2024 ICPC Brazil Subregional Programming Contest

> 난이도: A E FLH KB IC JDG

A. Attention to the Meeting

$N+(N-1)x \le K$가 성립하는 가장 큰 정수 x를 구하는 문제로, $\lfloor \frac{K-N+1}{N} \rfloor$을 출력하면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N, K; cin >> N >> K;
    cout << (K - N + 1) / N;
}
```

B. Bacon Number

영화를 정점으로 두고, 같은 배우가 출연한 두 영화를 간선으로 연결한 그래프를 생각해 봅시다. i번째 배우가 출연한 영화들을 $A_i = \{A_{i,1}, A_{i,2}, \cdots, A_{i,k}\}$라고 하면, 문제에서 주어지는 쿼리 x, y는 A_x에 속한 어떤 한 정점에서 A_y에 속한 어떤 한 정점으로 가는 경로를 아무거나 하나 찾는 것이라고 생각할 수 있습니다. 이렇게 출발점과 도착점이 여러 개인 상황에서 경로를 찾는 것은 multi-source BFS를 사용하는 것이 편합니다. BFS를 시작할 때 출발점을 모두 큐에 넣고 시작한 다음, 도착점 중 어느 한 정점이라고 방문하면 종료하도록 구현하면 됩니다.

다만, 그래프에 간선이 너무 많다는 사소한 문제점이 있습니다. 이 문제는 최단 경로를 찾지 않아도 되므로 연결성만 보존된다면 간선을 모두 저장하지 않아도 됩니다. 즉, 어떤 배우가 출연한 영화가 c개 있을 때 $c(c-1)/2$개의 간선을 모두 추가할 필요 없이, 인접한 두 영화($A_{i,j-1}$과 $A_{i,j}$)를 연결하는 간선만 추가해도 충분합니다. 또한, 같은 두 정점 쌍을 연결하는 간선은 하나만 있어도 충분하기 때문에 $O(M+\sum n_i)$ 시간에 최대 $O(N^2)$개의 간선을 모두 만들 수 있습니다.

따라서 정점이 N개, 간선이 최대 $O(N^2)$개인 그래프를 $O(M+\sum n_i)$ 시간에 구축한 뒤, multi-source BFS를 이용해 쿼리를 매번 $O(N^2)$에 처리하면 $O(QN^2+M+\sum n_i)$ 시간에 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, M, G[111][111];
vector<int> Actor[1010101];
void AddEdge(int u, int v, int i){ G[u][v] = G[v][u] = i; }

int D[111], P[111], End[111];
vector<int> MultiBFS(vector<int> src, vector<int> snk){
    memset(End, 0, sizeof End);
    for(auto i : snk) End[i] = 1;

    queue<int> Q;
    memset(D, -1, sizeof D);
```

```
    for(auto i : src) Q.push(i), D[i] = 0, P[i] = -1;

    int ed = -1;
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        if(End[v]){ ed = v; break; }
        for(int i=1; i<=N; i++){
            if(G[v][i] && D[i] == -1) Q.push(i), D[i] = D[v] + 1, P[i] = v;
        }
    }
    if(ed == -1) return {};

    vector<int> res;
    for(int i=ed; i!=-1; i=P[i]) res.push_back(i);
    reverse(res.begin(), res.end());
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++){
        int c; cin >> c;
        for(int j=1,t; j<=c; j++) cin >> t, Actor[t].push_back(i);
    }
    for(int i=1; i<=M; i++){
        for(int j=1; j<Actor[i].size(); j++) AddEdge(Actor[i][j-1], Actor[i][j],
i);
    }

    int Q; cin >> Q;
    for(int q=1; q<=Q; q++){
        int s, t; cin >> s >> t;
        auto path = MultiBFS(Actor[s], Actor[t]);
        if(path.empty()){ cout << "-1\n"; continue; }
        cout << path.size() + 1 << "\n";
        cout << s << " " << path[0] << " ";
        for(int i=1; i<path.size(); i++) cout << G[path[i-1]][path[i]] << " " <<
path[i] << " ";
        cout << t << "\n";
    }
}
```

C. Couple of BipBop

수열 $A = \{A_1, A_2, \cdots, A_N\}$이 주어지면, 두 명의 사람이 수열의 접미사(suffix)를 선택하는 총 N^2가지 방법에 대해, 두 사람이 선택한 접미사의 최장 공통 접두사(longest common prefix)의 길이의 합을 구하는 문제입니다.

