Зачёт

A. Корень кубического уравнения

Разбор

Кубический многочлен – непрерывная функция, поэтому если у него есть один корень x_0, то либо

$$\forall x < x_0 : f(x) < 0 \wedge \forall x > x_0 : f(x) > 0$$

либо

$$\forall x < x_0 : f(x) > 0 \wedge \forall x > x_0 : f(x) < 0$$

В обоих случаях функция $sgn(f(x))$ – монотонная и равна нулю в точке x_0, а значит её можно найти с помощью вещественного бинпоиска.

В качестве границ бинпоиска возьмём числа, которые заведомо лежат по разные стороны от x_0.

Решение

```
eps = 0.0000001


def f(x):
    return a * x ** 3 + b * x ** 2 + c * x + d


def root():
    left = -2000
    right = 2000
    while right - left > eps:
        middle = (right + left) / 2
        if f(middle) > 0:
            right = middle
        else:
            left = middle
    return (left + right) / 2


a, b, c, d = map(float, input().split())
if a < 0:
    a = -a
    b = -b
    c = -c
    d = -d
print(root())
```

B. Введите одностороннее движение

Разбор

Если в графе есть мосты, то ввести одностороннее движение точно нельзя, т.к. концы этого моста точно окажутся в разных компонентах сильной связности. Если же мостов в графе нет, то достаточно запустить DFS и все древесные рёбра ориентировать вниз по дереву, а все обратные – вверх. Теперь от корня дерева можно добраться до любой вершины, и если мы докажем, что от любой вершины можно добраться до корня, то докажем, что полученный граф сильно связен.

Пусть от какой-то вершины v нельзя добраться до корня. Пусть самый высокий её предок, до которого можно добраться — u. Тогда древесное ребро, ведущее в u — мост. Противоречие.

Осталось заметить, что на самом деле проверять граф на наличие мостов необязательно: достаточно ориентировать рёбра описанным выше способом, затем развернуть их и убедиться, что из корня DFS достижимы все остальные вершины. Для этого нужно запустить DFS на графе с инвертированными рёбрами.

Получили решение с асимптотикой $\mathcal{O}(V + E)$

```
import sys

sys.setrecursionlimit(200000)

def bridge(v):
    global timer
    low[v] = timer
    visitV[v] = timer
    timer = timer+1
    for i in graph[v]:
        if visitE[i[1]]:
            continue
        visitE[i[1]] = True
        if v == l_e[i[1]][0]:
            direction[i[1]] = True
        else:
            direction[i[1]] = False
        nv = i[0]
        if visitV[nv] == -1:
            bridge(nv)
            low[v] = min(low[v], low[nv])
            if low[nv] > visitV[v]:
                global bridges
                bridges = bridges + 1
        else:
            low[v] = min(low[v], low[nv])


n = int(input())
m = int(input())
s = []
e = set()
l_e = [-1] * m
visitV = [-1] * n
visitE = [False] * m
low = [-1] * n
direction = [-1] * m
bridges = 0
timer = 0
graph = [[] for _ in range(n)]
for i in range(m):
    u, v = [int(i) - 1 for i in input().split()]
    graph[u].append((v, i))
    graph[v].append((u, i))
    l_e[i] = (u, v)
for i in range(n):
    if visitV[i] == -1:
        bridge(i)

if bridges == 0:
    for i in range(m):
```

```
        u = l_e[i][0]
        v = l_e[i][1]

        if direction[i]:
            print(u + 1, v + 1)
        else:
            print(v + 1, u + 1)
else:
    print(0)
```

C. Потерянные пробелы

Разбор

Будем считать динамическое программирование $dp[i]$ = можно ли префиксы длины i разбить на слова из словаря, и если можно, то какое слово будет стоять последним. Если теперь для каждой длины префикса пытаться подобрать слово из словаря, которое будет стоять в конце, то получим TL. Вместо этого заметим следующее: нам важно не столько, какое слово будет стоять в конце нашего префикса, сколько его длина.

