1 A. Отказоустойчивость сети

1.1 Условие

В стране K существует сеть атомных электростанций, связанных линиями электропередач. Ивану, как главному инженеру энергетической сети, нужно определить её показатель отказоустойчивости. Этот показатель равен минимальному количеству линий электропередач, которое необходимо для вывода одной станции из сети.

Помогите Ивану рассчитать этот показатель.

1.2 Входные данные

В первой строке даны два целых числа: N и M ($1 \leq N \leq 10^6, 1 \leq M \leq 10^6$) - количество атомных станций и количество линий электропередач соответственно.

В следующих M строках записаны пары чисел a_i и b_i ($1 \leq a_i, b_i \leq N$), каждая из которых описывает линию электропередачи, соединяющую станции a_i и b_i ($a_i \neq b_i$).

Гарантируется, что каждая пара электростанций имеет не более одного соединения.

1.3 Вывод

Выведите одно целое число - показатель отказоустойчивости данной энергосети.

1.4 Пример входных данных

Sample Input:
5 5
1 2
1 3
2 4
3 4
4 5
Sample Output:
1

Примечание В данном примере минимальная степень вершины равна 1, так как станция 5 связана только с одной другой станцией (станцией 4).

1.5 Решение

В данной задаче необходимо было найти вершину с наименьшей степенью и вывести эту степень.

```
n, m = map(int, input().split())
d = [0] * n

for i in range(m):
    a, b = map(int, input().split())
    d[a-1] += 1
    d[b-1] += 1

print(min(d))
```

2 B. Оптимальная последовательность

2.1 Условие

На атомной электростанции требуется провести плановую замену топливных стержней в ядерном реакторе. В реакторе находятся n стержней, каждый из которых имеет свой остаточный энергетический потенциал p_i и уровень радиоактивности r_i. Извлечение стержней необходимо проводить по одному, и при этом нужно минимизировать суммарный показатель опасности в процессе замены.

Показатель опасности при извлечении i-го стержня вычисляется как произведение уровня радиоактивности этого стержня r_i на сумму остаточных энергетических потенциалов всех стержней, которые еще не были извлечены на момент извлечения i-го стержня (не учитывая сам i-й стержень).

Требуется найти такой порядок извлечения топливных стержней, при котором суммарная радиоактивность, получаемая персоналом, будет минимальной.

2.2 Входные данные

В первой строке задано целое число n ($1 \leq n \leq 9$) - количество топливных стержней в реакторе. В следующих n строках через пробел заданы два целых числа p_i и r_i ($1 \leq p_i, r_i \leq 100$) - остаточный энергетический потенциал и уровень радиоактивности i-го стержня соответственно.

2.3 Вывод

Выведите единственное целое число - минимальный возможный суммарный показатель опасности, получаемый при оптимальном порядке извлечения топливных стержней.

2.4 Пример входных данных

Sample Input:
3
10 5
20 10
30 15
Sample Output:
550

Примечание
Оптимальный порядок извлечения стержней в примере: 1, 2, 3.
Показатель опасности на каждом шаге:
Шаг 1: $5 * (20 + 30) = 250$

Шаг 2: $10 * 30 = 300$
Шаг 3: $15 * 0 = 0$
Итого: $250 + 300 = 550$

2.5 Решение

Ограничения на n позволяют сделать перебор всех возможных последовательностей за $O(n!)$.

```
import itertools

n = int(input())
p = [0] * n
r = [0] * n
for i in range(n):
    (p[i], r[i]) = map(int, input().split())

answers = []
for var in itertools.permutations(range(n)):
    ans = 0
    psum = sum(p)
    for num in var:
        psum -= p[num]
        ans += r[num] * psum
    answers.append(ans)

print(min(answers))
```

3 C. Задачка на строки

3.1 Условие

Дано дерево на N вершинах. На каждом ребре записана строчная латинская буква.
Простой путь в дереве - это такой путь, который не проходит через одну и ту же вершину более одного раза. Другими словами, все вершины в простом пути уникальны.

Дано число K ($K \leq (N-1)$). Среди всех простых путей в дереве с K ребрами найдите лексикографически минимальный, а также их количество. В лексикографическом пути элементы сравниваются по алфавиту.

Лексикографическое сравнение двух строк − это способ определения того, какая из двух строк должна идти раньше в словарном порядке. Алгоритм сравнивает строки посимвольно слева направо, и если на какой-то позиции символы различаются, то меньшей считается строка с меньшим символом. Если одна строка является префиксом другой, меньшей считается более короткая строка.