수열 A의 suffix array sa와 lcp array lcp를 생각해 보면, 각각 사전 순으로 i, j번째인 접미사의 lcp 길이는 $\min(lcp_{i+1}, lcp_{i+2}, \cdots, lcp_j)$로 계산할 수 있습니다. 따라서 이 문제의 정답은 lcp의 모든 구간의 최솟값의 합을 구한 뒤 2를 곱하고 $N(N+1)/2$를 더한 것과 같습니다.

"모든 구간의 최솟값의 합"을 구하는 것은 monotone stack을 이용해 아래 두 배열을 구하면 $O(N)$에 해결할 수 있습니다.

- $Le[i] := i$보다 왼쪽에 있으면서 순서쌍 (lcp_i, i)보다 작은 원소가 나타나는 가장 오른쪽 인덱스

- $Ri[i] :=$ i보다 오른쪽에 있으면서 순서쌍 (lcp_i, i)보다 작은 원소가 나타나는 가장 왼쪽 인덱스

두 배열을 구했다면, (lcp_i, i)가 최소 원소인 구간은 총 $(i - Le[i])(Ri[i] - i)$개임을 알 수 있고, 따라서 "모든 구간의 최솟값의 합"은 $(i - Le[i])(Ri[i] - i) \times lcp_i$의 합과 같습니다.

```
ll N, Le[101010], Ri[101010], S;
vector<int> A, C;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; A.resize(N);
    for(auto &i : A) cin >> i;
    C = A;
    sort(C.begin(), C.end());
    C.erase(unique(C.begin(), C.end()), C.end());
    for(auto &i : A) i = lower_bound(C.begin(), C.end(), i) - C.begin();

    auto [sa,lcp] = SuffixArray(A, C.size());
    vector<pair<int,int>> V;
    for(int i=1; i<N; i++) V.emplace_back(lcp[i], i-1);

    stack<pair<int,int>> stk;
    for(int i=0; i<V.size(); i++){
        while(!stk.empty() && stk.top() > V[i]) stk.pop();
        Le[i] = !stk.empty() ? stk.top().second : -1;
        stk.push(V[i]);
    }

    while(!stk.empty()) stk.pop();
    for(int i=(int)V.size()-1; i>=0; i--){
        while(!stk.empty() && stk.top() > V[i]) stk.pop();
        Ri[i] = !stk.empty() ? stk.top().second : V.size();
        stk.push(V[i]);
    }

    for(int i=0; i<V.size(); i++) S += V[i].first * (i - Le[i]) * (Ri[i] - i);
    S = S * 2 + N * (N + 1) / 2;

    ll total = N * N, g = __gcd(S, total);
    cout << S / g << "/" << total / g;
}
```

D. Decrease the Boss Strength

시간 복잡도를 생각하지 않고, $O(NM)$ 정도 시간에 정답을 구하는 방법부터 생각해 봅시다.

$D(n) :=$ 현재 보스 체력이 n일 때 체력을 0으로 만드는 방법의 수라고 정의하면, $D(n) = \sum_{i;\ 2^{b_i}|n} D(n - a_i)$로 계산할 수 있습니다. $D(0) = 1$에서 시작해서 $D(N)$까지 계산해야 하는데, $N \le 10^{18}$이므로 행렬 등을 이용한 최적화를 생각해 볼 수 있습니다. 구체적으로, $D(x), D(x), \cdots, D(x-99)$의 값을 알고 있을 때 $D(x+2^k), D(x-1+2^k), \cdots, D(x-99+2^k)$처럼 인덱스를 2^k 만큼 점프시킬 수 있으면 문제를 빠르게 해결할 수 있습니다.