Ключевое наблюдение: если сумма длин строк равна L, то количество различных длин строк асимптотически оценивается как $\mathcal{O}(\sqrt{L})$. Действительно, в лучшем случае различные длины слов – это ряд $1, 2, \ldots, maxlen$. Но $1 + 2 + \ldots + maxlen = \mathcal{O}(maxlen^2) \leqslant L$, откуда и следует предыдущее утверждение.

Следовательно, нам достаточно перебирать порядка $\sqrt{L}$ различных длин для каждого префикса $s[:i]$. Единственное, нам осталось научиться понимать, является ли подстрока $s[i - len : i]$ словом из словаря. Для этого нам достаточно завести map/dict, в котором запомним для каждого хеша слова из словаря, какому слову этот хеш соответствует.

Получили решение за $\mathcal{O}(n \times \sqrt{L})$

```
#include <bits/stdc++.h>

using namespace std;

#define FAIL() ((*(int*)(0))++)

const int MAXN = 200010;
const int64_t MOD = 1000000007;
const int64_t BASE = 43;

int64_t add(int64_t a, int64_t b) {
    return (a + b) % MOD;
}

int64_t sub(int64_t a, int64_t b) {
    return (a - b + MOD) % MOD;
}

int64_t mul(int64_t a, int64_t b) {
    return (a * b) % MOD;
}

int n;
string s;
int64_t pw[MAXN];
int64_t h[MAXN];
bool f[MAXN];
int nxt[MAXN];
unordered_set<int64_t> words;
unordered_set<int> sizes_set;
vector<int> sizes;

int64_t get_sub(int l, int r) {
    if (l == 0) {
        return h[r];
    } else {
        return sub(h[r], mul(h[l - 1], pw[r - l + 1]));
    }
}

void solve() {
    int nwords;
    cin >> nwords;
    for (int i = 0; i < nwords; ++i) {
        string word;
        cin >> word;
        if (!sizes_set.count(word.size())) {
            sizes_set.insert(word.size());
            sizes.push_back(word.size());
```

```
        }
        int64_t hsh = 0;
        for (char ch : word) {
            hsh = add(ch - 'a' + 1, mul(hsh, BASE));
        }
        words.insert(hsh);
    }
    std::sort(sizes.begin(), sizes.end());
    cin >> s;
    n = s.size();
    pw[0] = 1;
    h[0] = s[0] - 'a' + 1;
    for (int i = 1; i < n; ++i) {
        h[i] = add(s[i] - 'a' + 1, mul(h[i - 1], BASE));
        pw[i] = mul(pw[i - 1], BASE);
    }

    f[n] = true;
    for (int pos = n - 1; pos >= 0; --pos) {
        f[pos] = false;
        for (auto sz : sizes) {
            int npos = pos + sz;
            if (npos > n) {
                break;
            }
            if (!f[npos]) {
                continue;
            }
            auto hsh = get_sub(pos, npos - 1);
            if (words.count(hsh)) {
                f[pos] = true;
                nxt[pos] = npos;
                break;
            }
        }
    }
    if (!f[0]) {
        FAIL();
    }
    int space = nxt[0];
    for (int i = 0; i < n; i = nxt[i]) {
        cout << s.substr(i, nxt[i] - i) << ' ';
    }
}

int main() {
#ifdef _MY_DEBUG
    freopen("input.txt", "r", stdin); freopen("output.txt", "w", stdout);
#endif

    ios_base::sync_with_stdio(false); cin.tie(0);

    solve();

    return 0;
}
```

D. Защищенное соединение

Разбор

Добавим фиктивную стартовую вершину, которую соединим со всеми вершинами первой компании ребром веса 0. Точно так же добавим фиктивную конечную вершину, которую соединими со всеми вершинами второй компании ребром веса 0. Заметим, что в получившемся графе кратчайший путь между стартовой и конечной вершиной будет соответствовать кратчайшему пути между какими-то двумя вершинами первого и второго множества. Осталось запустить Дейкстру на получившемся графе из стартовой вершины.