3.2 Входные данные

В первой строке вводится одно число N ($2 \leq N \leq 3000$) - число вершин дерева.

Следующие $(N-1)$ строк содержат 2 числа a_i, b_i ($1 \leq a_i, b_i \leq N$, $a_i \neq b_i$) и символ c_i (c_i - строчная латинская буква) - концы ребра и букву, записанную на ребре.

В последней строке вводится одно число K ($1 \leq K \leq (N-1)$) - количество ребер в искомых простых путях.

3.3 Вывод

В первой строке выведите одно число - количество лексикографически минимальных простых путей в дереве. Выведите 0, если нет ни одного простого пути из K ребер.

Во второй строке выведите без пробелов последовательность символов на лексикографически минимальном пути в дереве. Если нет ни одного простого пути из K ребер, ничего во вторую строку выводить не нужно.

3.4 Пример входных данных

Sample Input:
5
1 2 c
1 3 c
1 4 c
4 5 d
2
Sample Output:
6
cc

Sample Input:
5
1 2 c
1 3 c
1 4 c
4 5 d
3
Sample Output:
2
ccd

Sample Input:
5
1 2 c
1 3 c
1 4 c
4 5 d
4
Sample Output:
0

Sample Input:
6
1 2 c
2 3 b
3 4 b
4 5 b
5 6 c
5

Sample Output:
2
cbbbc

3.5 Решение

Базовая идея решения - обход в глубину. Запустим его от каждой вершины. При обходе запоминаем получающуюся строку. Как ее длина достигла K - обновляем ответ.

Однако для полного балла необходимо сделать несколько оптимизаций.

Будем хранить строку оптимально. При проходе по ребру добавляем в конец за O(1) новый символ, при выходе из поддерева его удаляем за O(1). Это можно сделать, используя связный список или std::vector в c++, который позволяет удалять и добавлять последний элемент за O(1).

Будем прекращать обход, если нет шанса получить минимальный путь. Давайте сохраним строку, которая будет равна лексикографически минимальному пути. Если добавление очередного символа делает путь лексикографически больше оптимального - обход прекращаем.

Например, наилучший ответ на данный момент - строка $abracadabra$. Если текущий путь ab, а новый символ z, тогда нет смысла обходить все пути, начинающиеся с abz.

При этом сложность решения все еще $O(n^3)$, ведь в худшем случае произойдет n^2 сравнений строк и n^2 копирований строк при выборе оптимального ответа, однако, такие решения набирали полный балл.

Будем сравнивать строки при помощи хешей. Таким образом, сложность копирования снижается до $O(n^2 * log(n))$.

Докажем, что количество копирований будет $O(log(n))$. И в таком случае, итоговая сложность решения будет $O(n^2 * log(n))$.

Доказательство факта из пункта выше. Возможен случай, при котором мы честно n^2 раз скопируем ответ, однако вероятность этого события крайне мала. Мы должны рассмотреть все пути в порядке их лексикографического убывания. Оценим математическое ожидание числа копирований при случайном порядке появлений $O(n^2)$ путей длины k. Копирование происходит, когда встречается путь меньше, чем оптимальный на данный момент.

Допустимб - всего 1 путь, тогда мы его скопируем 1 раз. Допустимб - 2 пути, тогда мы с вероятностью $\frac{1}{2}$ встретим сразу оптимальный путь, с вероятностью $\frac{1}{2}$ выберем сначала больший путь, потом меньший. Тогда произойдет $\frac{1}{2} * 1 + \frac{1}{2} * 2 = 1.5$ копирования. То есть с вероятностью $\frac{1}{2}$ мы свели задачу к подсчету числа вариантов из 0 путей, с вероятностью $\frac{1}{2}$ - к задаче подсчета числа вариантов из 1 пути.

Пусть a_n - мат. ожидание числа копирований при выборе из n путей. Тогда $a_0 = 0$, $a_1 = 1$, $a_2 = \frac{1}{2} * (a_0 + 1) + \frac{1}{2} * (a_1 + 1) = 1 + \frac{1}{2}(a_0 + a_1)$. Аналогично получится $a_3 = 1 + \frac{1}{3}(a_0 + a_1 + a_2)$. Искомое значение $a_n = 1 + \frac{1}{n}(a_0 + a_1 + ... + a_{n-1})$.