행렬을 이용해 선형 점화식을 계산하는 문제는 보통 매 순간 취할 수 있는 행동의 집합이 같지만, 이 문제는 그렇지 않아서 행렬을 구성하고 곱하는 것이 어렵습니다. 따라서 조금 더 제약 조건을 추가해서, $2^k|x$일 때 $D(x-1), D(x-2), \cdots, D(x-100)$을 $D(x), D(x-1), \cdots, D(x-99)$로 한 칸 전이하는 방법부터 찾아봅시다. 이제 보스의 체력이 2^k의 배수인 상황만 고려해도 됩니다.

현재 보스의 체력이 2^k의 배수라면 $b_i \le k$인 스킬을 사용할 수 있습니다. 따라서 아래와 같은 행렬을 사용하면 인덱스가 1씩 증가된 항의 값을 얻을 수 있습니다. 이때 $f_k(x)$는 $a_i = x, b_i \le k$인 스킬의 개수를 의미합니다.

$$\begin{pmatrix} f_k(1) & f_k(2) & f_k(3) & \cdots & f_k(99) & f_k(100) \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} D(x-1) \\ D(x-2) \\ D(x-3) \\ D(x-4) \\ \vdots \\ D(x-100) \end{pmatrix} = \begin{pmatrix} D(x) \\ D(x-1) \\ D(x-2) \\ D(x-3) \\ \vdots \\ D(x-99) \end{pmatrix}$$

좌변에서 앞에 있는 행렬을 $X[k]$라고 합시다. 인덱스에 1씩 더하면 더 이상 인덱스가 2^k의 배수가 아닐 수 있으므로 $X[k]$를 연달아서 여러 번 곱하는 것은 불가능합니다. 따라서 인덱스를 2^k씩 점프시키려면 다른 행렬을 또 만들어야 합니다.

보스의 체력이 $1, 2, \cdots, 8$일 때 사용할 수 있는 스킬의 목록을 보면, 차례대로 b_i가 $(0, 1, 0, 2, 0, 1, 0), 3$ 이하인 스킬을 사용할 수 있습니다. 또한, 체력의 범위를 16까지로 늘리면 b_i가 $(0, 1, 0, 2, 0, 1, 0), 3, (0, 1, 0, 2, 0, 1, 0), 4$ 이하인 스킬을 사용할 수 있습니다. 이런 식으로 규칙을 찾으면, $Y[k] = Y[k-1] \times X[k-1] \times Y[k-1]$라고 정의했을 때, $Z[k] = X[k] \times Y[k]$를 곱해서 2^k의 배수인 x를 $x + 2^k$으로 2^k만큼 증가시킬 수 있다는 것을 알 수 있습니다. 다시 말해, 아래와 같은 코드를 이용해 행렬을 N번 곱하는 대신, $Z[k]$를 $O(\log N)$번 곱해서 $D(n)$을 계산할 수 있습니다.

```
// naive
for(int i=1; i<=N; i++) res = X[__builtin_ctz(i)] * res;
// optimize
for(int i=B-1; i>=0; i--) if(N >> i & 1) res = Z[i] * res;
```

$A = 100, B = 60$이라고 하면 $X[0], X[1], \cdots, X[B]$은 $O(M + A^2B)$ 정도에 구할 수 있고, $Y[k], Z[k]$는 모두 $O(A^3B)$ 시간에 구할 수 있습니다. 실제 정답을 찾는 것은 N에서 켜진 비트의 위치에 따라 $Z[k]$를 열 벡터에 적절히 곱하면 되므로 $O(A^2 \log N)$ 시간에 처리할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll MOD = 1e9+7;
constexpr int A = 100, B = 61;

struct matrix{
    int n, m;
    vector<vector<ll>> a;
    matrix() : matrix(0, 0) {}
    matrix(int n, int m, int o=0) : n(n), m(m), a(n, vector<ll>(m, o)) {}
    void eye(){ for(int i=0; i<min(n,m); i++) a[i][i] = 1; }
    vector<ll>& operator [] (size_t idx) { return a[idx]; }
    const vector<ll>& operator [] (size_t idx) const { return a[idx]; }
};

matrix operator * (const matrix &a, const matrix &b){
    int n = a.n, m = a.m, k = b.m; matrix res(n, k);
    for(int i=0; i<n; i++) for(int j=0; j<m; j++) for(int t=0; t<k; t++) res[i][t] = (res[i][t] + a[i][j] * b[j][t]) % MOD;
    return res;
}

