;
     do {
             a->d[i++] = c - '0';
             c = getchar();
} while (c >= '0' && c <= '9');
a->c = i--; j = 0;
while (j < i) {
             swap(a->d[i], a->d[j]);
             i--; j++;
}
for (i = a->c - 1; i > 0 && !a->d[i]; i--);
a->c = i + 1;
}

int lshr(tlong *a) { // деление пополам
     int i, c;
     for (c = 0, i = a->c - 1; i >= 0; i--)
     a->d[i] = (c = (c & 1) * B10 + a->d[i]) >> 1;
     for (i = a->c - 1; i > 0 && !a->d[i]; i--);
     a->c = i + 1;
     return c & 1;
}

void lmuls(tlong *a, int d) { // умножение длинного на короткое
     int i;
     for (i = a->c - 1; i >= 0; i--)
     a->d[i] *= d;
     for (a->c += 5, i = 0; i < a->c; i++)
     if (a->d[i] >= B) {
     int q = a->d[i] / B;
     a->d[i + 1] += q;
     a->d[i] -= q * B;
     }
     for (i = a->c - 1; i > 0 && !a->d[i]; i--);
     a->c = i + 1;
}

void ladd(tlong *a, tlong *b) {
     int i, c, m = imax(a->c, b->c);
     for (c = i = 0; i < m || c ; i++)
     if (c = ((a->d[i] += b->d[i] + c) >= B))
     a->d[i] -= B;
     while (i > 0 && !a->d[i]) i--;
     a->c = i + 1;
}

int s[MAXN];

int main() {
     int n, i, x;
     static tlong a, b, c, d;
     tlong *d0, *d1, *t0, *t1;
     freopen("digit.in", "r", stdin);
     freopen("digit.out", "w", stdout);
     lget(&a);
     n = 0;
     while (a.c > 1 || a.d[0])
     s[n++] = lshr(&a);
     memset(&a, 0, sizeof(tlong));
```

```
	d1 = &a; d0 = &b; t0 = &c; t1 = &d;
	d0->c = d1->c = d1->d[0] = 1;
	for (x = -1, i = 0; i < n; i++)
	if (s[i]) { // it's 1
		tlong *tp;
		static tlong z;
		// leave it
		*t1 = *d0;
		ladd(t1, d1);
		// try to move
		*t0 = *t1;
		lmuls(t0, i - x - 1);
		ladd(t0, d0);
		x = i;
		pswap(t0, d0);
		pswap(t1, d1);

		ladd(d0, d1);
	  }
	  printf("%d", d0->d[d0->c - 1]);
	  for (i = d0->c - 2; i >= 0; i--)
		printf(LF, d0->d[i]);
	  return 0;
}
```

ГЛАВА 8. СОРТИРОВКА И ПОИСК.

Задача 59. Найдись! [NEERC'2005, NORTHERN SUBREGION QF]

Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 1 секунда
Ограничение по памяти: 64 мегабайт

Задан список целых чисел. Требуется для заданного списка чисел определить, содержится ли каждое из этих чисел в списке.

Входные данные. Первая строка входного файла содержит целое число N и целое число K (1≤K, N≤10000). Во второй строке N чисел — элементы списка. В третей строке K чисел, представляющих запросы.

Выходные данные. Для каждого из K запросов выведите в отдельную строку файла YES, если число-запрос встречается в списке, и NO в противном случае.

Пример.

input.txt	output.txt
3 4 1 6 9 7 9 10 1	NO YES NO YES

Комментарий. Данная задача решается с помощью бинарного поиска. Следует обратить внимание на то, что входной список может быть не отсортирован. Поэтому сначала необходимо произвести сортировку списка, а потом уже обрабатывать запросы с помощью бинарного поиска. Сортировку следует производить с помощью алгоритма с временем работы *O(N log(N))*, такого как quicksort, например. В таком случае время на сортировку списка – *O(N log(N))*, на обработку запросов – *O(K log(N))*, что в сумме нам даст *O((N + K) log(N))*

```
#include <stdio.h>
#define MAXN 10000
#define swap(a, b) (a ^= b, b ^= a, a ^= b)

int a[MAXN];
// сортируем a
int partition(int l, int r) {
	int i = l - 1, j = r + 1, x = a[(l + r) >> 1];
	while (1) {
		do i++; while (a[i] < x);
		do j--; while (a[j] > x);
		if (i < j)
			swap(a[i], a[j]);
		else
			return j;
	}
}

void sort(int l, int r) {
	while (l < r) {
		int m = partition(l, r);
		sort(l, m);
		l = m + 1;
	}
}
// бинарный поиск
int search(int x, int n) {
	register int l = -1, r = n;
	while (r > l + 1) {
		int m = (l + r) >> 1;
		if (a[m] < x) l = m; else
		if (a[m] > x) r = m; else
			return m;
	}
	return -1;
}

int main() {
	int n, k, i;
```

```
	// читаем ввод
	scanf("%d %d", &n, &k);
	for (i = 0; i < n; i++)
		scanf("%d", a + i);
	// сортируем a
	sort(0, n - 1);
	// обрабатываем запросы
	while (k--) {
		int l = -1, r = n, x;
		scanf("%d", &x);
		puts(search(x, n) != -1 ? "YES" : "NO");
	}
	return 0;
}
```

Задача 60. Сумма двух

Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 1 секунда
Ограничение по памяти: 64 мегабайт

Задан массив целых чисел. Требуется для заданного числа определить, представимо ли оно в виде двух различных чисел из массива.

Входные данные. Первая строка входного файла содержит целое число N ($1 \le N \le 100000$). Во второй строке N чисел — элементы массива. В третей строке заданное число.

Выходные данные. Если заданое число можно представить в виде суммы двух различных элементов массива, выведите индексы этих элементов, разделенные пробелами. В противном случае выведите строку “IMPOSSIBLE”

Пример.

input.txt	output.txt
4 -5 1 6 9 7	2 3

Комментарий. Обозначим массив A, а заданное число – x. Если N=1(массив состоит из одного элемента), то ответ заведомо “Нет”. Заведем второй массив POS, где будем хранить позиции соответствующих элементов до в исходном массиве. Далее, отсортируем массив по неубыванию. При этом меняя местами A_i и A_j, будем также менять POS_i и POS_j.

Теперь, перебирая все элементы массива, для каждого A_i будем искать с помощью бинарного поиска элемент A_j равный $(x - A_i)$ такой, что POS_i не совпадает с POS_j (Чтобы это сделать, будем искать самое первое вхождение $(x-A_i)$ в отсортированный массив A. Если таковое имеется, но $POS_i=POS_j$, то следует увеличить j на единицу. Если теперь $A_i=A_j$, то нужный нам элемент найден). Если такой элемент есть, то ответом на задачу будут числа POS_i и POS_j. Если же поиск не был успешен ни для одного элемента массива, то, очевидно, представить x в виде суммы двух различных элементов A невозможно.

На сортировку алгоритм тратит $O(N \log(N))$, на каждый поиск – $O(\log(N))$. Учитывая, что всего поиск производится N раз, получает следующую оценку на время работы алгорита: $O(N \log(N))$.

Задача 61. Сортировка-2 [фольклор]
Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 1 секунда
Ограничение по памяти: 64 мегабайт

Задан список целых чисел. Требуется вывести список в отсортированном по невозрастанию виде.

Входные данные. Первая строка входного файла содержит целое число N – количество элементов списка ($1 \le N \le 1000000$). Во второй строке N чисел — элементы списка ($0 < A_i < 10^5$).

Выходные данные. Выведите список в отсортированном по неубыванию виде.

Пример.

input.txt	output.txt
5 1 3 5 3 1	1 1 3 3 5

Комментарий. Здесь в первую очередь следует обратить внимание на ограничения. Во-первых, при максимальном N предполагается, что алгоритмы со временем работы *O(N log(N))* не справятся с ограничениями по времени. Во-вторых, ограничения на сами элементы списка дают возможность придумать для этой задачи алгоритм со временем работы O(N).

Для каждого $0 < i < 10^5$ будем считать, сколько раз это число встречается в массиве. После этого просто выведем каждое число равно столько раз, сколько оно встретилось в массиве. Ниже приведен код на языке C++, реализующий эти операции:

```
for (i = 0; i < n; i++)
  c[a[i]]++;
for (i = 1; i <= 100000; I++)
  for (j = c[i]; j > 0; j--)
    cout << i << " ";
```

Очевидно, таким образом мы получаем время работы алгоритма *O(N)*.

Задача 62. Сортировка-3 [Бентли]
Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 1 секунда
Ограничение по памяти: 0.5 мегабайта

Задан список различных целых чисел. Требуется вывести список в отсортированном по невозрастанию виде.

Входные данные. Первая строка входного файла содержит целое число N – количество элементов списка ($1 \le N \le 1000000$). Во второй строке N чисел — элементы списка ($0 \le A_i < 10^6$).

Выходные данные. Выведите список в отсортированном по неубыванию виде.

Пример.

input.txt	output.txt
5 1 3 5 3 1	1 1 3 3 5

Комментарий. Эта задача решается с помощью того же метода, что и предыдущая. Однако если обратить внимание на ограничения по памяти, то станет понятно, что также просто она не решается, поскольку мало того, что мы не сможем завести в памяти массив C (*i*-й элемент которого хранит, сколько раз в массиве A встречается число *i*), мы даже прочитать в память весь массив A не в состоянии.

Заметим, что каждое число встречается в массиве A лишь один раз. Поэтому нам достаточно отметить данное число в массиве C, а потом вывести все помеченные числа. Будем отмечать не ячейки массива, а биты элементов массива C. Так как в одном байте 8 бит, нам понадобится максимум лишь $10^6/8 = 125000$ байт, что меньше 0.5 мегабайта. То есть массив C можно объявить как char C[125000] в C.

Теперь рассмотрим, как нам установить *i*-й бит (считая с нулевого) в массиве C. Каждый элемент C состоит из 8 бит, поэтому нужный нам бит находится в ячейке с номером $[i / 8]$, где [x] – целая часть числа x. В этой ячейке нужный нам бит имеет номер $i \bmod 8$, где $x \bmod m$ – остаток от деления числа x на число m. Тогда установка *i*-го

бита в массиве C будет выглядеть следующим образом: C[*i* / 8] |= 1 << (*i* % 8). Мы просто выравниваем единицу с нужной нам позицией и применяем операцию | (побитовое OR).

Те же идеи используются и для проверки того, установлен ли бит с номером *i*: ((C[*i* / 8] >> (*i* % 8)) & 1). Здесь мы сдвигаем нужную нам ячейку так, чтобы интересующий нас бит оказался на 0-й позиции, и применяем операцию & (побитовое AND), чтобы обнулить все не интересующие нас биты.

Ниже приведен код на языке C++, реализующий все вышесказанное:

```
char C[125000];
int i, x, n;

for (i = 0; i < n; i++) {
      cin >> x; // читаем очередное число
      C[x / 8] |= 1 << (x % 8); //устанавливаем в C бит с номером x
}
for (i = 0; i < 1000000; i++)
      if ((C[i / 8] >> (i % 8)) & 1) // проверяем, установлен ли бит i
        cout << i << " ";
```

Задача 63. Drying (Сушка) [NEERC'2005, NORTHERN SUBREGION QF]

Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 1 секунда
Ограничение по памяти: 64 мегабайт

Иногда сушить одежду зимой очень трудно. Но Джейн – очень умная девушка. Она не боится этого скучного процесса. Джейн решила использовать радиатор, чтобы сделать сушку быстрее. Но радиатор маленький, поэтому в любой момент времени он может сушить только одну вещь.

Джейн хочет высушить все вещи за минимально возможное время. Она просит вас написать программу, которая вычислит минимально возможное время, необходимое для сушки данного набора вещей.

Джейн только что постирала *N* вещей. Каждая из них вобрала в себя A_i единиц вожы во время стирки. Каждую минуту количество воды, которая содержится в каждой вещи, уменьшается на единицу (конечно, если вещь еще не сухая). Когда количество воды в вещи становится равным 0, вещь считается высохщей и готовой к упаковке.

Каждую минуту Джейн может выбрать одну вещь для сушки на радиаторе. Радиатор очень горячий, поэтому количество воды в вещи, лежащей на нем, уменьшается на *K* за минуту (понятно, что количество воды в вещи не может стать меньше 0).

Задача состоит в том, чтобы минимизировать суммарное время сушки, эффективно используя радиатор. Сушка заканчивается, когда все вещи сухие.

Входные данные. Первая строка входного файла содержит целое число N ($1 \le N \le 100000$). Во второй строке N чисел A_i, разделенные пробелами ($1 \le A_i < 10^9$). В третьей строке записано число K($1 \le K < 10^9$).

Выходные данные. Выведите единственное число – минимальное число минут, необходимое для сушки.

Пример.

input.txt	output.txt
3 2 3 9 5	3
3 2 3 6 5	2

Комментарий. Идея решения этой задачи – бинарный поиск.

Предположим, что на всю сушку у нас имеется ровно M минут времени. Для каждой вещи будем считать минимальное количество минут, которое вещь должна провести на радиаторе, чтобы высушить эту вещь за M минут. Очевидно, это число равно ceil((A_i - M) / (K-1)), где ceil(x) – минимальное целое число, не меньшее x. Тогда просуммировав эти величины по всем *i* от *1* до *N*, получим минимальное время, в течение которого мы должны использовать радиатор, чтобы успеть высушить все вещи за M минут. Если это время превышает M, то, очевидно, закончить сушку за M минут не получится. В противном случае вещи можно высушить за M минут.

То есть мы получили логическую функцию с одним параметром. Заметим, что она монотонна.

Действительно, если мы успеваем высушить вещи за M минут, то и за любое большее время мы успеваем это сделать. Аналогично, если мы не укладываемся в M минут, то и за любое меньшее время не успеем высушить вещи.

Таким образом, здесь применим бинарный поиск. Левая граница – 0, правая – максимальное из значений A_i.

Время работы алгоритма – O(N log(max(A_i)))

Программа.

```
#include <stdio.h>
#define MAXN 100000

typedef __int64 i64;

int getint() { // читаем очередное число из стандартного ввода
	int v = 0;
	char c;
	do c = getchar(); while (c < '0' || c > '9');
	do {
		v = v * 10 + c - '0';
		c = getchar();
	} while (c >= '0' && c <= '9');
	return v;
}

int a[MAXN];

int main() {
	int n, i, k, l, r;
	// читаем ввод
	// по ходу ищем max(a[i])
	n = getint();
	for (l = r = i = 0; i < n; i++)
		if ((a[i] = getint()) > r) r = a[i];
	// читаем k и обрабатываем тривиальный случай k == 1
	if ((k = getint()) == 1) {
		printf("%d\n", r);
		return 0;
	}
	// бинарный поиск (l – левая граница, r – правая граница)
	i = k - 1;
	r++;
	while (r > l + 1) {
		int m = (l + r) >> 1, j;
		i64 c; // время загрузки радиатора, которое нам потребуется для того, чтобы
			// закончить сушку за m единиц времени
		for (c = j = 0; j < n; j++)
			if (a[j] > m)
				c += (a[j] - m) / i + ((a[j] - m) % i != 0);
		if (c <= (i64)m) r = m; else l = m;
	}
	printf("%d\n", r);
	return 0;
}
```

Задача 64. Railroad sorting (Железнодорожная сортировка) [Andrew Stankevich contest 5]

Входной файл: input.txt
Выходной файл: output.txt
Ограничение по времени: 2 секунды
Ограничение по памяти: 64 мегабайт

Consider the railroad station that has n dead-ends designed in a way shown on the picture. Dead-ends are numbered from right to left, starting from 1.

Let $2n$ railroad cars get from the right. Each car is marked with some integer number ranging from 1 to $2n$, different cars are marked with different numbers.

You can move the cars through the dead-ends using the following two operations. If the car x is the first car on the path to the right of the dead-end i, you may move this car to this dead-end. If the car y is the topmost car in the dead-end j you can move it to the path on the left of the dead-end. Note, that cars cannot be moved to the dead-end from the path to its left and cannot be moved to the path on the right of the dead-end they are in.

Your task is to rearrange the cars so that the numbers on the cars listed from left to right were in the ascending order and all the cars are to the left of all the dead-ends.

One can prove that the required rearranging is always possible.

Входные данные. The first line of the input file contains n — the number of dead-ends (1 _ n _ 13). The second line contains 2n integer numbers — the numbers on the cars, listed from left to right.

Выходные данные. Output the sequence of operations. Each operation is identified with the number of the car moved in this operation. The type of the operation and the dead-end used are clearly determined uniquely.

Пример.

input.txt	output.txt
2 3 2 1 4	3 3 2 2 1 1 4 4 3 2 1 1 2 3 4 4

The sequence of the operations in the example is the following: first we move all cars through dead-end 1 without changing their order, after that we put cars 3, 2 and 1 to the dead-end 2 and take them out of it, changing their order to the reverse. Finally we move the car 4 through the dead-end 2.

Комментарий. Идея решения этой задачи – сортировка слияниями.

На первом тупике сортируем вагоны по два, на втором – по четыре, и т.д. Следует также помнить, что для слияния двух массивов нам необходимо, чтобы массив, помещаемый в стек, вводился туда в убывающем порядке, поэтому необходимо также отслеживать, чтобы первая из двух сливаемых половинок была отсортирована в убывающем порядке.

Время работы алгоритма – O(N log N).

Программа.