Введем $b_n = a_0 + a_1 + ... + a_n$.

Тогда для $n \geq 1$ $b_n = b_{n-1} + a_n$, $b_0 = 0$.

Тогда $a_n = 1 + \frac{1}{n}(a_0 + a_1 + ... + a_{n-1}) = 1 + \frac{1}{n}b_{n-1}$.

А значит, $b_n = b_{n-1} + a_n = b_{n-1} * \frac{n+1}{n} + 1$.

Подставим туда выражение для b_{n-1}. $b_n = (1 + \frac{n}{n-1} * b_{n-2}) * \frac{n+1}{n} + 1 = 1 + \frac{n+1}{n} + \frac{n+1}{n-1}b_{n-2}$.

Выделим целую часть из 2-го слагаемого: $2 + \frac{1}{n} + \frac{n+1}{n-1}b_{n-2}$.

Подставим туда выражение для b_{n-2}: $2 + \frac{1}{n} + \frac{n+1}{n-1}(1 + \frac{n-1}{n-2}b_{n-3}) = 2 + \frac{1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2}b_{n-3} = 3 + \frac{1}{n} + \frac{2}{n-1} + \frac{n+1}{n-2}b_{n-3}$

Подставляя дальше, придем к сумме вида $n + \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + ... + \frac{n-1}{2}$ (помним, что $b_n = 0$).

Докажем, что сумма вида $\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + ... + \frac{1}{2} < log_2(n)$ также известная как гармонический ряд.

Тогда наша сумма $n + \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + ... + \frac{n-1}{2} =$
$n + (\frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2}) + (\frac{1}{n-1} + \frac{1}{n-2} + ... + \frac{1}{2}) + (\frac{1}{n-2} + \frac{1}{n-3} + ... + \frac{1}{2}) + ... + 1/2$
$\leq n + (n-1) * (\frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2})$
$\leq n + (n-1) * log(n)$ - что есть $O(n * log(n))$.

Доказательство $\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + ... + \frac{1}{2} < log_2(n)$.
$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} \leq \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + ... \leq 2 * \frac{1}{2} + 4 * \frac{1}{4} + ... + 2^k * \frac{1}{2^k} = 1 + 1 + .. + 1 = k$, где $k = log_2(n)$ округленное вверх.

Вредный совет: после первых двух оптимизаций можно было на глазок оценить, что копирований и сравнений будет не очень много. Написать, получить свои 100 баллов и идти дальше решать задачи. Ведь оптимизировать надо в первую очередь результат своего участия. Заметьте, Вы как участник олимпиады, всегда оптимизируете алгоритм ровно до момента, что время его работы будет меньше ограничения по времени. Вы не пытаетесь оптимизировать решение с $O(n^2)$ до $O(n * log(n)$, если $O(n^2)$ с запасом успевает. Так что не обязательно было доказывать этот факт для сдачи этой задачи :) Однако для математической строгости и общего развития это доказательство было приведено.

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <algorithm>
#include <cmath>
#include <cassert>

using namespace std;

using ll = long long;

////////////////////////////////////////////////////////////////////

const int MAXN = 3000;

const ll P = 31;
const ll MOD = 1e9+7;

class THashString {
    vector<ll> my_hash = {0};
public:
    string s;
    void push_back(char x) {
        s.push_back(x);
        ll add = x - 'a';
        my_hash.push_back((my_hash.back() * P + add) % MOD);
    }
    void pop_back() {
        s.pop_back();
        my_hash.pop_back();
    }
    bool operator <(const THashString& other) const {
        assert(s.size() == other.s.size());
        int lb = 0;
        int rb = my_hash.size();
        while (rb - lb > 1) {
            int mid = (lb + rb) >> 1;
            if (my_hash[mid] == other.my_hash[mid]) {
                lb = mid;
            } else {
                rb = mid;
            }
        }
        int pos_diff = lb;
        return pos_diff < s.size() && s[pos_diff] < other.s[pos_diff];
    }
    bool operator ==(const THashString& other) const {
        assert(s.size() == other.s.size());
        return my_hash.back() == other.my_hash.back();
    }
};

vector<pair<int, char>> g[MAXN];
int K;