// Z[i] = \prod_{k=0}^{2^i-1} X[ctz(k)] = X[i] * Y[i]
// Z[0] = X0
```

```
// Z[1] = X1 X0
// Z[2] = X2 X0 X1 X0
// Z[3] = X3 X0 X1 X0 X2 X0 X1 X0

// Y[i] = Y[i-1] * X[i-1] * Y[i-1]
// Y[0] = id
// Y[1] = X0
// Y[2] = X0 X1 X0
// Y[3] = X0 X1 X0 X2 X0 X1 X0

ll N, M, C[B][A];
matrix X[B], Y[B], Z[B];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1,a,b; i<=M; i++) cin >> a >> b, C[b][a-1]++;

    for(int i=0; i<B; i++){
        X[i] = matrix(A, A);
        for(int j=0; j+1<A; j++) X[i][j+1][j] = 1;
        for(int j=0; j<=i; j++){
            for(int k=0; k<A; k++) X[i][0][k] += C[j][k];
        }
    }

    Y[0] = matrix(A, A); Y[0].eye();
    for(int i=1; i<B; i++) Y[i] = Y[i-1] * X[i-1] * Y[i-1];
    for(int i=0; i<B; i++) Z[i] = X[i] * Y[i];

    matrix res(A, 1); res[0][0] = 1;
    for(int i=B-1; i>=0; i--) if(N >> i & 1) res = Z[i] * res;
    cout << res[0][0];
}
```

E. Enigma of the Jewelry Case

배열을 90도씩 회전시켜 보면서 정렬되어 있는지 확인하면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> Rotate(vector<vector<int>> a){
    int n = a.size();
    vector<vector<int>> b(n, vector<int>(n));
    for(int i=0; i<n; i++) for(int j=0; j<n; j++) b[n-j-1][i] = a[i][j];
    return b;
}

bool Check(vector<vector<int>> a){
    int n = a.size();
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            if(i > 0 && a[i-1][j] > a[i][j]) return false;
            if(j > 0 && a[i][j-1] > a[i][j]) return false;
        }
    }
```

```
    return true;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; cin >> N;
    vector<vector<int>> A(N, vector<int>(N));
    for(auto &v : A) for(auto &i : v) cin >> i;
    for(int i=0; i<4; i++, A=Rotate(A)){
        if(Check(A)){ cout << i; return 0; }
    }
}
```

F. Fractions are better when continued

손으로 몇 번 계산하다 보면 피보나치 수열이 나온다는 것을 알 수 있습니다. 개인적으로는 예제 출력에 89 있는 것을 보고 추측하는 것이 가장 쉬운 방법이라고 생각합니다.

구체적으로는, $F(1) = 1; F(2) = 2; F(n) = F(n-1) + F(n-2)$라는 수열을 정의하면 $P_i = \frac{F(i)}{F(i+1)}$와 같이 나타낼 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll F[44] = {1, 1};