```
#include <stdio.h>
#define MAXN 13
#define MAXM (1 << MAXN)

void putint(int k, int howmany) {
	while (howmany--)
		printf("%d ", k);
}

int sign(int x) {
	if (x < 0) return -1;
	return x > 0;
}

int a[MAXM], aa[MAXM]; // a - исходный массив, aa - вспомогательный

void sort(int l, int r, int s) { // l, r - границы сортируемого участка, s - либо (-1 или 1)
	if (l < r) {		// -1 - сортировать в возрастающем порядке, 1 - наоборот
		int m = (l + r) >> 1, i, j;
		sort(l, m, -s); // первая половинка отсортирована в обратном порядке
		sort(m + 1, r, s);
		for (i = l; i <= m; i++) // загоняем первую половину в стек
			putint(a[i], 1);
		i = m; j = m + 1; m = l; // сливаем половинки
		while (l <= i && j <= r) {
			if (sign(a[i] - a[j]) == s) {// a[i]
				putint(aa[m++] = a[i--], 1);
			} else { // a[j]
				putint(aa[m++] = a[j++], 2);
			}
		}
		while (i >= l)
			putint(aa[m++] = a[i--], 1);
		while (j <= r)
			putint(aa[m++] = a[j++], 2);
		for (i = l; i <= r; i++)
			a[i] = aa[i];
	}
}
```

```
int main() {
	int n, i;
	scanf(“%d”, &n);
	n = 1 << n;
	for (i = 0; i < n; i++)
		scanf(“%d”, a + i);
	sort(0, n - 1, -1); // сортируем вагоны
	return 0;
}
```

ГЛАВА 9. СТРОКОВЫЕ ЗАДАЧИ

Задача 65. Ахо+Корасик [ЛКШ'2004]

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 4 секунды
Ограничение по памяти 32 мегабайта

Найдите количество вхождений каждой строки в большом тексте.

Входные данные.

Первая строка текстового файла input.txt содержит одно целое число n (1 <= n <= 10000), которое равно количеству строк. В следующих n строках записаны слова, количество вхождений которых требуется найти. В последней строке записан текст.

Длина текста и общая длина всех строк не превышает 2×10^6.

Выходные данные.

В первой строке укажите общее количество вхождений всех строк в данный текст. Далее укажите количество вхождений каждой строки в текст.

Количество вхождений всех строк в текст не превосходит 10^7.

Пример.

input.txt	output.txt
3 ababaa aaba ba ababaaba	5 1 1 3

Комментарий. Задача на реализацию алгоритма Ахо-Корасика. Ссылки могут быть найдены в *Гасфилд Д. «Строки, деревья и последовательности в алгоритмах»*.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 2000000
#define MAXA 26
#define R 0

typedef struct {
	int next[MAXA];		// next - указатели на дочерние узлы, проиндексированные алфавитом
	int p, sl, ol, id;	// p - родительский узел, sl - суффиксная ссылка, ol - исходящая
	char c;			// ссылка, id - строка, которая заканчивается в этом узле
} trie;				// c - символ на дуге, входящей в узел

trie x[MAXN];		// x - список узлов
int q[MAXN];		// q - очередь

int main() {
	int n, i, h, t = 0;
	char c;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	scanf("%d\n", &n);
	x[R].id = x[R].sl = x[R].ol = -1;
	for (i = 0; i < n; i++) {
		int w = R;	// вводим текущую строчку в trie
		c = getchar();
		while (c >= 'a' && c <= 'z') {
			c -= 'a';
			if (!x[w].next[c]) {
				x[++t].p = w;
				x[t].c = c;
				x[t].ol = x[t].id = -1;
				x[w].next[c] = t;
			}
			w = x[w].next[c];
```

```
                c = getchar();
        }
        x[w].id = i;
    }
    q[h = t = 0] = R;
    while (h <= t) {    // последовательно обходим trie
        int v = q[h++];
        char c =  x[v].c;
        if (v != R) {       // определяем суффиксную ссылку для узла
            int w = x[v].p;
            if (w != R) {
                w = x[w].sl;
                while (w != R && !x[w].next[c])
                    w = x[w].sl;
                if (x[w].next[c]) {
                    w = x[v].sl = x[w].next[c];
                    x[v].ol = (x[w].id >= 0)?w:x[w].ol;
                }
            }
        }
        for (i = 0; i < MAXA; i++) // добавляем дочерние узлы в очередь
            if (x[v].next[i])
                q[++t] = x[v].next[i];
    }
    memset(q, 0, n * sizeof(int));
    c = getchar() - 'a'; i = R; t = 0;
    // читаем строку запрос
    do {
        while (x[i].next[c]) {
            int j;
            i = x[i].next[c];
            if (x[i].id >= 0) { // нашли очередное вхождение
                t++;
                q[x[i].id]++;
            }
            j = x[i].ol;
            while (j > 0) {
                t++;
                q[x[j].id]++;
                j = x[j].ol;
            }
            if ((c = getchar()) != EOF)
                c -= 'a';
            else
                break;
        }
        if ((i = x[i].sl) == -1) {
            i = R;
            if ((c = getchar()) != EOF)
                c -= 'a';
            else
                break;
        }
    } while (!feof(stdin));
    // выводим статистику
    printf("%d\n", t);
    for (i = 0; i < n; i++)
        printf("%d ", q[i]);
    return 0;
}
```

Задача 66. Строчки [ЛКШ'2004]

Входной файл	**input.txt**
Выходной файл	**output.txt**
Ограничение по времени	**1 секунда**
Ограничение по памяти	**32 мегабайта**

Мальчик Кирилл написал однажды на листе бумаги строчку, состоящую из больших и маленьких латинских букв, а после этого ушел играть в футбол. Когда он вернулся, то обнаружил, что его друг Дима написал под его строкой еще одну строчку такой же длины. Дима утверждает, что свою строчку он получил циклическим сдвигом строки Кирилла направо на несколько шагов(циклический сдвиг строки abcde на 2 позиции направо даст

строку deabc). Однако Дима известен тем, что может случайно ошибиться в большом количестве вычислений, поэтому Кирилл в растерянности – верить ли Диме? Помогите ему!

По данным строкам выведите минимальный возможный размер сдвига или –1, если Дима ошибся.

Входные данные.

Первые две строки входного файла содержат строки Кирилла и Димы соответственно. Длины строк одинаковы, не превышают 100000 и не равны 0.

Выходные данные.

В выходной файл выведите единственное число – ответ на поставленную задачу.

Пример.

input.txt	output.txt
abcde deabc	2

Комментарий. Дописываем первую строку к себе сзади. Ищем в получившейся строке вторую строку.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 100000

char s1[(MAXN << 1) + 1], s2[MAXN + 1];
int p[MAXN];

int main() {
	int n, i, k, m;
	freopen("input.txt", "r", stdin);
	freopen("output.txt", "w", stdout);
	// читаем входные данные
	gets(s1); n = strlen(s1);
	gets(s2);
	memcpy(s1 + n, s1, n); // приписываем s1 к себе
	if (n == 1) {
		printf("%d\n", (s1[0] != s2[0])?-1:0);
		return 0;
	}
	// строим префикс - функцию для s2
	for (k = 0, i = 1; i < n; i++) {
		while (k > 0 && s2[k] != s2[i])
			k = p[k - 1];
		if (s2[k] == s2[i]) k++;
		p[i] = k;
	}
	// ищем вхождения s2 в s1
	for (m = n << 1, k = i = 0; i < (n << 1); i++) {
		while (k > 0 && s2[k] != s1[i])
			k = p[k - 1];
		if (s2[k] == s1[i]) k++;
		if (k == n) {
			int z = (n << 1) - i - 1;
			//if (z > n - z) z = n - z;
			if (z < m) m = z;
		}
	}
	printf("%d\n", (m > n)?-1:m);
	return 0;
}
```

Задача 67. Ненокку [ЛКШ'2004]

Входной файл nenokku.in
Выходной файл nenokku.out
Ограничение по времени 10 секунд
Ограничение по памяти 128 мегабайт

Очень известный автор не менее известной книги решил написать продолжение своего произведения. Он писал все свои книги на компьютере, подключенном к интернету. Из-за такой не осторожности мальчику Ненокку

удалось получить доступ к еще ненаписанной книге. Каждый вечер мальчик залазил на компьютер писателя и записывал на свой компьютер новые записи. Ненокку, записав на свой компьютер очередную главу, заинтересовался, а использовал ли хоть раз писатель слово «книга». Но он не любит читать книги (он лучше полазает в интернете), и поэтому он просит вас узнать есть ли то или иное слово в тексте произведения. Но естественно его интересует не только одно слово, а достаточно много.

Входные данные.

В каждой строчке входного файла записано одна из двух записей.

1 - ? <слово> (<слово> - это набор не более 50 латинских символов)

2 - A <текст> (<текст> - это набор не более 10^5 латинских символов)

1 означает просьбу проверить существование подстроки <слово> в произведении.

2 означает добавление в произведение <текст>.

Писатель только начал работать над произведением, поэтому он не мог написать более 10^5 символов. А входной файл содержит не более 20 мегабайт информации.

Выходные данные.

Выведите на каждую строчку типа 1 «YES», если существует подстрока <слово>, и «NO» в противном случае. Не следует различать регистр букв.

Пример.

nenokku.in	nenokku.out
? love ? is A Loveis ? love ? WHO A Whoareyou ? is	NO NO YES NO YES

Комментарий. Задача на использование алгоритма Укконена построения суффиксных деревьев, которые обсуждаются в главе 10.

Программа.

```
#include <stdio.h>
#include <string.h>
#define MAXN 200005
#define MAXA 27
#define Q 1
#define A 2
#define olen(x, d) (t[x].next[s[d]] ? ilen(t[x].next[s[d]]) : 0)

int readcmd() {
	char c;
	do c = getchar(); while (!feof(stdin) && c != '?' && c != 'A');
	switch (c) {
		case '?': return Q;
		case 'A': return A;
		default : return 0;
	}
}

int reads(char* s) {
	int l = 0;
	char c;
	do c = getchar(); while ((c < 'a' || c > 'z') && (c < 'A' || c > 'Z'));
	do {
		s[l++] = c - ((c > 'Z')?'a':'A');
		c = getchar();
	} while (!feof(stdin) && c != '\n');
	return l;
}

typedef struct {
	int sp, fp, p, sl, d;
	int next[MAXA];
	char l;
} node;
```

```
node t[(MAXN << 1) + MAXN];
char s[MAXN << 1], p[MAXN << 1];
int curr, last, e, w;

int ilen(int x) {
	int r = (t[x].l) ? e : t[x].fp;
	return r - t[x].sp + 1;
}

int makeleaf(int x, int d) {
	register int z;
	if (t[x].next[s[d]]) return 0;
	t[x].next[s[d]] = z = ++last;
	t[z].sp = t[z].fp = d;
	t[z].p = x; t[z].d = t[x].d + 1;
	t[z].l = 1;
	return 1;
}

int split(int x, int d, int w, char f) {
	register int z;
	int l, m, c = t[x].next[s[d]];
	l = t[c].sp;
	if (s[m = l + w] == f) return 0;

	z = ++last;
	t[x].next[s[d]] = z;
	t[c].p = z; t[c].sp = m;
	t[z].sp = l; t[z].fp = m - 1; t[z].d = t[x].d + w;
	t[z].next[s[m]] = c;
	return 1;
}

int extend(int j, int i) {
	int k, g, l;
	k = j + t[curr].d;
	g = i - k;
	l = olen(curr, k);
	while (l && g && l <= g) {
		curr = t[curr].next[s[k]];
		g -= l;
		k += l;
		l = olen(curr, k);
	}
	if (g)
		if (split(curr, k, g, s[i])) {
			if (w) t[w].sl = last;
			curr = last;
			w = (j + 1 < i) ? curr : 0;
		} else
			return 0;
	else
		if (w) {
			t[w].sl = curr;
			w = 0;
		}
	return makeleaf(curr, i);
}

int search(int n, int md) { // поиск строки p в s (n – длина p,
	int i = 0, z = 0;   // md – максимальная глубина поиска)
	while (i < n && t[z].next[p[i]]) {
		int j, r = t[z].next[p[i]];
		j = t[r].sp;
		z = r;
		r = t[r].l ? md : t[r].fp;
		while (j <= r && i < n && p[i] == s[j]) {
			i++; j++;
		}
		if (j <= r && i < n) break;
	}
	return i == n;
```

```
}

int main() {
	int f, i, j, n;
	freopen("nenokku.in", "r", stdin);
	freopen("nenokku.out", "w", stdout);
	i = j = n = 0;
	// читаем запросы и параллельно их обрараты ваем
	while (f = readcmd()) {
		switch (f) {
			case A:	// добавление строки в суффиксное дерево
				n += reads(s + i);
				for (; i < n; e = ++i) {
					if (!extend(j, i)) continue;
					j++;
					while (j <= i) {
						while (curr && !t[curr].sl)
							curr = t[curr].p;
						curr = t[curr].sl;
						if (extend(j, i)) j++;
						else break;
					}
				}
				break;
			case Q:	//  запрос
				f = reads(p);
				puts(search(f, n - 1) ? "YES" : "NO");
				break;
		}
	}
	return 0;
}
```

Задача 68. Палиндром.

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 3 секунды
Ограничение по памяти 16 мегабайт

Палиндром — это строка, которая читается одинаково как справа налево, так и слева направо.

Во входном файле записан набор больших латинских букв (не обязательно различных). Разрешается переставлять буквы, а также удалять некоторые буквы. Требуется написать программу, которая из данных букв по указанным правилам составит палиндром наибольшей длины, а если таких палиндромов несколько, то первый в алфавитном порядке.

Входные данные.

В первой строке входного файла записано число N ($1 \le N \le 100000$). Во второй строке записана последовательность из N больших латинских букв (буквы записаны без пробелов).

Выходные данные.

В единственной строке выходного файла выдайте искомый палиндром.

Пример:

input.txt	output.txt
3 AAB	ABA
6 QAZQAZ	AQZZQA
6 ABCDEF	A

Комментарий.

Для начала маленький и практически очевидный факт: каждая буква в палиндроме встречается четное количество раз, за исключением, может быть, одной – средней.

Теперь подсчитаем, сколько раз встречается каждая буква в исходной строке. Среди всех букв, которые встречаются нечетное количество раз, выберем лексикографически минимальную (если, конечно же, такая существует) – обозначим ее через f. Построим первую половину строки следующим образом: будем

последовательно двигаться по буквам в алфавитном порядке и приписывать к текущему результату эту букву ровно половину от количества раз, которое она встречается. Выведем ее в ответ. Если символ f существует, выведем его в ответ также. Выведем в ответ построенную строку в обратном порядке.

Программа.