THashString opt_ans;
ll opt_count = 0;
void registerAnswer(const THashString& s) {
    if (opt_count == 0 || s < opt_ans) {
        opt_ans = s;
        opt_count = 1;
    } else if (s == opt_ans) {
        ++opt_count;
    }
}

void dfs(int v, int p, THashString& s, bool already_better) {
    if (opt_count > 0 && s.s.size() && !already_better && opt_ans.s[s.s.size() - 1] < s.s[s.s.size() - 1]) {
        return;
    }
    if (s.s.size() == K) {
        registerAnswer(s);
    }
    if (s.s.size() >= K) {
        return;
    }
    for (auto [to, let] : g[v]) {
        if (to != p) {
            s.push_back(let);
            dfs(to, v, s, (already_better || opt_count == 0 || s.s.back() < opt_ans.s[s.s.size() - 1]));
            s.pop_back();
        }
    }
}

void solve() {
```

```
    int n;
    cin >> n;
    for (int i = 1; i < n; ++i) {
        int a, b;
        char c;
        cin >> a >> b >> c;
        --a;
        --b;
        g[a].push_back({b, c});
        g[b].push_back({a, c});
    }
    cin >> K;
    THashString s;
    for (int i = 0; i < n; ++i) {
        dfs(i, -1, s, false);
    }
    if (opt_count) {
        cout << opt_count << endl << opt_ans.s << endl;
    } else {
        cout << "0" << endl;
    }
}

////////////////////////////////////////////////////////////////////

signed main() {
    cin.tie(nullptr);
    cout.tie(nullptr);
    ios_base::sync_with_stdio(false);
    int t = 1; // cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}

////////////////////////////////////////////////////////////////////
```

4 D. Необычное исследование

4.1 Условие

Исследуя поведение частиц, лаборант Гриша разработал новый метод - сталкивать частицы под углом. Эксперимент проводится на квадратной платформе размером N на N клеток. Клетка $(1, 1)$ находится в нижнем левом углу, ось X направлена вправо, ось Y - вверх.

В ходе эксперимента частицы влетают на платформу с левой и нижней сторон. Каждая частица движется с постоянной скоростью: частицы, влетающие слева, движутся вправо со скоростью 1 клетка в единицу времени, а частицы, влетающие снизу, движутся вверх с той же скоростью.

Гриша записал время влета каждой частицы на платформу, но из-за высокой скорости частиц он не смог определить, какие частицы столкнулись. Столкновение происходит, когда две частицы оказываются в одной клетке одновременно.

Задача: Для каждой частицы определите, столкнулась ли она с другой частицей. Если столкновение произошло, укажите, с какой именно частицей.

Частицы после столкновения исчезают.

Частицы нумеруются следующим образом:
- частицы, летящие горизонтально, нумеруют сверху вниз от 1 до N.
- частицы, летящие вертикально, нумеруются слева направо с N+1 до 2 * N.

4.2 Входные данные

В первой строке дано число N $(2 \leq N \leq 10^6)$ - размер платформы.

Во второй строке даны N чисел v_i $(0 \leq i \leq N-1)$, $(1 \leq v_i \leq 10^9)$ - момент времени, когда на платформу слева влетела частица в клетку (1, N - i).

В третьей строке даны N чисел h_j $(1 \leq j \leq N)$, $(1 \leq h_j \leq 10^9)$ - момент времени, когда на платформу снизу влетела частица в клетку $(j, 1)$.

4.3 Вывод

Для каждой частицы выведите информацию о частице, с которой она столкнулась, или -1, если частица пролетела платформу без столкновений.

В первой строке через пробел выведите N чисел Av_i $(0 \leq i \leq N-1)$ - ответ для частицы, которая начала своё движение вправо в клетке $(1, N-i)$.

Во второй строке через пробел выведите N чисел Ah_j $(1 \leq j \leq N)$ - ответ для частицы, которая начала своё движение вверх в клетке $(j, 1)$.

4.4 Пример входных данных

Sample Input:
3
1 2 3
1 2 5
Sample Output:
-1 4 6
2 -1 3

Примечание
Группы тестов
1) ($2 \leq N \leq 10^3$) (35 баллов)
2) Без дополнительных ограничений (65 баллов)

4.5 Решение

Сначала найдём для каждой частицы, влетающей слева, её высоту h.
Зная высоту и время влёта частицы на поле tv_i, мы можем посчитать, во сколько должна влететь самая левая частица, влетающая на поле снизу, чтобы пересечься с данной. Заметим, что для второй слева частицы необходимое время влёта для пересечения будет отличаться на 1.
Тогда будем идти по вертикально летящим частицам слева направо, обрабатывая их столкновения с горизонтально летящими.