Получили решение за $\mathcal{O}((E+V)\log(V))$

```
#include <iostream>
#include <vector>
#include <string>
#include <numeric>
#include <cassert>
#include <algorithm>
#include <cmath>
#include <set>
#define INF 1791791791
using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;
    vector<int> a(n);
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    vector<vector<pair<int, int>>> g(n);
    for (int i = 0; i < m; ++i) {
        int s, e, c;
        cin >> s >> e >> c;
        --s, --e;
        g[s].push_back({ e,c });
        g[e].push_back({ s,c });
    }
    vector<bool> used(n, false);
    set<pair<int, int>> cur;
    vector<int> dist(n, INF);
    for (int i = 0; i < n; ++i) {
        if (a[i] == 1) {
            cur.insert({ 0, i });
            dist[i] = 0;
        }
    }
    vector<int> pr(n, -1);
    while (cur.size()) {
        int x = cur.begin()->second;
        cur.erase(cur.begin());
        used[x] = true;
        for (auto e : g[x]) {
            if (!used[e.first] && dist[e.first] > e.second + dist[x]) {
                cur.erase({ dist[e.first], e.first });
                dist[e.first] = dist[x] + e.second;
                cur.insert({ dist[e.first],e.first });
                pr[e.first] = x;
            }
        }
    }
    int answer = INF;
    int y1, y2;
    for (int i = 0; i < n; ++i) {
        if (used[i] && a[i] == 2 && answer > dist[i]) {
            answer = dist[i];
            y1 = i;
        }
    }
```

```
    if (answer == INF)
        cout << -1;
    else {
        y2 = y1;
        while (pr[y2] != -1)
            y2 = pr[y2];
        cout << y2 + 1 << " " << y1 + 1 << " " << answer;
    }
}
```

E. Type Printer

Разбор

В этой задаче достаточно добавить все слова в бор и обойти его рекурсивно. Спускаясь по ребру, мы печатаем букву, поднимаясь – убираем. Заходя в вершину, соответствующую слову, мы его печатаем. Можно доказать, что такая процедура вывода слов оптимальна по количеству операций, но доказательство этого факта оставляется читателю.

```
#include <iostream>
#include <memory.h>
#include <queue>
#include <string>
#include <cstdio>
using namespace std;

#define HASFILES 1

#define TASK ololo

const int N = 25000;

struct vertex
{
    bool finish;
    vertex* Q[26];
    vertex()
    {
        finish = 0;
        memset(Q, 0, sizeof(Q));
    }
} *root = new vertex;

int cnt = 1;

void add(string& s)
{
    vertex* cur = root;
    for (int i = 0; i < s.size(); i++)
    {
        if (cur->Q[s[i] - 'a'] != NULL)
            cur = cur->Q[s[i] - 'a'];
        else
            cur = (cur->Q[s[i] - 'a'] = new vertex);
    }
    cur->finish = 1;
}
string mx;
vertex* mxv[N];
queue<char> ans;
void go(vertex* cur, int deep)
{
    if (cur->finish)
        ans.push('P');

    for (int i = 0; i < 26; i++)
    {
        if (mx[deep] == i + 'a' && cur == mxv[deep])
            continue;
        if (cur->Q[i] != NULL)
        {
            ans.push(i + 'a');
            go(cur->Q[i], deep + 1);
            ans.push('-');
        }
    }
    if (cur == mxv[deep])
    {
        ans.push(mx[deep]);
        go(cur->Q[mx[deep] - 'a'], deep + 1);
```

```
        return;
    }

}

int main(int argc, char **argv)
{
    int n;
    cin >> n;
    string t;
    for (int i = 0; i < n; i++)
    {
        cin >> t;
        add(t);
        if (t.size() > mx.size())
            mx = t;
    }
    vertex* cur = root;
    for (int i = 0; i < mx.size(); i++)
        mxv[i] = cur, cur = cur->Q[mx[i] - 'a'];

    go(root, 0);
    cout << ans.size() << '\n';
    while (!ans.empty())
    {
        cout << ans.front() << '\n';
        ans.pop();
    }
    return 0;
}
```