```
#include <stdio.h>
#define MAXN 100000
#define MAXA 26

int c[MAXA];
char s[MAXN + 1];

int main() {
	int n, i;
	char f;
	scanf("%d\n", &n);
	gets(s);
	for (i = 0; i < n; i++) // считаем, статистику
		c[s[i] - 'A']++;
	for (f = i = 0; i < MAXA; i++) { // ищем f, уменьшаем c[i] вдвое
		if (c[i] & 1) {
			if (!f)
				f = 'A' + i;
		}
		c[i] >>= 1;
	}
	for (n = i = 0; i < MAXA; i++) { // строим результат
		int j;
		for (j = c[i]; j > 0; j--)
			putchar(s[n++] = i + 'A'); // параллельно выводим его
	}
	if (f) putchar(f); // выводим f
	while (n--) putchar(s[n]); // выводим s в обратном порядке
	return 0;
}
```

ГЛАВА 10. СУФФИКСНЫЕ ДЕРЕВЬЯ

В данной главе представлены некоторые приложения суффиксных деревьев в часто встечающихся задачах биоинформатики (задачах на строках). Сравниваются основные специфические подходы к решению задач и методы, основанные на суффиксных деревьях.

Во-первых, рассматриваются приложения в области классической задачи точного совпадения. В работе показано, как суффиксные деревья позвояют решать основные задачи точного совпадения наравне с существующими специфическими методами для решения той или иной задачи. В частности, представлен простой алгоритм решения задачи линеаризации циклической строки.

Во-вторых, рассматриваются приложения к неточному совпадению. Здесь зачастую существующие методы не могут дать такой высокой производительности, как суффиксные деревья.

Строки применяются во многих областях деятельности человека. Сходство строк, например, широко используется в таких различных областях, как анализ биологических последовательностей, поиск информации, распознование шаблонов, обработка изображений и сигналов, оптически-символьное распознввание и т.д. Сравнение биологических последовательностей очень важно для молекулярных биологов и других ученых. Когда биологические последовательности двух организмов схожи, их владельцы могут происходить от общего предка в эволюционном дереве или появились с целью выполнения подобных функций. Ученые могут прогнозировать некоторые факты относительно генотипа человека посредством изучения последовательностей человеческих ДНК или основываясь на известных и схожих последовательностя ДНК других видов, таких как мышь, например. К сожалению, последнее неприемлимо, когда речь идет о сравнении последовательностей в большом, геномном масштабе.

С другой стороны, поиск образца в тексте – также хорошо известная задача. В настольных приложениях, таких как текстовые процессоры, можно искать небольшие образцы в изменяющемся тексте. В биоинформатике ищут сложные шаблоны в тестах существенно болшего размера. В поисковых движках, в интернете часто возникает необходимость поиска очень простых строк в огромных объемах информации. Существует также множество других, более сложных задач поиска образца в тексте.

Суффиксное дерево – это структура данных, которая выявляет очень глубоко внутреннее строение строки. Поэтому суффиксные деревья можно использовать при для решения задач точного совпадения за линейное время (достигая той же границы для наихудшего случая, как в классических алгоритмах Кнута-Морриса-Пратта и Бойера-Мура). Но их подлинное превосходство становится очевидным при решении за линейное время многих строковых задач, более сложных, чем точные совпадения. Более того, суффиксные деревья наводят мост между задачами точного совпадения и неточного совпадения.

Теперь можно дать более формальное определение нашей задачи. Цель этой главы состоит в том, чтобы продемонстрировать технику и мощь использования суффиксных деревьев и сравнить предложенную технику с существующими специфическими алгоритмами, а также предложить ряд усовершенствований к и тем, и другим алгоритмам.

Сравнение строк – это чрезвычайно важная задача идентификации и представления биологически важных, но пока еще скрытых или широко разбросанных общих характеристик набора строк. Эти общие участки могут заключать в себе историю эволюции, критически скрытые узоры ДНК или белков, общие двумерные или трехмерные структуры или решения задач об обших биологических функциях строк. Такие общности также ипользуются для характетизации семейств или суперсемейств белков. Эти характеризации позднее используются при поиске в базах данных для идентификации других потенциальных членов семейства.

Первый алгоритм конструирования суффиксных деревьев за линейное время был предложен Вайнером [20] в 1973 году. Другой алгоритм, более экономный, был предложен Мак-Крейгом [13] несколько лет спустя. Чуть больше десяти лет назад Укконен [19] разработал принципиально иной алгоритм построения суффиксных деревьев за линейное время, который обладает всеми преимуществами своих предшественников.

При хорошей реализации они очень удобны и разнообразны в применениях. До настоящего момента не известна другая структура данных (если не говорить о структурах, эквивалентных суффиксным деревьям), которая бы допускала эффиктивное решение столь широкого класса трудных строковых задач [7].

ОПРЕДЕЛЕНИЯ И ДОГОВОРЕННОСТИ

Всюду в данной работе будем считать, что алфавит ∑ фиксирован и состоит из строчных букв латинского алфавита (‘a’..’z’) и символа ‘$’. Такой выбор не случаен: ведь, например, ДНК состоит всего лишь из 4 нуклеотидов, существует только 20 аминокислот. Предложенный алфавит кроет и ту, и другую задачи.

Определение. *Строка* – это упорядоченный список символов из ∑, выпичанный последовательно слева направо.

Строку символов (возможно, пустую) из ∑ \ {‘$’} будем обозначать S. Некоторый символ из ∑ обозчачим x, y или z.

Определение. *Подстрока* строки S, S[i..j] – это фрагмент строки S, начинающийся в позиции i и заканчивающийся в позиции j. Подстрока пуста, если i > j.

Определение. *Префикс* w строки t – это такая строка, что wv = t для некоторой (возможно, пустой) строки v. Префикс называется собственным, если |v|>0.

Определение. *Суффикс* w строки t – это такая строка, что vw = t для некоторой (возможно, пустой) строки v. Суффикс называется собственным, если |v|>0.

Определение. Если два символа оказались одинаковыми при сравнении, то говорят о *совпадении*, в противном случае – о *несовпадении*.

Определение. *Суффиксное дерево* T для n-символьной строки S – это ориентированное дерево с корнем, имеющее ровно n листьев, занумерованных от 1 до n. Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей, а каждая дуга помечена непустой подстрокой строки S (дуговой меткой). Никакие две дуги, выходящие из одной и той же вершины, не могут иметь меток, начинающихся с одного и того же символа. Главная особенность суффиксного дерева заключается в том, что для каждлго листа i конкатенация дуговых меток на пути от корня к листу i в точности составляет(произносит) суффикс строки S, который начинается в позиции i. То есть этот путь произносит S[i..n].

Пример суффиксного дерева для строки S = “xabxac” показан на рисунке 6.1:

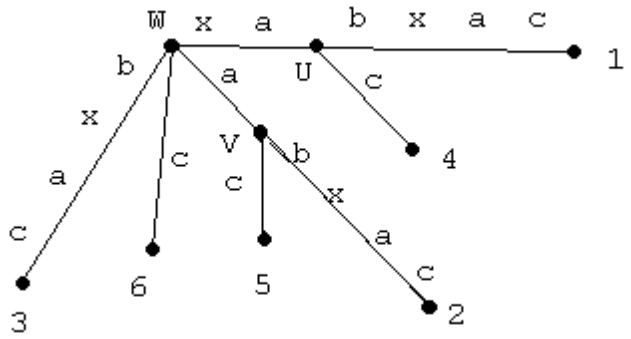


рис. 2.1 Пример суффиксного дерева

Определение. *Метка пути* от корня до некоторой вершины – это конкатенация подстрок, помечающих дуги пути в порядке их прохождения. Путевая метка вершины – это метка пути отпути от корня T до этой вершины.

Определение. Для любой вершины v суффиксного дерева T *строковой глубиной* называется число символов в путевой метке v.

Определение. Путь, который кончается в середине дуги (u, v), делит метку (u, v) в своей точке назначения. Определим *метку этого пути* как путевую метку u с добавлением символов дуги (u, v) до делящей дугу точки назначения.

Например, для суффиксного дерева, показанного на рисунке 2.1, путевой меткой листовой вершины 1 будет являться строка “xabxac”, которая в свою очередь складывается из меток двух дуг - “xa” и “bxac”.

Надо сказать, что если некий суффикс строки S является префиксом какого-нибудь другого суффикса этой строки, то этот вершина, соответствующая этому суффиксу, будет отсутствовать в дереве. Поэтому будем считать, что последний символ строки S обльше нигде в строке не появляется. На практике этого можно добиться добавлением символа ‘$’.

Теперь об алгоритме построения суффиксных деревьев:

Суффиксное дерево для строки S длины n может быть построено за время O(n) с помрщью алгоритма Укконена [Гасфилд Д., “Строки, деревья и последовательности в алгоритмах”] (далее – просто Гасфилд).

Приведем реализацию алгоритма Укконена на языке C:

```
// структура для хранения узла суффиксного дерева
typedef struct {
int sp, fp, p, sl, d, cc, f, w;
int next[MAXA];
char l;
} node;
/*
```

```
sp, fp – начальная и конечная позиция подстроки, помечающей ребро, входящее в эту вершину;
p – родительский узел;
sl – суффиксная ссылка;
d – строковая глубина вершины;
cc – количество листьев в поддереве, корнем которого является эта вершина;
w – номер строки, при добавлении которой в суффиксное дерево была создана эта вершина;
f – флаги, вершины
*/

int e, w, curr, last, sc, tot;
node t[(MAXN << 1) + MAXN];
char s[(MAXN << 2) + 1], u[MAXN << 2];
/*
e – текущий символ строки;
w – последняя созданная внутренняя вершина;
curr – текущая вершина;
last – последняя распределенная вершина из списка вершин t;
sc – количество строк в суффисном дереве;
tot – суммарная длина строк в суффиксном дереве;
*/
#define olen(x, d) (t[x].next[s[d]] ? ilen(t[x].next[s[d]]) : 0)
// длина подстроки, помечающей дугу, исходящую из вершины x по символу s[d]

int ilen(int x) { // длина подстроки, помечающей дугу, входящую в x
	register int r = (t[x].l && t[x].w >= sc) ? e : t[x].fp;
	return r - t[x].sp + 1;
}

void makeleaf(int x, int d) { // создать потомка вершины x по символу s[d]
	register int z = ++last;
	t[x].next[s[d]] = z;
	t[z].sp = t[z].fp = d;
	t[z].p = x; t[z].d = t[x].d + 1;
	t[z].l = 1;
	t[z].f |= (1 << (t[z].w = sc));
}

int split(int x, int i, int q, char f) {
/*
попробовать разбить дугу, выходящую из x, на q-м символе при условии, что эта дуга не
продолжается символом f
*/
	register int z;
	int c, l, r, m;
	c = t[x].next[s[i]];
	l = t[c].sp; r = t[c].fp;
	if (s[m = l + q] == f) return 0;
	z = ++last;
	t[x].next[s[i]] = z;
	t[c].p = z; t[c].sp = m;
	t[z].p = x; t[z].sp = l; t[z].fp = m - 1; t[z].d = t[x].d + q;
	t[z].next[s[m]] = c;
	return 1;
}

int extend(int j, int i) { // j-е продолжение i-й фазы алгоритма
	int k, l, g;
// скачок по счетчику (“skip counter”)
	k = j + t[curr].d; // текущий символ в строке
	g = i - k; // количество символов, которые остается пройти
	l = olen(curr, k);
	while (l && g && l <= g) {
		curr = t[curr].next[s[k]];
		k += l;
		g -= l;
		l = olen(curr, k);
	}
	if (g) // если есть, куда идти
		if (split(curr, k, g, s[i])) { // пытаемся разбить эту дугу
			if (w) t[w].sl = last;  // при этом появилась внутренняя вершина
			curr = last;            // создаем суффиксную ссылку
			w = (j + 1 < i) ? curr : 0;
		} else
```

```
                if (s[i] + 1 == MAXA) {
                        t[t[curr].next[s[k]]].f |= (1 << sc);
                        return 1;
                } else
                        return 0;
                //}
        else
                if (w) {
                        t[w].sl = curr;
                        w = 0;
                }
        if (!t[curr].next[s[i]])
                makeleaf(curr, i, j); // создаем новую листовую вершину
        else
        if (s[i] + 1 == MAXA) t[t[curr].next[s[i]]].f |= (1 << sc);
        else
                return 0;
        return 1;
}

void build(int n) { // построить суффисное дерево для (глобальной) строки s длины n
        int i, j;
        e = w = curr = last = sc = 0; tot = n + 1;
        for (j = i = 0; i <= n; e = ++i) { // i-я фаза
                if (!extend(j, i)) continue; // j-е продолжение
                j++;
                while (j <= i) {
                        while (curr && !t[curr].sl) // движемся вверх до первой
                                curr = t[curr].p;    // вершины с суффисной ссылкой
                        curr = t[curr].sl;          // проходим по ней
                        if (!extend(j, i)) break; // j-е продолжение
                        j++;
                }
        }
        for (i = last; i >= 0; i--) // обновляем
                if (t[i].l) t[i].fp = n,  t[i].d = t[t[i].p].d + ilen(i);
}

void add(char *p, int n) { // добавить p длины n в суффискное дерево
        int i = tot, j, z = last;
        w = curr = 0; tot += n + 1;
        memcpy(s + i, p, n + 1); // копируем p в глобальный буфер
        for (n += (e = j = i); i <= n; e = ++i) { // повторяем все шаги из build
                if (!extend(j, i)) continue;
                j++;
                while (j <= i) {
                        while (curr && !t[curr].sl)
                                curr = t[curr].p;
                        curr = t[curr].sl;
                        if (!extend(j, i)) break;
                        j++;
                }
        }
        for (i = last; i > z; i--)
                if (t[i].l) t[i].fp = n, t[i].d = t[t[i].p].d + n - t[i].sp;
        sc++;
}

int ss[MAXN << 1];
/*
ss – стек. Стек реализован вручную, поскольку по умолчанию размер стека прогрммы составляет
1MB. При большом числе вложенных рекурсивных вызовов может произойти переполнение стека.

Данная проблема в принципе может быть решена с помощью деректив компилятору на увеличение
размера стека, например:
{$minstacksize $4000000} // Delphi
{$minstacksize $4000000}

#pragma comment(linker, "/STACK:0x4000000") // Microsoft Visual C/C++
*/

void traverse(int v) { // обойти дерево, посчитать статистику для каждой вершины
/*
```