```
n = int(input())

tv = list(map(int, input().split()))
th = list(map(int, input().split()))

ansv = [-1] * n
ansh = [-1] * n
who = {}
for i in range(n):
    h = n - i - 1
    who.setdefault(tv[i] - h, []).append(i)

for i in range(n):
    can = who.get(th[i] - i, [])
    if can:
        x = can.pop()
        ansv[x] = n + i + 1
        ansh[i] = x + 1
print(*ansv)
print(*ansh)
```

5 E. Задачка на числа

5.1 Условие

Дан массив из N чисел и два числа K и X.

Подпоследовательность — это последовательность, которая может быть получена из другой последовательности путём удаления некоторого количества элементов без изменения порядка оставшихся.

Например, если у нас есть последовательность [1, 2, 3, 4, 5], возможные подпоследовательности включают [1, 3, 5], [2, 4], [1, 2, 3] и многое другое. Пустые подпоследовательности в данной задаче не рассматриваются.

Первая последовательность лексикографически меньше второй, если при поэлементном сравнении этих двух последовательностей слева направо, первый элемент, в котором они отличаются, меньше в первой последовательности.

Найдите K-тую лексикографическую подпоследовательность массива с произведением X.

Наивное решение можно представить так.

1. Выпишем все подпоследовательности исходного массива.

1. Оставим только те, произведение элементов которых равняется X.

1. Лексикографически отсортируем эти подпоследовательности. K-ый по счету элемент будет ответом.

Гарантируется, что у числа X будет не более 2000 делителей.

5.2 Входные данные

В первой строке вводятся 3 числа N, K, X ($1 \le N \le 3000$), ($1 \le K, X \le 10^{18}$).

Во второй строке вводятся N чисел a_i ($2 \le a_i \le 10^{18}$) - элементы массива.

5.3 Вывод

В единственной строке вывести N чисел - K-тую лексикографическую подпоследовательность исходного массива с произведением X или -1, если такой нет.

5.4 Группы

Группа 1. Гарантируется, что ($N \le 11$). За эту группу Вы получите 5 баллов.

Группа 2. Гарантируется, что ($N \le 100$) и что все a_i различны. За эту группу Вы получите 11 баллов. Будет тестироваться только при прохождении группы 1.

Группа 3. Гарантируется, что ($N \leq 1000$) и что все a_i различны. За эту группу Вы получите 17 баллов. Будет тестироваться только при прохождении группы 2.

Группа 4. Без дополнительных ограничений. Будет тестироваться только при прохождении группы 3.

Пояснение к 1 примеру из условия. Всего есть 5 подпоследовательностей массива с произведением 16. Перечислим их в лексикографическом порядке. 2, 2, 2, 2, 2, 2, 4, 2, 4, 2, 4, 2, 2, 4, 4

5.5 Пример входных данных

Sample Input:
6 3 16
2 2 4 2 2 4
Sample Output:
2 4 2

Sample Input:
6 3 36
2 3 4 6 9 12
Sample Output:
4 9

Sample Input:
6 3 16
2 2 4 2 2 4
Sample Output:
2 4 2

Sample Input:
2 2 49
7 7
Sample Output:
-1

5.6 Решение

Для решения первой группы достаточно перебрать все подмножества из 2^n возможных, посчитать произведение в каждом, оставить те, произведение которых равняется X, отсортировать их как строки и вывести требуемую по счету подпоследовательность. При этом есть 2 ловушки.

Произведение чисел может выйти за пределы целочисленного типа. Безопасно будет проверять, что X делится на очередное число без остатка, а затем X делить на это число. Если после проверки всех чисел подпоследовательности X станет единицей, то это значит, что произведение чисел равняется 1.

Числа в массиве могут повторяться, каждую подпоследовательность нужно учесть 1 раз. Можно сложить все получившиеся варианты в *set* перед сортировкой для исключения повторов.

Рассмотрим идею полного решения. Допустим, X делится на 2, и в массиве есть двойка. Тогда лексикографически минимальными подпоследовательностями будут те, что начинаются с 2. Это будут все подпоследовательности с произведением $\frac{X}{2}$, собранные из чисел правее самой левой двойки.

Пусть следующий по возрастанию делитель X это 5. Тогда следующими в сортировке подпоследовательностей будут идти те, что начинаются с 5. И так далее.

Допустим, у нас есть десять последовательностей, начинающихся с 2, семь - начинающихся с 5. Какое первое число будет в подпоследовательности для $K = 12$? 5, так как первые 10 начинаются с 12, а 11-я, 12-я,...15-ая начинаются с 5.