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    for(int i=2; i<44; i++) F[i] = F[i-1] + F[i-2];
    int n; cin >> n; cout << F[n];
}
```

G. Geography of Rivers

문제 지문이 틀렸습니다. 업데이트 쿼리에서의 tie-break 조건은 인덱스 최소가 아닌 **기존 값 유지**입니다.

이름이 같은 정점들은 "리프부터 부모 정점을 타고 올라가는 경로" 형태로 나열되어 있습니다. 따라서 트리를 총 N개의 **체인**으로 분할할 수 있고, 루트 정점이 속한 체인의 가장 아래에 있는 정점이 문제의 정답입니다. 리프 정점의 가중치가 증가할 때마다 체인의 변화를 효율적으로 추적하면 문제를 해결할 수 있습니다.

small to large의 원리를 생각해 보면, 리프에서 루트로 가는 경로는 최대 $O(\log \sum A_i)$개의 서로 다른 체인을 지난다는 것을 알 수 있습니다. 따라서 각 쿼리에서 정보가 변경되는 체인의 개수는 최대 $O(\log \sum A_i)$개이므로, 체인의 정보를 수정하는 작업을 $O(\log N)$ 시간에 처리할 수 있다면 전체 문제를 $O(N \log N + Q \log N \log \sum A_i)$ 시간에 해결할 수 있습니다.

HLD나 오일러 투어 트릭에 익숙하다면 어렵지 않게 구현할 수 있으므로 구체적인 설명은 생략합니다. HLD를 명시적으로 사용하지는 않습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 17;
```

```
namespace fenwick_tree{
    ll T[SZ];
    void add(int x, ll v){ for(x+=3; x<SZ; x+=x&-x) T[x] += v; }
    ll sum(int x){ ll r = 0; for(x+=3; x; x-=x&-x) r += T[x]; return r; }
    ll sum(int l, int r){ return l <= r ? sum(r) - sum(l-1) : 0; }
}

namespace segment_tree{ // {top of chain, leaf node}
    pair<int,int> T[SZ<<1];
    void set(int x, pair<int,int> v){
        for(T[x|=SZ]=v; x>>=1; ) T[x] = max(T[x<<1], T[x<<1|1]);
    }
    pair<int,int> get(int l, int r){
        pair<int,int> res;
        for(l|=SZ, r|=SZ; l<=r; l>>=1, r>>=1){
            if(l & 1) res = max(res, T[l++]);
            if(~r & 1) res = max(res, T[r--]);
        }
        return res;
    }
}

ll N, S, A[101010];
int In[202020], Out[202020], Par[202020], Top[101010];
vector<int> G[202020];

ll SubTree(int x){ return fenwick_tree::sum(In[x], Out[x]); }
void UpdateChain(int x, int top){
    Top[x] = top;
    segment_tree::set(In[x], {top, x});
}

int Init(int v){
    static int pv = 1;
    if(v <= N){
        In[v] = Out[v] = pv++;
        fenwick_tree::add(In[v], A[v]);
        UpdateChain(v, v);
        return v;
    }

    In[v] = pv;
    int l = G[v][0], r = G[v][1];
    Par[l] = Par[r] = v;
    int lc = Init(l), rc = Init(r);
    Out[v] = pv - 1;

    ll ls = SubTree(l), rs = SubTree(r);
    if(ls > rs || ls == rs && lc < rc){ UpdateChain(lc, v); return lc; }
    else{ UpdateChain(rc, v); return rc; }
}

int Update(int x){
    while(Top[x] != S){
        int p = Par[Top[x]];
        int l = G[p][0], r = G[p][1];
        if(l != Top[x]) swap(l, r); // (p, l) is light edge, (p, r) is heavy
edge
```

```
        ll ls = SubTree(l), rs = SubTree(r);
        auto [lt, lc] = segment_tree::get(In[l], Out[l]);
        auto [rt, rc] = segment_tree::get(In[r], Out[r]);
        if(ls > rs){
            UpdateChain(lc, max(lt, rt));
            UpdateChain(rc, r);
            x = lc;
        }
        else{
            UpdateChain(lc, l);
            UpdateChain(rc, max(lt, rt));
            x = rc;
        }
    }
    return x;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; S = N*2-1;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1,u,v; i<N; i++) cin >> u >> v, G[N+i] = {u, v};
    cout << Init(S) << "\n";

    int Q; cin >> Q;
    for(int q=1; q<=Q; q++){
        ll x, v; cin >> x >> v;
        fenwick_tree::add(In[x], v);
        cout << Update(x) << "\n";
    }
}
```