```
статистика:
сколко листьев в поддереве, корнем которого является данная вершина
какие строки содержат путевую метку вершины в качестве подстроки
*/
	int t0 = 0;
	memset(u, 0, sizeof(u));
	ss[0] = v;
	while (t0 >= 0) {
		int x = ss[t0];
		if (!u[x]) {
			int i;
			for (i = 0; i < MAXA; i++)
				if (t[x].next[i]) ss[++t0] = t[x].next[i];
			u[x] = 1;
		} else {
			int i;
			if (t[x].l) t[x].cc = 1; else
			for (i = 0; i < MAXA; i++)
				if (t[x].next[i]) {
					t[x].cc += t[t[x].next[i]].cc;
					t[x].f |= t[t[x].next[i]].f;
				}
			t0--;
		}
	}
}

int reconstruct(int x, char *p) { // восстановить (развернутую) путевую метку вершины
	int n = 0;
	while (x > 0) {
		int k = t[x].sp, r = t[x].fp;
		while (r >= k) {
			if (s[r] + 1 != MAXA) p[n++] = 'a' + s[r];
			r--;
		}
		x = t[x].p;
	}
	return n;
}
```

Задача 69. Точное совпадение строк

Формулировка. Задача точного совпадения строк состоит в том, чтобы по заданной строке P, которая называется образцом, и более длинной строке T, которая называется текстом, найти все вхождения, если, конечно, таковые имеются, образца P в текст T. Например, пусть P=”xyz”, T= “rstxyzuvxyz”, тогда P входит в T дважды, с позиций 4 и 9.

Подходы, не основынные на суффиксных деревьях.

Здесь мы проилюстрируем основные подходы, применяемые для решения задани точного совпадения строк. Их можно разделить на несколько групп:

Основной препроцессинг. Алгоритмы, следующие этому подходу, “умно” пропускают некоторые сравнения символов строк, сначала затратив некоторое время на изучение внутренней структуры текста или образца. Эта часть алгоритма называется фазой препроцессинга. После препроцессинга следует этап поиска, в котором та информация, которая была собрана на фазе препроцессинга, используется для организации “умного” поиска вхождений P в T.

Типичным представителем служит Z-алгоритм[6].

Классические методы, основанные на сравнениях. Эти алгоритмы выравнивают образец P и текст T и смотрят, совпадают ли стоящие напротив друг друга символы P и T. Далее образец P сдвигается вправо относительно T. Если |P| = n, |T| = m, то этот сдвиг реализуется, следуя нескольким правилам, что ведет к тому, чтосравниваются не более m+n символов, - т. е. Алгоритм работает за линейное время в худшем случае.

Вот некоторые примеры алгоритмов этой группы: алгоритм Бойера-Мура[8, 9, 15, 16, 18], алгоритм Кнута-Морриса-Пратта[11].

Получисленные методы сравнения. Эти методы основываются скорее на битовых или арифметических операциях, чем на сравнении символов. Поэтому они имеют совершенно другой механизм. Но иногда сравнение символов кроется на промежуточных уровнях работы этих методов.

К таким методам относятся: алгоритм Shift-And[1], Быстрое Преобразование Фурье [4,5], “дактилоскопические” методы Карпа-Рабина[10].

Подход, использующий суффиксные деревья.

Алгоритм. Если у нас имеется суффиксное дерево, построенное для текста T, то поиск всех вхождений образца P в T производится банальным спуском от корня дерева вдоль единственного пути, определяемого P. Если вдруг окажется, что пути нет(т.е. из некоторой верщины мы не можем пойти дальше), то P не входит как подстрока в T. В противном случае для того, чтобы найти все вхождения P в T, необходимо обойти все листовые вершин поддерева, определяемого вершиной, где закончился путь.
Очевидно, в этом случае на подготовку и поиск мы затрачиваем время, пропорциональное сумме длин текста и образца.

В случае, когда рассматривается подход, основанный на суффиксных деревьях, можно выделить несколько ситуаций:

текст и образец становятся известны алгоритму в одно и то же время. Этот случай полностью покрывается классическими алгоритмами, и никакого выигрыша по сравнению с ними суффиксные деревья не дают;
текст T известен заранее и на некоторое время остается фиксированным. После обработки текста вводится длинная последовательность образцов, кажлый из которых должен быть обработан как можно скорее. Пусть |P|=n, а количество вхождений P в T равно k. Тогда с использованием суффиксного дерева для T все вхождения можно найти за время O(n + k) абсолютно независимо от размера T. Это уже существенное преимущество суффиксных деревьев перед классическими методами, поскольку на каждый запрос они тратят время O(n + m).
образец P задан заранее, и его можно обработать заранее. Здесь более естественно применение классических методов Кнута-Морриса-Пратта и Бойера-Мура. Однако здесь также возможно применение суффиксных деревьев.

Задача. В первой строке входного файла строка T, во второй – строка P. Длины обеих строк не превосходят 100000. Вывести все вхождения строки P в T как подстроки или сообщить, что таковых нет.

```
#include "..\\global.h"
#include "..\\func.h"

char p[MAXN];
int q[(MAXN << 1) + MAXN];

int main() {
	int n, m, i, j;
// читаем строки
	gets(s); n = strlen(s);
	gets(p); m = strlen(p);
	for (i = 0; i < n; i++)
		s[i] -= 'a';
	for (i = 0; i < m; i++)
```

```
            p[i] -= 'a';
        s[n] = MAXA - 1;
        build(n); // строим суффиксное дерево для T(которая в это время находится в s)
        j = i = 0;
        while (i < m) {
        // спускаемся по пути, помечаемом P
            int k, r;
            if (!t[j].next[p[i]]) break;
            j = t[j].next[p[i]]; k = t[j].sp; r = t[j].fp;
            while (k <= r && i < m && p[i] == s[k]) i++, k++;
        }
        if (i >= m) { // нашли эту строку
            int h0, t0;
            // обходим все листовые вершины и выводим их по очереди
            q[h0 = 0] = j; t0 = 1;
            while (h0 < t0) {
                int x = q[h0++];
                if (t[x].l) {
                    printf("matching at %d\n", t[x].sid);
                    continue;
                }
                for (i = 0; i < MAXA; i++)
                    if (t[x].next[i]) q[t0++] = t[x].next[i];
            }
        } else
            puts("no matching"); // P не входит в T
        return 0;
}
```

Задача 70. Поиск наибольшей общей подстроки.

Формулировка. Пусть заданы две строки S_1 и S_2. *Задача о наибольшей общей подстроке* состоит в том, чтобы найти наибольшую подстроку, общую для заданных строк S_1 и S_2.

Например, если S_1=”gnusnotunix”, а S_2=”tuning”, то наибольшей общей подстрокой строк S_1 и S_2 будет “tun”. Подходы, не основанные на суффиксных деревьях.

Очевидно, для решения этой задачи можно применить динамическое программирование. Пусть $|S_1|=n$, $|S_2|=m$. Тогда для всех пар (i, j), где $1 \le i \le n$, $1 \le j \le m$, будем считать длину наибольшей подстроки S_1 и S_2, которая заканчивается в позиции i в S_1 и в позиции j в S_2. По окончании этой процедуры останется выбрать максимум по всем парам (i, j) – это и будет ответом к задаче.

Несложно подсчитать, что этот прямой метод требует времени O(n * m), что при больших значениях n и m неприемлимо.

Другой, более эффективный подход к решению этой задачи состоит в применении задачи о наибольшем общем префиксе.

Интересно отметить, что в 1970 г. Д.Кнут предположил, что линейный алгоритм для этой задачи не существует[7].

Подход, использующий суффиксные деревья.

Алгоритм.

Построим обощенное суффиксное дерево для S_1 и S_2. Каждый лист этого дерева представляет собой либо суффикс одной из этих строк, либо суффикс обеих строк одновременно;

Каждую внутреннюю вершину пометить числом 1 (или 2), если в поддереве, корнем которого является эта вершина есть лист, кодирующий суффикс S_1(или, соответственно, S_2). Путевая метка любой вершины, помеченной и 1, и 2, есть подстрока, общая для S_1 и S_2. И самая длинная такая подстрока и есть искомая.

Таким образом, алгоритм должен выбрать максимум из строковых глубин вершин, помеченных 1 и 2 одновременно. Построение суффиксного дерева может быть выполнено за время, пропорциональное сумме длин строк S_1 и S_2. Количество вершин в дереве также линейно зависит от длин строк S_1 и S_2. Поэтому обход дерева при поиске максимума занимает линейное время. Получили теорему.

Теорема. *Наибольшая общая подстрока двух строк с использованием обощенного суффиксного дерева может быть найдена за линейное время.* ↑

Задача. Строки S1 и S2 находятся в, соответственно, первой и второй строках входного файла. Длины обеих строк не превосходят 100000.

```
#include "..\\global.h"
#include "..\\func.h"

char p[MAXN + 1]; // буфер
int st[(MAXN << 1) + MAXN]; // стек

int main() {
```

```
	int n, m, i, at;
	//memset(s, 0, sizeof(s));
	// читаем строки и добавляем их в суффиксное дерево
	gets(p); n = strlen(p);
	for (i = 0; i < n; i++)
		p[i] -= 'a';
	p[n] = MAXA - 1;
	add(p, n);
	//go(0);
	gets(p); n = strlen(p);
	for (i = 0; i < n; i++)
		p[i] -= 'a';
	p[n] = MAXA - 1;
	add(p, n);
	// считаем статистику
	traverse(0);
	m = st[i = 0] = 0; at = -1;
	// ищем вершины, помеченные одновременно обеми строками
	while (i >= 0) {
		int x = st[i--];
		if (t[x].f == 3) {
			int j;
			if (t[x].d > m) m = t[at = x].d;
			for (j = 0; j < MAXA; j++)
				if (t[x].next[j]) st[++i] = t[x].next[j];
		}
	}
	if (!m) {
		puts("no common substring");
		return 0;
	}
	// восстанавливаем ответ
	n = reconstruct(at, p);
	while (n > 0) putchar(p[--n]);
	return 0;
}
```

Задача 71. Обратная роль суффиксных деревьев.

В пункте 1 (apl1) рассматривались методы решения задачи точного совпадения как классическими методами, так и с помощью суффиксных деревьев. Эффективность суффиксных деревьев ярко проявилась в случае, когда текст был заранее известен, - время O(m) на обработку текста и O(n + k) на поиск всех вхождений образца в текст незасвисимо от m. Однако в обратном случае, когда заранее дан образец, а не текст, такой подход не пройдет. Здесь эффективными оказываются алгоритмы Кнута-Морриса-Пратта и Бойера-Мура, которые предварительно обрабатывают образец, тратя время и память O(n), а затем производят поиск за O(m).

В этом случае возможно строить суффиксное дерево не для текста, а для образца, экономя таким образом память. Чтобы продемонстрировать технику, используемую в данном случае, мы обратимся к более общему результату, называемому *задачей о статистике совпадений*[2].

Определение. Пусть ms(*i*) – длина наибольшей подстроки T, начинающейся с позиции *i*, которая совпадает с какой-то подстрокой P. Эти значения называются *статистикой совпадений*.

Например, если T=”abcxabcdex”, P=”wyabcwzqabcdw”, то ms(1) = 3, ms(5) = 4.

Ясно, что вхождение P, начинающееся с позиции *i* в T, существует в том и только в том случае, если ms(*i*) = |P|.

Заметим, что статистику совпадений можно использовать для уменьшения размера суффиксного дерева в задачах точного совпадения – ведь как правило размер текста значительно превосходит размер образца. Статистика совпадений применяется также во многих других приложениях, таких как поиск быстрого приближенного совпадения, разработанный для быстрого поиска в базах данных.

Для вычисления статистики совпадений можно использовать следующий алгоритм:

Построить суффиксное дерево для P, используя алгоритм Укконена [19]. Суффиксные связи, используемые алгоритмом Укконена, не удалять;

ms(1) найти прямым поиском в построенном суффиксном дереве, двигаясь вдоль текста. Пусть при этом поиск остановился в вершине b;

Зная ms(*i*) и b, ms(*i* + 1) можно найти с помощью приемов, используемых алгоритмом Укконена: переходами по суффиксным связям и скачками по счетчику.

Теорема. *Имея суффиксное дерево для P, полную статистику совпадений можно найти за время O(m).*

Детали алгоритма и доказательство теоремы можно найти в [7].

Как уже было сказано, подсчет статистики совпадений как подзадача возникает во многих других задачах, например lcp с экономией памяти. Поэтому бывает полезно также для каждой позиции *i* в тексте T найти p(*i*).

Определение. Число p(*i*) указывает начальную позицию в P, такую что подстрока, начинающаяся в p(*i*), совпадает с подстрокой, начинающейся в позиции *i* в T на длину ms(*i*).

Нахождение всех значений p(*i*) не представляет сложности: для этого надо совершить обход суффиксного дерева для P в глубину и каждой внутренней вершине назначить номер любого суффикса, для которого его представляющая листовая вершинапринадлежит поддереву, корнем которого является данная вершина. При подсчете значений ms(*i*) достаточно брать метку вершины b[7].

Задача. Young Hackers [Andrew Stankevich Contest 15]

Time limit: 2 seconds

Memory limit: 64 megabytes

Young hackers Andrew and Beth are trying to find some secret information in a long log of the chat conversation of their friend Chris. They know that the secret part of the conversation is related to the message to his friend Dan they intercepted before. But the log is too long to search by hand.

Therefore they decided to find the parts that are most similar to the message to analyze them later.

They ask you to help them to write the corresponding program.

Let us represent the log as one string t with all spaces, line feeds, and punctuation marks deleted. We will also ignore the case of the letters. Thus, the string consists of capital letters of the English alphabet and digits. The message in turn is also converted to the same form, let us denote the corresponding string as p.

For each position i in t Andrew and Beth want to find the length of the maximal substring of p that starts in t at i-th position. Thus, for each i you have to find such j = j(i) that t[i...i+j−1] = p[k...k+j−1] for some k, and j is maximal possible.

Input

The input file contains two lines. The first line contains t, the second line contains p. The length of t does not exceed 100 000, the length of p does not exceed 5 000. Both strings are not empty.

Output

Let the length of t be n. Output n numbers — for each i output the corresponding j(i).

Example

hackers.in	hackers.out
ABBAABABC ABAAB	2 1 4 3 3 2 2 1 0

```
#include "..\\global.h"
#include "..\\func.h"
#ifdef MAXA
#undef MAXA
#endif
#define MAXA 37
#define olen2(x, d) (t[x].next[tt[d]] ? ilen(t[x].next[tt[d]]) : 0)
#define ilen2(x) (t[x].fp - t[x].sp + 1)

void putint(int k) { // вывести k в стандартный вывод
	int c, d[20];
	c = 0;
	do {
		int q = k / 10;
		d[c++] = k - q * 10;
		k = q;
	} while (k);
	while (c > 0) putchar('0' + d[--c]);
}

int prepare(char *s) {
	int n, i;
	gets(s); n = strlen(s);
	for (i = 0; i < n; i++)
		if (s[i] >= 'a' && s[i] <= 'z')
			s[i] -= 'a';
		else
		if (s[i] >= '0' && s[i] <= '9')
			s[i] -= '0' - 26;
		else
			putchar('!');
	return n;
}
```

```
int ans[MAXT];

int main() {
	int n, m, i, j, k;
	m = prepare(tt);
	n = prepare(s);
	//gets(tt); m = strlen(tt);
	//gets(s); n = strlen(s);
	//s[n++] = MAXA - 1;
	build(n);
	for (k = j = i = 0; i < m; i++) { // повторяем шаги алгоритма Укконена
		int l, z;
		// идем вверх по первой суффискной ссылки
		while (j && !t[j].sl) {
			k += ilen2(j);
			j = t[j].p;
		}
		if (!(j = t[j].sl) && k) k--;
		// спускаемся вниз
		// k - сколько символов еще осталось пройти
		// z - позиция в s, где мы сейчас находимся