Это значит, что нам нужна 2-я подпоследовательность с произведением $\frac{X}{5}$ из чисел, расположенных правее самой левой пятерки. Получили ту же задачу, но массив короче, а X меньше. Опытные олимпиадники увидят аналогии со спуском по дереву при поиске K-той единички.

Применим метод динамического программирования, будем двигаться справа налево, вычисляя $dp[suf][mult]$ - сколько подпоследовательностей существует из последних suf элементов массива с произведением $mult$. Причем имеет смысл рассматривать только такие $mult$, которые являются делителями X, которых не более 2000 по условию.

База динамики $dp[n][1] = 1$. Пересчет динамики. $dp[suf][mult] = dp[suf + 1][mult] + dp[suf + 1][mult/a[suf]]$, если $mult$ делится на $a[suf]$ - делитель X, иначе $dp[suf][mult] = dp[suf + 1][mult]$. За этим переходом логика очень простая. Надо сложить число вариантов для случая, когда наше число попадает в считаемые подпоследовательнсти, и когда не попадает. Сложность подсчета $O(n*divs(X))$ или $O(n*divs(X)*log(divs(X)))$, где $divs(X)$ - число делителей X, она зависит от реализации.

Восстановление ответа. Изначально $suf = 0, Xnow = X, Know = K$. Вычислим первое число ответа, как в нескольких абзацах выше. $suf =$ (ближайшая справа позиция выбранного числа), $Xnow =$ (выбранное число), а K изменяется в процессе выбора числа[1].

[1] Некоторые детали опускаются как очевидные для человека, собирающегося решить последнюю задачу на полный балл.

Последнее препятствие на пути к полному баллу - повторы чисел. Пересчет динамики выше будет корректным для случая, когда все числа различны. Прибавлять надо не $dp[suf+1][mult/a[suf]]$, а $dp[suf+1][mult/a[suf]] - dp[next_position[a[suf]]][mult/a[suf]]$. Рассмотрим массив [2, 3, 3, 2, 9]. Пересчитывая по неправильной формуле правую двойку, dp[3][18] = 1, а левую dp[0][18] = dp[1][18] + dp[1][9] = 1 + 2 = 3, т.е. 2 9 посчитано дважды! Логически, мы посчитаем дважды подпоследовательности, которые начинаются с текущего числа Z и дальше состоят из чисел после следующего справа вхождения Z. Остается реализовать.

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <algorithm>
#include <cmath>
#include <cassert>

using namespace std;

using ll = long long;

////////////////////////////////////////////////////////////////////

const ll INF = 2e18+1;

void add(ll& a, ll b) {
    a = min(a + b, INF);
}

ll safe_mult(ll a, ll b) {
    if ((INF + b - 1) / b <= a) {
        return INF;
    }
    return min(a * b, INF);
}

void solver(int n, vector<ll>& v, ll K, ll X) {
    vector<map<ll, ll>> pathsToFinishCount(n + 1);
    map<ll, ll> numberPrevPos;
    vector<pair<int, pair<ll, ll>>> numberPrevPosEvents; // position; keyChanged; prevNumber;
    map<ll, map<ll, ll>> graph; // product; nextValue, itsPosition;
    vector<pair<pair<ll, ll>, pair<ll, ll>>> graphEvents; // position; outKeyChanged; inKeyChanged; diff;
    pathsToFinishCount[n][X] = 1;
    for (int pos = n - 1; pos >= 0; --pos) {
        pathsToFinishCount[pos] = pathsToFinishCount[pos + 1];
        for (const auto& [product, pathsCount] : pathsToFinishCount[pos + 1]) {
            if (product % v[pos] == 0) {
                ll prevPaths = 0;
                if (numberPrevPos.find(v[pos]) != numberPrevPos.end()) {
                    int prevPos = numberPrevPos[v[pos]];
                    prevPaths = pathsToFinishCount[prevPos + 1][product];
                }
                add(pathsToFinishCount[pos][product / v[pos]], pathsCount - prevPaths);
                { // graph[product / v[pos]][v[pos]] = pos;
                    if (graph[product / v[pos]].find(v[pos]) != graph[product / v[pos]].end()) {
                        graphEvents.push_back(
                            {{pos, product / v[pos]}, {v[pos], graph[product / v[pos]][v[pos]]}});
                    } else {
                        graphEvents.push_back({{pos, product / v[pos]}, {v[pos], -1}});
                    }
                    graph[product / v[pos]][v[pos]] = pos;
                }
            }
        }
        { // numberPrevPos[v[pos]] = pos;