H. Harmonics with Interference

입력 전체에서 `*` 문자가 최대 $c \le 16$개만 주어지기 때문에, 가능한 2^c가지 경우를 모두 확인하면 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

bool Check(const string &a, const string &b){
    int mod = 0, res = 0;
    for(auto i : b) mod = mod * 2 + (i - '0');
    for(auto i : a) res = (res * 2 + i - '0') % mod;
    return res == 0;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    string s[2]; cin >> s[0] >> s[1];
    vector<pair<int,int>> v;
    for(int i=0; i<2; i++){
        for(int j=0; j<s[i].size(); j++) if(s[i][j] == '*') v.emplace_back(i,
j);
    }
    for(int bit=0; bit<(1<<v.size()); bit++){
```

```
        for(int i=0; i<v.size(); i++) s[v[i].first][v[i].second] = (bit >> i &
1) + '0';
        if(Check(s[0], s[1])){ cout << s[0]; return 0; }
    }
}
```

I. Ingredients that may Harm You

쿼리로 x가 주어졌을 때, 입력으로 주어진 N개의 수 $V_1, V_2, \cdots, V_N$와 서로소인 수의 개수를 c라고 하면 정답은 2^c입니다. c를 구하는 방법을 알아봅시다.

여사건을 생각해서, x와 서로소가 아닌 수의 개수를 구하는 방법을 생각해 봅시다. 만약 $x = 6$이라면 포함 배제의 원리를 이용해 (2의 배수 개수) + (3의 배수 개수) - (6의 배수 개수)로 계산할 수 있습니다. 또한, $x = 12 = 2^2 \times 3$이라고 하더라도, 6과 소인수의 집합이 같기 때문에 답은 변하지 않습니다.

이를 일반화하면, x가 주어졌을 때 x와 서로소가 아닌 수의 개수는 아래 방법을 이용해 구할 수 있습니다.

1. x의 소인수 $p_1, p_2, \cdots, p_k$를 구한다.
2. 소인수를 홀수 개 곱한 수의 배수는 더하고, 짝수 개 곱한 수의 배수는 뺀다.

x와 서로소인 수의 개수를 구하는 것은, 소인수를 홀수 개 곱한 수의 배수를 빼고, (0개를 포함해서) 소인수를 짝수 개 곱한 수의 배수를 더하면 됩니다.

이런 식의 포함 배제를 편하게 할 수 있도록 도와주는 함수로 뫼비우스 함수(Möbius function)가 있습니다. 뫼비우스 함수 $\mu : \mathbb{Z}^+ \to \{-1, 0, +1\}$은 다음과 같이 정의됩니다.

- n에 같은 소수가 두 번 이상 곱해졌다면(제곱 인수가 있다면) $\mu(n) = 0$
- 그렇지 않은 경우, n의 소인수 개수가 k일 때 $\mu(n) = (-1)^k$

즉, 입력으로 주어진 수 중에서 n의 배수가 $M(n)$개 있다면, 6과 서로소인 수의 개수는 $M(1) - M(2) - M(3) + M(6) = M(1)\mu(1) + M(2)\mu(2) + M(3)\mu(3) + M(6)\mu(6)$으로 계산할 수 있습니다. 이를 일반화하면 x와 서로소인 수의 개수는 x의 모든 약수 d에 대해 $M(d)\mu(d)$를 더한 것, 즉 $\sum_{d|x} M(d)\mu(d)$과 같습니다.

$M(1), M(2), \cdots, M(X)$는 $O(X \log X)$에 계산할 수 있고, $\mu(1), \mu(2), \cdots, \mu(X)$는 에라토스테네스의 체를 이용하면 $O(X \log \log X)$, 선형 체를 이용하면 $O(X)$에 계산할 수 있습니다. 모든 n에 대해 $\sum_{d|n} M(d)\mu(d)$를 계산하는 것도 $O(X \log X)$에 가능합니다.

따라서 모든 n에 대해 n과 서로소인 수의 개수를 $O(N + Q + X \log X)$에 구할 수 있습니다. (단, $X = 10^6$)