		z = i + t[j].d;
		l = olen2(j, z);
		while (l && k && l <= k) {
			j = t[j].next[tt[z]];
			z += l;
			k -= l;
			l = olen2(j, z);
		}
		if (k) {
			int c = t[t[j].next[tt[z]]].fp;
			// спускаемся к ближайшей вершине
			l = t[t[j].next[tt[z]]].sp + k;
			k += i + t[j].d;
			while (k < m && l <= c && tt[k] == s[l])
				l++, k++;
			if (l > c) {
				j = t[j].next[tt[z]];
			} else {
				ans[i] = (k -= i);
				k -= t[j].d;
				continue;
			}
		} else
			k += i + t[j].d;
		while (k < m && (z = t[j].next[tt[k]])) {
			int c;
			l = t[z].sp; c = t[z].fp;
			while (k < m && l <= c && tt[k] == s[l])
				l++, k++;
			if (l > c)
				j = z;
			else
				break;
		}
		ans[i] = (k -= i);
		k -= t[j].d;
	}
	for (i = 0; i < m; i++) {
		putint(ans[i]);
		putchar(i + 1 < m ? ' ' : '\n');
	}
	return 0;
}
```

Задача 72. Линеаризация циклической строки.

Определение. Циклическая строка длины n – это такая строка, n-й символ которой предшествует первому.

Определение. При заданном упорядочении символов алфавита строка S_1 *лексически*(или *лексикографически*) меньше строки S_2, если строка S_1 должна стоять рашьше S_2 в нормальном словарном упорядочении этих строк. Т.е.

если при счете от левого конца этих двух строк *i* – первая позиция, где строки различны, то S_1 лексически меньше S_2 в том и только в том случае, если S1(i) предшествует S2(i) в упорядочении алфавита, использованного для этих строк.

Формулировка задачи. Пусть строка S состоит из n символов. Необходимо выбрать место разреза так, чтобы получающаяся линейная строка была лексически наименьшей среди всех n возможных линейных строк, созданных разрезанием S. Назовем получившуюся строку *канонической*.

Эта задача имеет многочисленные применения. Так в геометрии она может применяться для установления подобия многоугольников[lyn.ps], в теории графов – сравнение последовательности степеней двух циклов [lyn.ps]. Также она встречается в химических базах данных для циклических молекул. Каждая такая молекула представляется циклической строкой химических символов; поэтому для ускорения поиска удобно хранить каждую такую молекулу в виде *канонической* линейной строки. Простая циклическая молекула сама бывает частью более сложной молекулы, так что задача может встречаться во “внутреннем цикле” более сложных задач химического поиска и сравнения.

Методы решения данной задачи за линейное время существуют, – например, алгоритм Дюваля[3]. Здесь будет предложен еще один простой метод решения данной задачи, а также будет продемонстрировано, как решать эту задачу с помощью суффиксных деревьев.

Подход, использующий суффиксные деревья.

Алгоритм решения данной задачи с помощью суффксных деревьев необычайно прост и элегантен.

Алгоритм.

1. Разрезать циклическую строку S в любом месте, образовав линейную строку L;
2. Приписать L саму к себе, получив строку LL;
3. Построить суффиксное дерево для S;
4. Пройти по самому левому пути в построенном суффиксном дереве до достижения строковой глубины ≥ n символам;
5. Вывести первые n символов пути.

Теорема. Приведенный алгоритм корректен.

Доказательство. Ясно, что самый левый путь кодирует самую длинную строку. Остается показать, что самый левый путь содержит не менее n символов. Действительно, если предположить, что самый левый путь содержит менее n символов, то он кодирует суффикс из второй копии L. Но мы можем его продолжить суффиксом, который начинается в той же позиции в первой копии L. Поэтому самый левый (и, вообще говоря, любой представляющий для нас интерес) путь имеет не менее n символов. ∎

Линейность приведенного алгоритма складывается из линейного по времени построения суффиксного дерева для строки LL и спуска в этом дереве вдоль самого левого пути за время, пропорциональное длине строки S.

Подход, не использующий суффиксные деревья.

Здесь будет приведен еще один простой алгоритм линеаризации циклической строки. Во-первых, мы предполагаем, что строка S разразана в кааом-то месте – так мы получим строку L. Припишем строку L к себе в конец – получим LL – как и в случае с суффиксными деревьями. Теперь допустим, что у нас есть некий кандидат (сдвиг) *i* и текущий минимум *k*. Будем сравнивать их до первого несовпадения или до тех пор, когда глубина сравнения достигнет n символов. Пусть глубина сравнения достигла d символов. Не умалая общности, будем считать, что кандидат оказался хуже, чем текущий минимум. Но тогда все строки, начинающиеся в позициях $i + 1..i + d – 1$, не могут быть кандидатами в минимум, т.к. для каждой из этих строк существует соответствующая строка из интервала $k + 1..k + d – 1$, которая заведомо меньше нее. Поэтому проверять эти строки не надо.

Таким образом, суммарная длина сравниваемых символов не превосходит O(n).

Алгоритм.

1. Приписать L к себе;
2. $k \leftarrow 1$;
3. для всех *i* из интервала 2..n, если позиция *i* не помечена, прямым сравнением выяснить, меньше ли *i*-й сдвиг, чем *k*-й. Пусть *s* – лексически больший из сдвигов. Пометить позиции $s + d – 1..s + 1$ до встречи первой помеченной. Если $s \mathrel{!}= k$, $k \leftarrow 1$

Теорема. *Приведенный алгоритм решает задачу линеаризации циклической строки за линейное время.* ∎

Резюме. Как мы видим, суффиксные деревья могут быть эффективно применены и при решении данной задачи с линейными требованиями по времени и по памяти. Однако специфические алгоритмы в данном случае ничуть не уступают суффиксным деревьям по производительности.

Задача. В первой строке входного файла задана строка S. Найти ее минимальный лексикографический сдвиг.

```
#include "..\\func.h"
#include "..\\global.h"
```

```
int st[(MAXN << 1) + MAXN], ll[(MAXN << 1) + MAXN];
char p[MAXN << 1];
/*
st – стек
p – буфер
ll – последний символ, по которому производился поиск от данной вершины
*/

int main() {
	int n, i;
	// читаем строку
	gets(s); n = strlen(s);
	for (i = 0; i < n; i++)
		s[i] -= 'a';
	memcpy(s + n, s, n);
	s[n << 1] = MAXA - 1;
	// строим суффискное дерево
	build(n << 1);
	// ищем самый "левый" путь в дереве длины n с помощью поиска в глубину
	st[i = 0] = 0;
	while (i >= 0) {
		int x = st[i], j;
		if (t[x].d >= n) break;
		for (j = ll[x]; j < MAXA; j++)
			if (t[x].next[j]) {
				st[++i] = t[x].next[j];
				ll[x] = j + 1;
				goto _next;
			}
		i--;
	_next:
		;
	}
	if (i < 0) {
		puts("oblom: not found");
		return 0;
	}
	i = reconstruct(st[i], p);
	//if (i > n) i = n;
	while (n > 0) putchar(p[--i]), n--;
	return 0;
}
```

Задача 73. Сжатие данных по методу зива-лемпеля.

Большие файлы часто сжимаются для уменьшения размера памяти при хранении и времени при пересылке. Это одно из приложений сжатия. Одноко не единственное. Оно также имеет приложения в криптографии и может быть использовано как оценка избыточности данных, которая, в свою очередь, имеет много приложений в машинном обучении.

Популярный метод Зива-Лемпеля[21,22] успешно реализуется с помощью суффиксных деревьев, еще раз иллюстрируя их эффективность.

Определение. Для каждой позиции i в строке S ($|S|= n$) $Prior_i$ – самый длинный префикс $S[i..m]$, встречающийся такжк в $S[1..i - 1]$ в качестве подстроки.

Например, если S=”abaxcabaxabz”, то $Prior_7$=”bax”.

Определение. Для каждой позиции i в S $l_i=|Prior_i|$. Для $l_i > 0$ определим s_i как начальную позицию крайней левой копии $Prior_i$.

В предыдещем примере $l_7=3$ и $s_7=2$.

Метод Зива-Лемпеля использует некоторые значения l_i и s_i для построения сжатого представления строки S: если текст $S[1..i-1]$ уже представлен в сжатой форме и $l_i > 0$, то следующие l_i символов не нужно представлять явно – достаточно описать эту подстроку парой (s_i, l_i) – и продолжать обработку S с символа $i + l_i$.

Алгоритм.

1. $i \leftarrow 1$
2. пока $i \le n$
3. вычислить l_i и s_i;
4. если $l_i > 0$ вывести (s_i, l_i), $i \leftarrow i + l_i$;
5. в противном случае вывести $S(i)$, $i \leftarrow i + 1$.

Например, если S=”abacabaxabz”, алгоритм выведет ab(1,1)c(1,3)x(1,2)z.

Задача. В первой строке входного файла задана строка S. Сжать S по методу Зива-Лемпеля.

```
#include "..\\func.h"
#include "..\\global.h"
#define inf 1000000000

int st[(MAXN << 1) + MAXN]; // стек
char p[MAXN << 1]; // буфер

int main() {
	int n, i;
	// читаем строку
	gets(s); n = strlen(s);
	for (i = 0; i < n; i++)
		s[i] -= 'a';
	s[n] = MAXA - 1;
	// строим суффискное дерево
	build(n);
	// считаем статистику
	st[i = 0] = 0;
	while (i >= 0) {
		int x = st[i];
		if (t[x].l) {
			t[x].cc = n - t[x].d;
			i--;
			continue;
		}
		if (!u[x]) {
			int j;
			for (j = 0; j < MAXA; j++)
				if (t[x].next[j]) st[++i] = t[x].next[j];
			u[x] = 1;
		} else {
			int j;
			for (t[x].cc = inf, j = 0; j < MAXA; j++)
				if (t[x].next[j] && t[x].cc > t[t[x].next[j]].cc)
					t[x].cc = t[t[x].next[j]].cc;
			i--;
		}
	}
	// идем по строке, попутно выводя сжатые куски
	i = 0;
	while (i < n) {
		int j = 0, l = 0;
		while (t[j].d + t[j].cc < i && i + t[j].d < n)
			j = t[j].next[s[i + t[j].d]];
		if (t[j].d + t[j].cc >= i) {
			int k = t[j].d;
			j = t[t[j].p].cc;
			while (j + l < i && s[j + l] == s[i + l]) l++;
		} else
			l = imin(t[j].d, n - i + 1), j = t[j].cc;
		if (l > 0) {
			printf("(%d,%d)", j + 1, l);
			i += l;
		} else
			putchar('a' + s[i++]);
	}
	return 0;
}
```

Задача 74. Нахождение наибольшего префиксного повтора.

Эта задача относится к тому классу задач, для которых до появления суффиксных деревьев линейные алгоритмы известны не были[7]. Но с появлением последних они стали тривиальны, хотя до этого самый эффективный алгоритм для этой задачи имел сложность O(n * log(n)). Надо отметить, что эта задача была для Вайнера одним из побуждений для создания суффиксных деревьев.

Определение. Подстрока α называется *префиксным повтором* строки S, если α является префиксом S и имеет вид ββ для некоторой строки β.

Формулировка. Пусть задана строка S. Необходимо найти наибольший префиксный повтор строки S. Например, если S=”abcabcabcdef”, то наибольшим префиксным повтором, очевидно, будет “abcabc”.

Подход, основанный на суффиксных деревьях.

Здесь мы приведем алгоритм, решающий данную задачу за линейное время.

Рассмотрим ветвь в суффиксном дереве для строки S, кодирующую первый суффикс строки S, т.е. саму строку. Просматривая четные позиции *i* в строке S в нисходящем порядке, будем обходить поддерево, определяемое вершиной *i/2*, и помечать все листовые вершины в этом поддереве меткой *i*. Если окажется, что метка листовой вершины, кодирующей суффикс *i/2*, $\geq i$, то *i* – длина наибольшего префиксного повтора.

Действительно, если метка листовой вершины, кодирующей суффикс *i/2*, $\geq i$, то префикс длинной $\geq i$ гарантированно совпадает с суффиксом *i/2* в первых *i/2* позициях. Поэтому этот префикс будет префиксным повтором. Поскольку значения *i* перебираются в нисходящем порядке, первое значение *i*, удовлетворяющее вышеприведенному условию, и будет ответом к задаче.

Линейность алгоритма очевидна: суффиксное дерево не может содержать более 2n вершин, каждое поддерево обходится ровно один раз.

Алгоритм.

1. Построить суффиксное дерево для S;
2. Двигаясь по пути от листовой вершины, кодирующей первый суффикс, к корню, для всех позиций *i* определить ближайшую снизу вершину на этом пути;
3. Перебирая значения *i* в нисходящем порядке, обходить поддеревья, определяемые позициями *i/2*. Достигнутые при обходе листовые вершины пометить меткой *i*. Остановиться, как только метка листовой вершины, кодирующей суффикс *i/2*, $\geq i$.

Подытоживая все вышесказанное, можно сформулировать теорему:

Теорема. Алгоритм, решающий задачу нахождения наибольшего префиксного повтора, линеен и корректен.

Подход, не основанный на суффиксных деревьях.

Как уже упоминалось, до появления суффиксных деревьев самый быстрый алгоритм решал эту задачу за время O(n * log(n)). Но здесь мы приведем другой алгоритм, который работает за линейное время.

Определение. Строка S называется периодической, если она имеет вид ββ..β для некоторой строки β.

Например, строка S="abcdefabcdefabcdef" – периодическая с периодом β = "abcdef",

а строка S = "ancjifn" – нет.

Определение. Строка S называется полупериодической, если какой-либо ее префикс периодичен.

Здесь мы будем решать задачу нахождения всех периодических префиксов строки S, используя алгоритм Кнута-Морриса-Пратта[11].

Лемма. Строка S длины n периодическая тогда и только тогда, когда $(n - \pi(n)) \mid n$, где π – префикс-функция для строки S.

Доказательство. Если S периодическая, учтверждение леммы очевидно. Обратно: пусть $k = n - \pi(n)$. Тогда так как строки α(префикс) и β(суффикс) длины π(n), суффикс β длины k должен быть также суффиксом α(рис.???). Продложая подобные рассуждения придем к тому, что отрезок длиной k полностью покрывает строку S (т.к. $k \mid n$). Следовательно, строка S периодическая.

Эта лемма дает нам механизм проверки префиксов строки на периодичность. Отсюда сразу вытекает алгоритм поиска наибольшего префиксного повтора:

Алгоритм.

1. Вычислить значения префикс-функции строки S с помощью препроцессинга алгоритма Кнута-Морриса-Пратта;
2. Перебирая позиции *i* в строке S в нисходящем порядке, проверять, является ли префикс *i* периодическим (т.е. проверять $(i - \pi(i)) \mid i$). Если да, то если $i / (i - \pi(i))$ четно, то остановиться и вывести *i* как ответ на задачу.

Теорема. Заданный алгоритм правильно решает задачу о наибольшем префиксном повторе и работает за линейное время.

Доказательство. Действительно, наибольший префиксный повтор – это периодическая строка. Поэтому т.к. алгоритм перебирает все префиксы и проверят их на периодичность, на одной из итераций наибольший префиксный повтор будет найден. Линейность алгоритма следует напрямую из доказательства линейности алгоритма Кнута-Морриса-Пратта.

Резюме. Как видно, и в этой задаче суффиксные деревья представляют довольно удобный метод решения за линейное время. Однако возможен и специальный алгоритм для этой задачи, решающий ее с не меньшей производительностью, но более простой в реализации.

Задача. В первой строке входного файла задана строка S. Найти ее наибольший прификсный повтор.

```
#include "..\\global.h"
#include "..\\func.h"

int idx[MAXN + 1], st[(MAXN << 1) + MAXN], m[MAXN + 1];