            if (numberPrevPos.find(v[pos]) != numberPrevPos.end()) {
                numberPrevPosEvents.push_back({pos, {v[pos], numberPrevPos[v[pos]]}});
            } else {
                numberPrevPosEvents.push_back({pos, {v[pos], -1}});
            }
            numberPrevPos[v[pos]] = pos;
        }
    }
    --K;
    if (pathsToFinishCount[0][1] <= K || X == 1) {
        cout << "-1\n";
        return;
    }
    ll nowValue = 1;
    int nowPos = -1;
    vector<ll> ans;

    { // Debug print
        // for (int pos = 0; pos <= n; ++pos) {
        //     for (const auto& [nextValue, counts] : pathsToFinishCount[pos]) {
        //         cout << "(" << nextValue << "; " << counts << ") ";
        //     }
        //     cout << endl;
        // }
    }

    while (nowValue < X) {
        { // Debug print
            // cout << nowValue << " " << endl;
            // for (const auto& [new_value, inps] : graph) {
            //     cout << new_value << ": ";
            //     for (const auto& [prev_value, pos] : inps) {
            //         cout << "(" << prev_value << "; " << pos << ") ";
```

```
            //      }
            //      cout << endl;
            // }
        }

        for (const auto& [nextValue, itsPosition] : graph[nowValue]) {
            ll newProduct = safe_mult(nowValue, nextValue);
            /* cout << nowValue << " " << nowPos << " " << newProduct << " "
            << pathsToFinishCount[itsPosition + 1][newProduct] << " " << itsPosition << " " << K << endl; */
            if (pathsToFinishCount[itsPosition + 1][newProduct] <= K) {
                K -= pathsToFinishCount[itsPosition + 1][newProduct];
            } else {
                ans.push_back(nextValue);
                { // Rollback numberPrevPos to nowPos + 1;
                    while (numberPrevPosEvents.size() && numberPrevPosEvents.back().first <= nowPos) {
                        // Rollback numberPrevPos before nowPos
                        if (numberPrevPosEvents.back().second.second != -1) {
                            numberPrevPos[numberPrevPosEvents.back().second.first] =
                                numberPrevPosEvents.back().second.second;
                        } else {
                            numberPrevPos.erase(numberPrevPosEvents.back().second.first);
                        }
                        numberPrevPosEvents.pop_back();
                    }
                    if (newProduct != X) {
                        assert(numberPrevPos.find(nextValue) != numberPrevPos.end());
                    }
                }
                nowPos = numberPrevPos[nextValue];
                nowValue = newProduct;
                { // Rollback graph to nowPos
                    while (graphEvents.size() && graphEvents.back().first.first <= nowPos) {
                        // Rollback graph to nowPos
                        if (graphEvents.back().second.second != -1) {
                            graph[graphEvents.back().first.second][graphEvents.back().second.first] =
                                graphEvents.back().second.second;
                        } else {
                            graph[graphEvents.back().first.second].erase(graphEvents.back().second.first);
                        }
                        graphEvents.pop_back();
                    }
                }

                break;
            }
        }
    }
    for (auto x : ans) {
        cout << x << " ";
    }
    cout << endl;
}

void solve() {
    int n;
    ll K, X;
    cin >> n >> K >> X;
    assert(1 <= n && n <= 3000);
    assert(1 <= K && K <= 1e18);
    assert(1 <= X && X <= 1e18);
    vector<ll> v(n);
    for (int i = 0; i < n; ++i) {
        cin >> v[i];
        assert(2 <= v[i] && v[i] <= 1e18);
    }
    solver(n, v, K, X);
}

////////////////////////////////////////////////////////////////////////

signed main() {
    cin.tie(nullptr);
    cout.tie(nullptr);
    ios_base::sync_with_stdio(false);
    int t = 1; // cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}

////////////////////////////////////////////////////////////////////////
```