int main() {
```

```
	int n, i, z;
	// читаем строку
	gets(s); n = strlen(s);
	for (i = 0; i < n; i++)
		s[i] -= 'a';
	s[n] = MAXA - 1;
	// строим суффиксное дерево
	build(n);
	// ищем лист, соответствующий всей строке
	for (i = last; i >= 0; i--)
		if (t[i].l && n == t[i].d) break;
	// определяем ближайшую снизу вершину
	while (i) {
		int j, r = t[t[i].p].d;
		for (j = t[i].d; j > r; j--)
			idx[j] = i;
		i = t[i].p;
	}
// выполняем третий шаг алгоритма
	memset(m, ~0, sizeof(m));
	for (z = 1, i = n; i > 0; i--)
		if (!(i & 1)) {
			if (m[i >> 1] == -1 && !u[idx[i >> 1]]) {
				int j = 0;
				st[0] = idx[i >> 1];
				while (j >= 0) {
					int x = st[j--], k;
					u[x] = 1;
					if (t[x].l) {
						m[t[x].sid] = i;
						continue;
					}
					for (k = 0; k < MAXA; k++)
						if (t[x].next[k] && !u[t[x].next[k]]) st[++j] =
t[x].next[k];
				}
			}
			if (m[i >> 1] >= i) break;
		}
	printf("%d\n", i);
	return 0;
}
```

Задача 75. Нахождение k-повтора.

Иногда в литературе по анализу последовательностей методы, предназначенные для отыскания интересных свойств биологической последовательности, начинают с каталогизации определенных подстрок длинной строки. Эти методы почти всегда выделяют *окно фиксированной длины* и затем находят все различные строки этой длины. Результат этого оконного, или *q-граммного*, подхода, конечно, очень сильно зависит от выбора длины окна.[7] Это задачи нахождения различных подстрок заданной строки, обладающих теми или иными статистическими свойствами.

Эта задача имеет непосредственное отношение к поднятому вопросу.

Здесь мы приведем пример мощи суффиксных деревьев при решении подобных задач.

Формулировка. По заданной строке S и числу k найти подстроку минимальной длины, встречающаяся в S ровно k раз.

Прежде чем перейти к непосредственно построению алгоритма нахождения наименьшего k-повтора, рассмторим, как в суффиксном дереве хранится информация о различных постороках строки S.

В строке S длины n есть n * (n + 1) / 2 подстрок. Некоторые из них одинаковы и встречаются в строке S больше одного раза. Так как число подстрок равно $O(n^2)$, мы не можем подсчитать, сколько раз появляется в S каждая из них за время O(n). Однако использование суффиксного дерева позволяет получить за такое время *неявное* представление этих чисел. В частности, если задана любая строка P длины m, это неявное представление позволит нам вычислить частоту вхождения P в S.

Определение. Назовем *вершинной частотой* вершины v суффиксного дерева T количество листовых вершин, находящихся в поддереве, корнем которого является v. Обозначим вершинную частоту вершины v через fr(v).

Более формально:

$$fr(v) = \begin{cases} 1, \text{ если v – листовая вершина;} \\ \Sigma\, fr(x), \text{ где x – непосредственный потомок v, в противном случае.} \end{cases}$$

Например, на рисунке 2.1 вершинная частота выршины 1 равна 1, а вершинная частота вершин U и V равна 2. Вершинная частота коневой вершины W равна 6 (по количеству листовых вершин).

Лемма. Пусть для строки S длиной n построено суффиксное дерево T. Задана строка P. Тогда если в T не существует пути, помеченного P, то строка P не встречается в S в качестве подстроки. В противном случае если путь, помеченный P кончается в вершине v или на дуге (u, v), то строка P встречается в S ровно fr(v) раз.

Доказательство. Очевидно, что строка P встречается в S в позиции i тогда и только тогда, когда P является префиксом суффикса S[i..n]. Следовательно, в T существует путь, помеченный P, тогда и только тогда, когда P является подстрокой S.

Далее, если путь, помеченный P, кончается в вершине v, то существует ровно fr(v) суффиксов S, для которых P является префиксом. Отсюда P появляется в S как подстрока ровно fr(v) раз.

Если же путь, помеченный P, кончается в дуге (u, v), то все суффиксы, для которых P является префиксом, находятся в поддереве, корнем которого является v. Отсюда строка P встречается в S ровно fr(v) раз. □

Возвращаясь к примеру со строкой S = “xabxac” и рисунку 2.1 можно сказать, что, например, строки “xa”, “x” и “a” встречаются в S ровно 2 раза. А строка “bxac”, например, встречается в S ровно 1 раз.

Вычислить все вершинные частоты можно с помощью обхода дерева T. Заметим, что время, затрачиваемое на обход дерева T, по лемме 3.1 равно O(n).

Развивая вышеприведенную технику можно также явно найти все вхождения P в S, совершив обход поддерева, корнем которого является v. Время, требуемое на выполнение обхода, равно O(fr(v)).

Теперь у нас есть все, чтобы сформулировать алгоритм решения задачи о наименьшем k-повторе.

Алгоритм.

Построить суффиксное дерево T для строки S.

Вычислить вершинные частоты дерева T.

Для всех вершин v, имеющих вершинную частоту k, выбрать такую вершину f, что ее непосредственный родитель в T имеет наименьшую строковую глубину.

Если шаг 3 завершился неудачей выдать соответствующее сообщение; в противном случае, выдать строку, помечающую путь от корня к родителю f и первый символ дуговой метки от родителя f к f.

Теорема. Алгоритм нахождения наименьшего k-повтора корректен.

Доказательство. Доказательство следует из того, что в суффиксном дереве неявно кодируются все подстроки строки S и леммы 4.1. □

Шаг 1 алгоритма строит суффиксное дерево для строки S. На это необходимо время O(n) (скажем, алгоритм Укконена). Шаг 2 также выполняется за линейное время (это уже обсуждалось в части 4). Шаг 3 просто перебирает все вершины суффиксного дерева. На выполнение этого шага уходит линейное время (лемма 3.1). Шаг 4 просто конкатенирует метки дуг на пти от корня до найденной вершины. На выполнение этого шага в худшем случае также потребуется время O(n).

Таким образом, наименьший k-повтор строки S может быть найден за суммарное время O(n). Затраты памяти опять таки также будут линейными.

Например, для строки S = “abcdefghiabxcdeyfghiabcdpefgqhi” и k = 2 ответом будет строка “bc”.

Задача. В первой строке входного файла задано число k. Во второй строке содержится строка S. Найти k-повтор строки S.

```
#include "..\\global.h"
#include "..\\func.h"

int st[MAXN << 1], is[MAXN << 1];
/*
st - стек
is - последный символ, по которому производился поиск от данной вершины
*/

int main() {
	int i, j, n, k;
	// читаем k и строку s
	scanf("%d\n", &k);
	gets(s);
	for (i = (n = strlen(s)) - 1; i >= 0; i--)
		s[i] -= 'a';
	s[n] = MAXA - 1;
	// строим суффиксное дерево для s
	build(n);
	// считаем статистику (количество потомков-листьев)
	is[0] = st[last = 0] = 0;
	curr = -1;
	while (last >= 0) {
		i = st[last];
```

```
            if (t[i].l) t[i].cc = 1;
            else {
                  j = is[last];
                  if (j) t[i].cc += t[t[i].next[j - 1]].cc;
                  while (j < MAXA) {
                        if (t[i].next[j]) {
                              is[last] = j + 1;
                              st[++last] = t[i].next[j];
                              is[last] = 0;
                              break;
                        }
                        j++;
                  }
                  if (j < MAXA) continue;
            }
            last--;
      // попутно запоминаем максимум
            if (t[i].cc == k)
                  if (curr == -1 || (t[i].sp != n && t[t[i].p].d < t[t[curr].p].d))
                        curr = i;
      }
      // выводим ответ
      if (curr >= 0) {
            printf("%d\n", curr);
            for (j = -1, i = t[curr].p; i; i = t[i].p)
                  st[++j] = i;
            while (j >= 0) {
                  for (i = t[st[j]].sp; i <= t[st[j]].fp; i++)
                        putchar(s[i] + 'a');
                  j--;
            }
            putchar(s[t[curr].sp] + 'a');
      } else
            puts("no such substring");
      return 0;
}
```

Задача 76. Построение k-покрытия строки.

В эукариотах ген составлен из чередующихся *экзонов*, конкатенация которых задает определенный белок, и *интронов*, функция которых не ясна. Сходные экзоны часто наблюдаются в разных генах. Белки нередко формируются из различныз доменов. Одни и те же домены наблюдаются во многих белках, правда, в разном порядке и комбинациях. Естественно предположить, что экзоны соответствуют отдельным белковым доменам. Считается, что все белки, расшифрованные к настоящему времени, составлены из нескольких тысяч экзонов. Этот феномен называется *тасованием экзонов*, а белки, созданные таким тасованием, - мозаичными. Отсюда вытекает следующая задача поиска:

Заданы расшифрованные строки ДНК из кодирующих белок областей, в которых экзоны и интроны неизвестны. Требуется распознать экзоны, находя общие области в двух или более строках ДНК[7].

Можно попытаться решить эту задачу многими методами поиска строк. Среди прочих Гасфилд предлагает задачу нахождения k-покрытия, которую мы рассмотрим здесь.

Формулировка. Пусть заданы две строки, S_1 и S_2 ($|S_1|=m$, $|S_2|=n$), и параметр k. Набор C подстрок S1, каждая из которых длиной не менее k, назовем *k-покрытием*, если S_2 можно представить как конкатенацию подстрок C в каком-то порядке. Заметим, что подстроки C могут перекрываться в S_1, но не в S_2. То есть S_2 является перестановкой строк S_1, каждая из которых длиной не менее k. Требуется построить k-покрытие.

Для решения этой задачи решим задачу подсчета статистики совпадений, обсуждавшуюся в пункте Apl3: орсд, для строк S_1 и S_2. Теперь, зная для каждой позиции в S2 самую длинную подстроку из S1, начинающуюся в этой позиции, нам остается построить алгоритм для провери существования k-покрытия.

Лемма 1. Если $ms(i) > ms(i+1)$, то $ms(i) = ms(i+1) + 1$.

Доказательство. Это напрямую следует из следующего правила: если для позиции *i* искомая подстрока в S_1 начинается в позиции $p(i)$, то для позиции *i+1* искомая подстрока в S_1 начинается в позиции $p(i)+1$. Следовательно, совпадение для позиции *i+1* будет иметь глубину $ms(i)$-1 символов. ↑

Теперь перейдем непосредственно к созданию алгоритма. Будем идти по строке S_2 слева направо и отмечать позиции, для префикса слева от которых существует k-покрытие. Для каждой отмеченной позиции *i* будем помечать все позиции из диапазона *i+k..i+ms(i)*. Однако при малых k и больших значениях $ms(i)$ время работы алгоритма при таком прямолинейном подходе составило бы $O(\sum ms(i))$, что пропорционально $O(n^2)$. Поэтому необходимо помечать нужные нам позиции несколько более разумно.

Наблюдение 1. Во-первых, если позиция *i* помечена и $ms(i+1)<ms(i)$, позицию *i+1* можно смело пропустить. Это прямое следствие леммы 1.

Наблюдение 2. В процессе вычислений можно поддерживать указатель на самую правую помеченную позицию (обозначим его *z*). Если *i+ms(i)* не больше этого значения, то *i* можно пропустить, так как по мере роста *i* левый край помечаемого региона также растет. А тот факт, что $i+ms(i) \le z$, означает, что весь отрезок *i+k..i+ms(i)* лежит внутри другого отрезка, который был помечен на той же итерации, когда была поменеча позиция *z*. В противном случае необходимо пометить все позиции *max(i+k, z)..i+ms(i)* и выполнить присвоение $z \leftarrow i+ms(i)$.

Теперь у нас есть все для создания алгоритма нахождения k-покрытия.

Алгоритм.

1. Решить задачу подсчета статистики совпадений для строк S_1 и S_2;
2. $z \leftarrow 1$, пометить позицию 1;
3. для всех значений *i* из 1..n, если позиция i помечена:
4. если *i-1* помечена и *ms(i-1)>ms(i)* или $i+ms(i) \le z$, перейти к следующему значению *i*;
5. в противном случае пометить все позиции интервала *max(i+k, z)..min(i+ms(i), n+1)*, присвоить $z \leftarrow min(i+ms(i), n+1)$.
6. Если позиция *n+1* помечена, то решение задачи существует, в противном случае – нет.

Теорема. Приведенный алгоритм решает задачу нахождения k-покрытия за линейное время.

Доказательство. Во-первых, перебирая все допустимые подстроки из всех возможных позиций строки S_2, мы всегда найдем k-покрытие, если оно, конечно, существует. Линейность алгоритма следует из наблюдений 1 и 2: мы не помечаем одну и ту же позицию дважды. ↑

Завершая этот пункт, можно сказать, что линейность решения этой задачи основывается на линейности подсчета статистики совпадений, который, в свою очередь, основывается на суффиксных деревьях. Суффиксные деревья и в этой задаче продемонстрируют свою высоую эффективность.

Задача. В первой строке входного файла содержится строка S1. Во второй строке содержится строка S2. В третьей строке задано число k. Найти k-покрытие строки S2.

```
#include "..\\global.h"
#include "..\\func.h"

int wh[(MAXN << 1) + MAXN], st[(MAXN << 1) + MAXN], a[MAXN + 1];
char s1[MAXN + 1], s2[MAXN + 1];
/*
wh - используется в альтернативном подходе к решению задачи из Apl3
a - статистика совпадений (решение задачи из Apl3)
st - обратные ссылки, необходимые для восстановления решения
*/

void restore(int x) { // восстановить ответ
	int i;
	if (x <= 0) return;
	restore(st[x]);
	printf("%d - %d\n", st[x], x - 1);
}

int main() {
	int n1, n2, i, k0;
	// читаем строки s1, s2 и число k
	gets(s1); n1 = strlen(s1);
	gets(s2); n2 = strlen(s2);
	scanf("%d", &k0);
	for (s1[n1] = MAXA - 1, i = 0; i < n1; i++)
		s1[i] -= 'a';
	for (s2[n2] = MAXA - 1, i = 0; i < n2; i++)
		s2[i] -= 'a';
	// добавляем строки s1 и s2 в суффиксное дерево
	add(s1, n1);
	add(s2, n2);
	// считаем статистику
	traverse(0);
	// решаем задачу из Apl3
	memset(wh, ~0, sizeof(wh));
	for (wh[0] = 0, i = last; i > 0; i--)
		if (wh[i] == -1) {
			int j, t0;
			for (t0 = -1, j = i; j && wh[j] == -1 && t[j].f != 3; j = t[j].p)
				st[++t0] = j;
			if (wh[j] == -1) wh[j] = j;
			while (t0 >= 0) wh[st[t0--]] = j;
		}
	for (i = last; i > 0; i--)
```

```
            if (t[i].l && (t[i].f & 2))
                  a[n2 - t[i].d] = t[wh[i]].d >= k0 ? t[wh[i]].d : -1;
      // инициализируем st
      memset(st, ~0, sizeof(st));
      st[0] = -1;
      for (i = k0; i <= a[0]; i++)
            st[i] = 0;
      // движемся вправо по строке
      i = 1;
      while (i < n2) {
            while (i < n2 && a[i] + 1 == a[i - 1]) i++;
            while (i < n2 && st[i] == -1) i++;
            if (i < n2) { // st[i] != -1 !!!
                  int j;
                  for (j = k0; i + j <= n2 && j <= a[i]; j++)
                        st[i + j] = i;
                  i++;
            }
      }
      // если мы не дошли до последнего символа, решения нет
      if (st[n2] == -1) {
            puts("no k-cover found");
            return 0;
      }
      // восстанавливаем решение
      restore(n2);
      return 0;
}
```

Задача 77. Задача выбора праймера в пцр.

Задача выбора праймера для ПЦР – еще одно применение суффиксных деревьев.

ПЦР(полимеразная цепная реакция) – это лабораторная техника для амплификации, т.е. создания множества копий, части ДНК. Например, из небольшого фрагмента ДНК на месте преступления с помощью ПЦР можно создать множество его копий для секвенцирования и определения его владельца. ПЦР также используется для проведения анализов на наличие тех или иных вирусов и бактерий при диагностике различных заболеваний.

Задача выбора праймера часто встречается в молекулярной биологии и имеет непосредственное отношение к ПЦР. Примером является “хождение по хромосоме” – техника, используемая в некоторых методах секвенцирования ДНК или локализации генов (хождение по хромосоме использовалось при локализации гена муковисцидоза в седьмой хромосоме человека[7]). Здесь мы рассмотрим только приложение, относящееся к расшифровке ДНК.

Целью секвенцирования является определение полной последовательности нуклеотидов в длинной строке ДНК. Надо заметить, что в данное время существующие лабораторные технологии позволяют расшифровывать только небольшие участки ДНК длиной от 300 до 500 нуклеотидов за раз, но имеется возможность копировать подстроки длинной строки ДНК, начиная почти с любого места, зная лишь небольшое число нуклеотидов слева от этой точки. Эту маленькую строку используют в качестве праймера, который добирается в длинной строке до места, комплиментарного ему, и гибридизируется там со длинной строкой. Это создает условия, позволяющие воспроизвести часть исходной строки справа от местонахождения праймера.

Этот процесс лежит в основе метода секвенцирования ДНК. Зная 9 нулеотидов в начале строки, мы можем расшифорвать первых (скажем) 300 нуклеотиов. Используя информацию с конца только что расшифрованной строки, мы можем синтезировать праймер, комплиментарный к последним 9 нуклеотидам строки. Затем скопировать строку, содержащую следующие 300 нуклеотидов, расшифровать ее и продолжать далее.

Однако в приведенном методе есть одна проблема: если строка, состоящая из последних 9 нуклеотидов встречается где-либо еще внутри большой строки, то праймер, возможно, гибридизируется в неправильной позиции. Мы знаем последовательность слева от текущей точки, поэтому мы можем проверить, нет ли в ней строки, комплиментарной синтезированному праймеру. Короче говоря, в качестве праймера нужно выбирать строку, которая не встречается нигде в расшифрованном участке.

Формулировка. Для каждой позиции *i* в строке S найти кратчайшую подстроку, которая начинается в этой позиции и не встречается больше нигде в S.

Подход, не основанный на суффиксных деревьях.

Здесь уместно вспомнить об оконном методе, упомянавшемся в пункте ???. Мы можем просто последовательно найти все подстроки длины 9, 8, …, 1 в строке S и для каждой позиции *i* проверить, совпадает ли соответствующая подстрока, начинающаяся в этой позиции, с какой-либо подстрокой из выделенных. Более того для задачи секвенцирования ДНК с помощью ПЦР это особо удобно, потому что алфавит состоит всего из 4 символов. Т.е. строка длиной 9 нуклеотидов может быть компактно представлена в виде числа из интервала $0..2^{18}-1$, что прекрасно помещается в машинное слово.

Таким образом, нам остается просто сформулировать алгоритм:

Алгоритм.

1. Для всех значений *i* от 9 до 1:
2. Двигая окно длиной *i* вдоль строки, создать список всех подстрок длины *i* строки S;
3. Отсортировать полученный список и удалить в нем дублирующиеся записи;
4. Для каждой позиции *j* строки S проверить наличие подстроки длиной *i*, начинающейся в этой позиции, в списке с помощью бинарного поиска. В случае отрицательного ответа установить искомое значение для позиции *j* равным *i*.

Теорема. Алгоритм находит для каждой позиции *i* строки S длину самой длинной подстроки, начинающейся в позиции *i* и не встречающейся нигде в S. При этом учитываются лишь строки длиной не более 9. Время работы алгоритма – O(9 * n * log(n)).

Доказательство. Первая часть алгоритма очевидна из построения. Сложность алгоритма складывается из сортировки и бинарного поиска. ∎

Подход, основанный на суффиксных деревьях.

Здесь мы получим алгоритм, работающий за линейное время.

Алгоритм.

1. Построить суффиксное дерево для S. Подсчитать количество листовых вершин в поддереве, определяемом каждой внутренней вершиной;
2. Для всех суффиксов строки S:
3. Найти вершину *x*, кодирующую этот суффикс. Пусть *p* – непосредственный предок этой вершины. Тогда искомой строкой будет путевая метка *p* + первый символ дуги, идущей от *p* к *x*.

Теорема. Приведенный алгоритм правильно решает задачу и работает за линейное время.

Доказательство. Действительно, любая внутренняя вершина имеет по крайней мере 2 потомков – поэтому на внутренней вершине уникальная подстрока кончаться не может. Следовательно, она заканчивается на дуге, входящей в листовую вершину. Поскольку мы ищем кратчайшую такую подстроку, достаточно для каждой листовой вершины взять путевую метку ее родительской вершины плюс первый символ дуги, входящей в нее.

Линейность алгоритма очевидна. ∎

Таким образом, мы видим, что суффиксные деревья за счет хорошего “знания” внутренней структуры строки S с легкостью решают за линейное время задачу, которая без использования последних кажется весьма не простой.

Задача. В первой строке входного файла содержится строка S. Необходимо решить задачу о выборе праймера для S.

```
#include "..\\global.h"
#include "..\\func.h"

int ans[MAXN]; // ответ

int main() {
	int n, i;
	// читаем строку
	gets(s); n = strlen(s);
for (i = 0; i < n; i++)
	s[i] -= 'a';
	s[n] = MAXA - 1;
	// строим суффискное дерево
	build(n);
	// обходим все листовые вершины и вычисляем ответ для соответствующего суффикса
	for (i = last; i >= 0; i--)
		if (t[i].l && t[i].sp != n)
			ans[n - t[i].d] = t[t[i].p].d + 1;
	for (i = 0; i < n; i++)
		printf("%d ", ans[i]);
	return 0;
}
```

Задача 78. Задача о наибольшем общем продолжении двух строк.

Задача о наибольшем общем префиксе отдельно не имеет серьезных приложений, однако возникает как вспомогательная во многих классических строковых алгоритмах. Умение вычислить быстро наибольшее общее продолжение двух строк позволяет существенно ускорить многие алгоритмы.

Формулировка. Пусть даны две строки, S_1 и S_2, общей длины n. Затем указывается длинная последовательность пар индексов. Для каждой такой пары (*i, j*) требуется найти длину наибольшей подстроки S_1,

начинающейся в позиции *i* и совпадающей с подстрокой S_2, начинающейся в позиции *j*. Таким образом, мы должны найти длину наибольшего общего префикса *i*-го суффикса строки S_1 и *j*-го суффикса строки S_2.

Можно, конечно, для каждой пары индексов искать длину наибольшего общего продолжения прямым сравнением за время, пропорциональное длине совпадения. Однако в худшем случае это приведет к существенным затратам времени в то время, как мы хотим, чтобы каждый запрос выполнялся за константное время. Более того, потребуем, чтобы время на препроцессинг также было линейным.

В этой проблеме нам сильно поможет *задача о нахождении наименьшего общего предшественника в дереве*. Эта задача формулируется так:

В дереве T с корнем *наименьший общий предшественник* (*least common ancestor - lca*) двух вершин, x и y, - это самая нижняя вершина в T, предшествующая и x, и y. Необходимо для любых заданных x и y определить их *lca*.

Эта задача может быть решена за время с линейным препроцессингом дерева и константными ответами на запросы с помощью алгоритма Шибера-Вишкина[17].

Алгоритм.

1. Построить обощенное суффиксное дерево для S_1 и S_2;
2. Подготовить построенное дерево для ответов за запросы *lca*;
3. Для каждой пары (*i, j*) найти наименьшего обшего предшественника v листовых вершин, кодирущих *i*-й суффикс строки S_1 и *j*-й суффикс строки S_2. Строковая глубина найденной вершины равна длине наибольшего общего продолжения.

Здесь важно то, что строка, помечающая путь от до v, в точности совпадает с наибольшей подстрокой S_1, начинающейся в позиции *i*, и с подстрокой S_2, начинающейся в позиции *j*. Следовательно, строковая глубина v равна длине *наибольшего общего продолжения*, так что на любой запрос о наибольшем общем продолжении можно ответить за константное время.

Задача. В первой и второй строках входного файла содержатся строки S1 и S2 соответственно. В оставшихся строках входного файла содержатся пары чисел (*i*, *j*) – запросы. Для каждого запроса необходимо вычислить длину наибольшей общей подстроки S1 и S2, начинающейся в S1 в позиции *i*, а в S2 – в позиции *j*.

```
#include "..\\global.h"
#include "..\\func.h"
#include "..\\lca.h"
/*
здесь используется модуль lca, который реализует алгоритм поиска наименьшего общего
предшественника со сложностью <O(n), O(1)>
*/

int idx1[MAXN + 1], idx2[MAXN + 1];
char s1[MAXN + 1], s2[MAXN + 1];

// idx1, idx2 – индексы листовых вершин, соответствующих суффиксам строк s1 и s2
int main() {
	int n1, n2, i;
	// читаем строки
	gets(s1); n1 = strlen(s1);
	gets(s2); n2 = strlen(s2);
	for (i = 0; i < n1; i++)
		s1[i] -= 'a';
	s1[n1] = MAXA - 1;
	for (i = 0; i < n2; i++)
		s2[i] -= 'a';
	s2[n2] = MAXA - 1;
	// строим суффиксное дерево для s1 и s2
	add(s1, n1);
	add(s2, n2);
	// инициализируем структуру для ответов на запросы об lca
	init();
	// заполняем idx1 и idx2
	for (i = last; i >= 0; i--)
		if (t[i].l) {
			if (t[i].f & 1) idx1[n1 - t[i].d] = i;
			if (t[i].f & 2) idx2[n2 - t[i].d] = i;
		}
	// читаем запросы и выводим на них ответы
	while (1) {
		int x, y;
		if (scanf("%d %d", &x, &y) < 2) break;
		printf("%d\n", t[lca(idx1[x], idx2[y])].d);
	}
	return 0;
```

```
}
```

Задача 79. Нахождение всех максимальных палиндромов в строке.

Данная задача относится к классу задач поиска интересных последовательностей в ДНК. К таковым можно отнести палиндромы. Палиндромы в строках ДНК интересны не только потому, что их много, но иакже из-за функций, которые они выполняют. Так комплиментарные палиндромы в ДНК и РНК участвуют в транскрипции ДНК, вложенные комплиментарные палиндромы в тРНК позволяют молекуле свернуться в структуру типа клеверного листа за счет создания комплиментарных пар и т.д. [7]

Определение. Подстрока S' строки S, имеющая четную длину k, называется *максимальным палиндромом радиуса k*, если S' читается одинаково в обоих направлениях на k символов, начиная с середины S', а на любое k' > k – не одинаково. Максимальный палиндром S' нечетной длины определяется аналогично после исключения центрального символа S'.

Например, если S="aabactgaaccaat", то и "aba", и "aaccaa" – максимальные палиндромы радиусов1 и 3 соответственно, а каждое вхождение "aa" – максимальный палиндром радиуса 1.

Определение. Строка называется *максимальным палиндромом*, если она является максимальным палиндромом радиуса k для какого-либо k.

Например, в строке "cabaabad" максимальные палиндромы и "aba", и "abaaaba". Любой палиндром содержится в каком-либо максимальном палиндроме с той же средней точкой, так что максимальные палиндромы могут служить для компактного представления всех палиндромов.

Формулировка. Пусть задана строка S длины n. Требуется найти все максимальные палиндромы в S.

Подход, основанный на суффиксных деревьях.

Здесь мы рассмотрим только случай палиндромов четной длины. Для нахождения палиндромов нечетной длины надо будет слегка модифицировать алгоритм для частного случая.

Пусть S^r – перевернутая строка S. Предположим, что в S имеется максимальный палиндром четной длины, середина которого находится сразу после символа i. Пусть k – длина этого палиндрома. Тогда строка длины k, начинающаяся в позиции $i + 1$ строки S, должна совпасть со строкой, начинающейся в позиции $n – i + 1$ строки S^r. Так как наш палиндром максимален, то следующие символы (в позициях $i + k + 1$ и $n – i + k + 1$ соответственно) не совпадают. Отсюда следует, что k равно длине наибольшего общего продолжения для позиций $i + 1$ в S и $n – i + 1$ в S^r. Итак, для любой фиксированной позиции i длина максимального палиндрома (если таковой существует) со средней точкой в i может быть найдена за константное время.

Это приводит к следующему простому алгоритму с линейным временем для нахождения в S всех максимальных палиндромов четной длины:

Алгоритм.

1. По заданной строке S построить перевернутую S^r и обработать обе строки так, чтобы любой запрос о наибольшем общем продолжении разрешался за константное время;
2. Для каждого i от 1 до $n – 1$ разрешить вопрос о наибольшем общем продолжении для пары индексов ($i + 1$, $n – i + 1$) в S и S^r соответственно. Если это продолжение имеет непустую длину k, то имеется максимальный палиндром радиуса k с центром в q.

Таким образом, получился метод со временем O(n). Это время складывается из времени на построение суффиксного дерева для S и S^r и из времени на каждый из O(n) запросов о продолжении за константное время.

Теорема. Все максимальные палиндромы четной длины в строке можно найти за линейное время.

Задача. В первой строке входного файла содержится строка S. Необходимо найти все максимальные палиндромы для S.

```
#include "..\\global.h"
#include "..\\func.h"
#include "..\\lca.h"

char s1[MAXN + 1];
int idx1[MAXN + 1], idx2[MAXN + 1];
// idx1, idx2 - индексы листовых вершин, соответствующих суффиксам строк s1 и s2

int main() {
	int n, i;
	// читаем строку s
	gets(s1); n = strlen(s1);
	for (i = 0; i < n; i++)
		s1[i] -= 'a';
	s1[n] = MAXA - 1;
	// добавляем ее в суффиксное дерево
	add(s1, n);
	// разворачиваем s
	for (i = (n >> 1) - 1; i >= 0; i--)
```

```
            swap(s1[i], s1[n - i - 1]);
        // добавляем ее в суффиксное дерево
        add(s1, n);
        // заполняем idx1 и idx2
        for (i = last; i > 0; i--)
            if (t[i].l) {
                if (t[i].f & 1) idx1[n - t[i].d] = i;
                if (t[i].f & 2) idx2[n - t[i].d] = i;
            }
// инициализируем структуру для ответов на запросы об lca
        init();
        for (i = 0; i + 1 < n; i++) {
            int k;
            // палиндромы четной длины
            k = lca(idx1[i + 1], idx2[n - i - 1]);
            if (t[k].d > 0) printf("e: %d - %d - %d\n", i - t[k].d + 1, i, i + t[k].d);
            // палиндромы нечетной длины
            if (i) {
                k = lca(idx1[i + 1], idx2[n - i]);
                if (t[k].d > 0) printf("o: %d - %d - %d\n", i - t[k].d, i, i + t[k].d);
            }
        }
        return 0;
}
```

В этой главе были рассмотрены суффиксные деревья – мощная структура данных, которая может быть применена во многих различных областях, так или иначе относящихся к задачам на строках. Были представлены некоторые из приложений суффиксных деревьев в разных областях и описана основная алгоритмическая техника, используемая при решении задач, связанных с обработкой строк. Было также приведено сравнение алгоритмов, основанных на суффиксных деревьях, и классических методов, применяемых для решения этих задач. В ряде задач были представлены методы, решающие эти задачи порой с производительностью не хуже, чем у суффиксных деревьев. К таковым относятся задача линеаризации циклической строки, а также задача о наибольшем префиксном повторе.

В дальнейшем, развивая технику суффиксных деревьев, можно получить также впечетляющие результаты при решении многих строковых задач.

ГЛАВА 11. ЭЛЕМЕНТЫ КРИПТОГРАФИИ.

АЛГОРИТМ ЭЛЕКТРОННОЙ ПОДПИСИ НА ЭЛЛИПТИЧЕСКИХ КРИВЫХ

В настоящее время, когда информационные технологии и сетевые инфраструктуры развиваются очень быстро, большую важность обретают эффективные методы и средства безопасной передачи информации. Среди них, особое место занимает стандартизованная американская схема электронной подписи – DSA[1], которая обеспечивает хороший уровень защиты и легкость внедрения и освоения. В данный момент, активно ведутся работы по улучшению этого стандарта и самое последнее достижение в этой сфере – внедрение эллиптических кривых в DSA (ECDSA).

Конечные поля

Конечное поле содержит конечное множество элементов F, с двумя бинарными операциями на F, которые удовлетворяют определенные арифметические свойства. Порядок конечного поля – число элементов этого поля. В криптографии, используются только поля порядка q, где q - степень простого числа. Существует много способов представления элементов F_q.

Если $q = p^m$, где p - простое, а m - положительное целое число, то p называется *характеристикой* F_q, m - *степенью расширения* F_q. Большинство стандартов ограничиваются использованием конечного поля нечетного простого порядка (т.е. $q = p$).

Пусть p - некоторое простое число. Конечное поле F_p, называемое *простым полем* состоит из множества целых чисел $\{1, 2, ..., p-1\}$ со следующими арифметическими операциями:

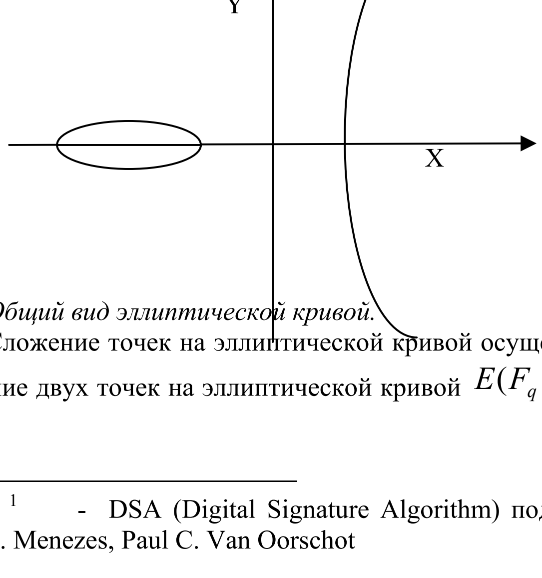
- Если $a, b \in F_p$, то $a + b = r$, где r – остаток от деления $a + b$ на p, и $0 \le r \le p-1$ (сложение по модулю p);
- Если $a, b \in F_p$, то $a \cdot b = s$, где s – остаток от деления $a \cdot b$ на p, и $0 \le s \le p-1$. (умножение по модулю p);
- Если a - ненулевой элемент F_p, то *обращением* числа a по модулю p, является однозначно определенный элемент $c \in F_p$, для которого $a \cdot c = 1$ (обозначается как a^{-1}).

Теперь определим *эллиптические кривые над полем* F_p:

Пусть $p > 0$ будет нечетным простым числом. Эллиптическая кривая E над полем F_p определяется уравнением

(1)

[illegible] $a, b \in F_p$ [illegible] $E(F_p)$ [illegible] O [illegible]



Общий вид эллиптической кривой.

Сложение точек на эллиптической кривой осуществляется по так называемому *правилу хорд и касательных*. Сложение двух точек на эллиптической кривой $E(F_q)$ дает точку на $E(F_q)$. Множество точек $E(F_q)$, вместе с

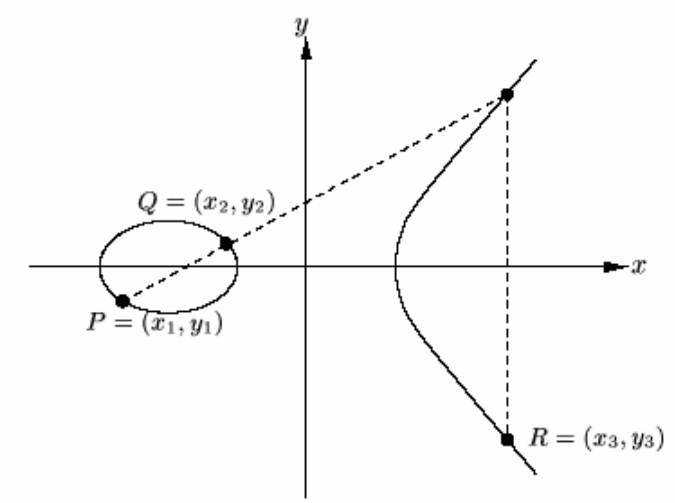
---

[1] - DSA (Digital Signature Algorithm) подробно описан в книге «Handbook of Applied cryptography» Alfred J. Menezes, Paul C. Van Oorschot

бесконечно удаленной точкой (которая играет роль нуля) и операцией сложения образуют группу. Именно такие группы используются для построения криптосистем.

Вышеназванное правило сложения легко объясняется геометрически. Пусть $P=(x_1,y_1)$ и $Q=(x_2,y_2)$ две отдельные точки на $E(F_q)$. Тогда суммой точек P и Q называется третья точка $R=(x_3,y_3)$, которая определяется так: проводим линию через наши точки P и Q, она пересекает кривую в некой третьей точке; симметричное отражение этой точки относительно оси X дает нам сумму - $R=(x_3,y_3)$.(рис.1)

Рисунок 1 (P+Q=R)



Если взять точку $P=(x_1,y_1)$ и провести через нее касательную, то она пересечет эллиптическую кривую в некой третьей точке. Ее симметрия относительно оси X дает нам отдельную точку $R=(x_3,y_3)$. Эта точка называется *удвоением* $P=(x_1,y_1)$ (, понятно что,).

Следующие свойства сложения и удвоения точек легко выводятся из геометрического толкования:

1. для всех $P\in E(F_p)$

2. Если тогда (Точка $(x,-y)$ является точкой противоположной P и обозначается «$-P$», эта точка также лежит на $E(F_p)$).

3. (Сложение точек) Пусть $P=(x_1,y_1)\in E(F_p), Q=(x_2,y_2)\in E(F_p)$ и $P\neq\pm Q$. Тогда , где

$$x_3=\left(\frac{y_2-y_1}{x_2-x_1}\right)^2-x_1-x_2 \; u \; y_3=\left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1-x_3)-y_1.$$

4. (Удвоение точек) Пусть и $P\neq -P$. Тогда $2P=(x_3,y_3)$, где

$$x_3=\left(\frac{3x_1^2+a}{2y_1}\right)^2-2x_1 \; u \; y_3=\left(\frac{3x_1^2+a}{2y_1}\right)(x_1-x_3)-y_1.$$

Основные факты

ПОРЯДОК ГРУППЫ. Пусть E будет эллиптическим кривым над конечным полем F_q. Теорема Хассе [illegible] $|t|\le 2\sqrt{q}$ $\#E(F_q)$ [illegible] E; t [illegible] E [illegible] $E(F_q)$ [illegible]

СТРУКТУРА ГРУППЫ. $E(F_q)$ является абелевой группой и изоморфна к $Z_{n_1}\times Z_{n_2}$, где n_2 делит n_1. Здесь как Z_n обозначена циклическая группа порядка n. Кроме того, n_2 делит $q-1$. В случае, когда n_2=1, $E(F_q)$ называется циклической кривой. В этом случае, $E(F_q)$ будет изоморфна к Z_{n_1} и тогда существует точка $P\in E(F_q)$, такая что $E(F_q)=\{kP:0\le k\le n_1-1\}$. Такая точка называется *порождающим* $E(F_q)$.

Параметры DSA на эллиптических кривых

Параметры ECDSA состоят в основном из «подходяще» выбранной[2] кривой E, определенной над конечным полем F_q характеристики p и базовой точки $G \in E(F_q)$:

1. порядок (размерность) поля - q, где $q = p$ нечетные простые;
2. некая индикация FR (представление поля) – представление применяемое к элементам поля F_q;
3. (необязательный параметр) битовая строка **seedE** длиной не менее 160 бит;
4. два элемента поля $a, b \in F_q$, которые определяют уравнение эллиптической кривой E над F_q (т.е. в случае $p > 3$ и в случае $p = 2$);
5. два элемента: $x_G, y_G \in F_q$, которые определяют конечную точку $G = (x_G, y_G)$ простого порядка[3] в $E(F_q)$;
6. Порядок n точки G (т.е. nG, $nG = \mathrm{O}$), $n > 2^{160}$, $n > 4\sqrt{q}$ и
7. кофактор

Нужно сказать, что к выбору этих параметров также предъявляются особые требования, в целях обеспечения надлежащего уровня безопасности.

Совместно с ЭЦП обычно применяются хэш-функции. Они служат для того, чтобы помимо аутентификации отправителя, обеспечиваемой ЭЦП, гарантировать, что сообщение не имеет искажений, и получатель получил именно то сообщение, которое подписал и отправил ему отправитель.

Хэш-функция — это процедура обработки сообщения, в результате действия которой формируется строка символов (дайджест сообщения) фиксированного размера. Малейшие изменения в тексте сообщения приводят к изменению дайджеста при обработке сообщения хэш-функцией. Таким образом, любые искажения, внесенные в текст сообщения, отразятся в дайджесте. Алгоритм применения хэш-функции заключается в следующем: 1. перед отправлением сообщение обрабатывается при помощи хэш-функции. В результате получается его сжатый вариант (дайджест). Само сообщение при этом не изменяется и для передачи по каналам связи нуждается в шифровании; 2. полученный дайджест шифруется закрытым ключом[4] отправителя (подписывается ЭЦП) и пересылается получателю вместе с сообщением; 3. получатель расшифровывает дайджест сообщения открытым ключом отправителя; 4. получатель обрабатывает сообщение той же хэш-функцией, что и отправитель и получает его дайджест. Если дайджест, присланный отправителем, и дайджест, полученный в результате обработки сообщения получателем, совпадают, значит, в сообщение не было внесено искажений. Существует несколько широко применяемых хэш-функции: MD5, SHA-1 и др. Математически, хэш-функция - это функция, которую просто вычислить в прямом направлении, но невозможно просчитать в обратном направлении: пусть $y = f(x)$, если f - безопасная хэш-функция с входным значением x и выходным y, тогда значение y для данного x вычисляется быстро и просто, но найти x' такое, что $y = f(x')$ для данного y крайне трудно.

Генерация пары ключей в ECDSA. Каждая сторона должна сделать следующее:

Выбрать случайное целое число d в интервале $[1, n-1]$.

Вычислить $Q = dG$.

Закрытым ключем будет d, а открытым Q.

Генерация подписи ECDSA. Для того, чтобы подписать сообщение m, сторона А (Алиса), используя множество параметров и пару ключей (d, Q) делает следующее:

1. Выбирает случайное целое число .

---

[2] Подразумевается грамотный выбор кривой, которая может обеспечить хороший уровень стойкости ЭЦП к атакам. К таким кривым предъявляются особые требования. Формирование и выбор таких кривых весьма непростое занятие и имеет свои каноны.

[3] Порядок циклической подгруппы группы точек эллиптической кривой (число точек в подгруппе) называется порядком точки эллиптической кривой. Точка $P \in E(F_p)$ называется точкой порядка q, если:

$qP = O$, где q – наименьшее натуральное число, при котором выполняется данное условие.

[4] Открытый и закрытый ключи – пара чисел, участвующие в шифровке и расшифровке. В ЭЦП, закрытый ключ используют для шифрования сообщения, а открытый для расшифровки. Однако есть другие криптографические алгоритмы, где назначение этих ключей обратно противоположно.

2. Вычисляет $kG = (x_1, y_1)$ и конвертирует[5] x_1 в целое $\overline{x_1}$.
3. Вычисляет $r = x_1 \bmod n$. Если $r = 0$, начинает обратно с п.1.
4. Вычисляет $k^{-1} \bmod n$.
5. Вычисляет $SHA-1(m)$ и конвертирует эту битовую строку в целое e.
6. Вычисляет . Если $s = 0$, начинает обратно с п.1.
7. Подписью для сообщения m будет пара (r, s).

Верификация подписи. Получив сообщение m с подписью (r, s) от Алисы, В (Bob) взяв проверенную копию параметров Алисы: и связанный с ними открытый ключ, проделывает следующее:

1. Проверяет, что r и s лежат в интервале $[1, n-1]$.
2. Вычисляет $SHA-1(m)$ и конвертирует эту битовую строку в целое e.
3. Вычисляет .
4. Вычисляет и .
5. Вычисляет .
6. Если $X = O$, отклоняет подпись. В противном случае, конвертирует X - x_1 в целое $\overline{x_1}$ и вычисляет $v = \overline{x_1} \bmod n$.
7. Принимает подпись тогда и только тогда, если $v = r$.

Теперь, используя все вышесказанное можно перейти непосредственно к полученным результатам. Суть ее – возможность подбора такого значения закрытого ключа, чтобы получить абсолютно идентичные подписи к разным сообщениям. Хоть эта ошибка и не актуальна, при грамотном выборе параметров, но при определенном стечении обстоятельств может быть достаточно опасной.

Все получается из равенства x - координат противоположных точек эллиптической кривой:
$x_G = x_{-G}$.

Из свойств конечного поля, легко увидеть, что: . Отсюда:
, где k - случайный ключ из алгоритма генерации подписи.

Мы можем подобрать закрытый ключ d так, что подписи для $m1$ и $m2$ будут совпадать:

Пусть , тогда . Теперь вспомнив п.6 **Генерации подписи ECDSA**, можем написать: , где $k_1^{-1} * k_1 \bmod n = 1; \quad k_1^{-1} = 1;$
$k_2^{-1} * (n - k_1) \bmod n = 1; \quad k_2^{-1} = n - 1$

и $e_1 = SHA-1(m_1), \quad e_2 = SHA-1(m_2)$. Теперь очевидно, что $s' = s'' = s$ если
$(e_1 + dr) = (n-1) * (e_2 + dr) \pmod n \Rightarrow 2dr = (n-1)(e_1 + e_2) \pmod n \Rightarrow d = \lambda(n-1)(e_1 + e_2) \bmod n$, где.

Избежать этой ошибки не сложно, нужно всего лишь взять как подпись не (s, r), а (s, R), где $R = kG$.

---

[5] Стандарт США ANSI X9.62 подробно описывает методы конвертирования элементов поля в целые числа. Там же даются методы для конвертирования битовых строк в целые.

Задача 80. Дискретный логарифм.

Входной файл input.txt
Выходной файл output.txt
Ограничение по времени 2 секунды
Ограничение по памяти 64 мегабайта

Вам даны числа g, p, b ($1 \le g, b < p \le 10^5$). Известно, что g – генератор группы Z_p^*, т.е. любое число a из интервала [1..p-1] представимо в виде $a = g^x \pmod p$ для некоторого x ($0 \le x \le p$), причем x называется дискретным логарифмом a по основанию g. Требуется найти дискретный логарифм b по основанию g.

Входные данные:

В первой строке входного файла записаны числа g, p, b ($1 \le g, b < p \le 10^5$).

Выходные данные:
В единственной строке выходного файла выдайте искомый логарифм.

Пример:

input.txt	output.txt
2 3 5	3
3 3 3	1

Комментарий.
Приведем здесь метод “baby step/giant step”, который позволит нам решить эту задачу.
Пусть a = ceil(√p), $L_1 = \{1, g^a, g^{2a}, \ldots, g^{a^2}\}$, $L_2 = \{b, bg, bg^2, \ldots, bg^{a-1}\}$. Найдем число z, которое встречается как в L_1, так и в L_2. $z \equiv bg^k \equiv g^{ln} \pmod p$, $b \equiv g^{la-k} \equiv g^x \pmod p$, $x \equiv la - k \pmod{p-1}$. Такое число всегда найдется, поскольку $x = x_1 * a + x_2$, где $0 \le x_1, x_2 < a$. Тогда $y = g^x = g^{x1 * a + x2}$. Умножив обе части равенства на g^{a-x2}, получим $yg^{a-x2} = g^{(x1+1)*a}$. Левая часть равенства принадлежит L_1, правая – L_2.
Найти такое число тоже не представляет сложности: отсортируем списки L_1 и L_2. Теперь, последовательно перебирая элементы L_1, будем искать их в L_2 с помощью бинарного поиска.

БИБЛИОГРАФИЯ.

[1] Baeza-Yates, R. and G. Gonnet. (1992). A new approach to text searching. In *Communications of the ACM*; 35:74-82.

[2] Chang W.I., Lawler E.L. (1994). Sublinear expected time approximate string matching and biological applications. // Algoritmica, Vol. 12, P. 327-344

[3] Duval, J. P. (1983). Factoring words over an finite alphabet. J. Algorithms 4 (1983), 363-381.

[4] Felsenstein, J., S. Sawyer, and R. Kochin. (1982). An efficient method for matching nucleic acid sequences. In *Nucleic Acids Research, Vol 10*:133-39.

[5] Fisher, M. and M. Paterson. (1974). String matching and other products. In Complexity of Computation*, SIAM-AMS Proceedings 7*; 113-25.

[6] Gusfield, Dan. (1996). Simple uniform preprocessing for linear time string matching. In *Technical Report CSE-96-5, University of California Davis. Dept. of Computer-Science.*

[7] Gusfield, Dan. (1997). Algorithms on Strings, Trees, and Sequences. Гасфилд Д. «Строки, деревья и последовательности в алгоритмах».- СПб.: Невский диалект; БХВ-Петербург, 2003;

[8] Horspool, N. (1980). Practical fast searching in strings. In *Software Practice and Experience*; 10:501-6.

[9] Hume, A. and D. M. Sunday. (1991). Fast string searching. In *Software Practice and Experience*; 21:1221-48.

[10] Karp, R. and M. Rabin. (1987). Efficient randomized pattern matching algorithms. In *IBM Journal of Research and Development, Vol. 31, No. 2*; 31:249-60.

[11] Knuth, D. E., J. H. Morris, and V.B. Pratt. (1977). Fast Pattern Matching in Strings. In *SIAM Journal of Computing*; 6:323-50.

[12] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2002.

[13] McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. In *Journal of the Association for Computing Machinery*; 23:262-72.

[14] Moritz Maaβ. Suffix trees and their applications. Techniche Universität München Facultät für Informatik, 1999;

[15] Smith, P. D. (1991). Experiments with a very fast substring search algorithm. In *Software Practice and Experience*; 21:1065-74.

[16] Smith, P. D. (1994). On tuning the Boyer-Moore-Horspool string searching algorithm. In *Software Practice and Experience*; 24:435-36.

[17] Schiber B., Vishkin U (1988). On finding lowest common ancestors: simplifications and parallelization. SIAM J. Comput. Vol. 17, P. 1253-1262

[18] Sunday, D. M. (1990). A very fast substring search algorithm. In *Communications of the ACM*; 33:132-42.

[19] Ukkonen, E. (1995). On-line construction of suffix-trees. In *Algorithmica*; 14:249-60.

[20] Weiner, P. (1973). Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*; 1-11.

[21] Ziv J., Lempel A. (1977) A universal algorithm for sequential data compression. IEEE Trans. on Info. Theory. Vol. 23, P. 337-343

[22] Ziv J., Lempel A. (1978) Compression of individual sequences via variable length coding. IEEE Trans. on Info. Theory. Vol. 24, P. 530-536

[23] Alfred J. Menezes, Paul C. Van Oorschot. Handbook of Applied cryptography. – CRC Press, 1992. – 425c.

[24] Don Johnson, Alfred Menezes, Scott Vanstone. The Elliptic Curve Digital Signature Algorithm. – Waterloo: Certicom, 2001.

[25] Мусиралиева Ш. Ж. Прикладная криптография. – Алматы: Print S, 2004.

[26] Gordon D. A survey of fast exponentation methods. – Journal of algorithms №27, 1998. – 128 – 150c.

[27] Романовский И.В. «Дискретный анализ».- СПб.: Невский диалект; БХВ-Петербург, 2004