```
#include <bits/stdc++.h>
using namespace std;
constexpr int SZ = 1'000'000;
constexpr int MOD = 1e9+7;

// input, multiplier, coprime, pow
int N, Q, A[SZ+1], Mul[SZ+1], Co[SZ+1], Pw[SZ+1];

int C[SZ+1], Mu[SZ+1]; // mobius
vector<int> Primes;
void Sieve(int n=SZ){
    Mu[1] = 1;
    for(int i=2; i<=n; i++){
        if(!C[i]) Primes.push_back(i), Mu[i] = -1;
        for(auto j : Primes){
            if(i * j > n) break; C[i*j] = 1;
```

```
            if(i % j == 0){ Mu[i*j] = 0; break; }
            Mu[i*j] = Mu[i] * Mu[j];
        }
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1,t; i<=N; i++) cin >> t, A[t]++;
    for(int i=1; i<=SZ; i++) for(int j=i; j<=SZ; j+=i) Mul[i] += A[j];

    Sieve();
    for(int i=1; i<=SZ; i++){
        for(int j=i; j<=SZ; j+=i) Co[j] += Mu[i] * Mul[i];
    }

    cin >> Q; Pw[0] = 1;
    for(int i=1; i<=SZ; i++) Pw[i] = Pw[i-1] * 2 % MOD;
    for(int i=1,t; i<=Q; i++) cin >> t, cout << Pw[Co[t]] << "\n";
}
```

J. Journey through Colors

무향 그래프에 오일러 투어가 존재할 필요 충분 조건은 다음과 같습니다.

- 모든 정점이 서로 연결되어 있음
- 모든 정점의 차수가 짝수

오일러 투어가 존재하지 않으면 `-1`을 출력하고 종료하면 됩니다. 반대로 오일러 투어가 존재하는 경우, 일단 오일러 투어 $(v_1, e_1, v_2, e_2, v_3, \cdots)$를 아무거나 하나 구하고 생각합시다.

위에서 구한 오일러 투어는 색깔 조건을 무시하고 구한 것이기 때문에 올바른 해가 아닐 수도 있습니다. 간선 e의 색을 $c[e]$라고 정의합시다. 또한, v_i의 양옆에 있는 두 간선 e_i, e_{i+1}의 색이 같다면 i를 **"올바르지 않은 위치"**라고 부르도록 하겠습니다.

정답이 존재하지만 위에서 구한 오일러 투어는 올바른 해가 아닌 경우, 즉 올바르지 않은 위치가 1개 이상 있는 경우를 생각해 봅시다. v_i가 올바르지 않은 위치라면, $v_i = v_j$이면서 $c[e_i] \neq c[e_j], c[e_i] \neq c[e_{j+1}]$인 j가 항상 존재합니다. **(증명 안 함)** 따라서 v_i와 v_j 사이의 구간을 뒤집으면 올바르지 않은 위치의 개수를 1개 또는 2개 감소시킬 수 있습니다.

따라서 서로 같은 두 정점을 중심으로 하는 올바르지 않은 위치를 찾은 다음 구간을 뒤집는 것을 반복하면 문제를 해결할 수 있습니다. 올바르지 않은 위치는 최대 M개 존재하므로 구간을 뒤집는 연산도 최대 M번 수행합니다. 따라서 전체 시간 복잡도는 $O(M^2)$입니다.

```
#include <bits/stdc++.h>
using namespace std;
void Die(){ cout << -1; exit(0); }

int N, M, K, C[1010], D[1010], U[1010];
vector<pair<int,int>> G[1010];
vector<int> P;

void DFS(int v){
    while(true){
        while(!G[v].empty() && U[G[v].back().second]) G[v].pop_back();
        if(G[v].empty()) break;
```

```
        auto [x,i] = G[v].back(); G[v].pop_back();
        U[i] = 1; DFS(x); P.push_back(x); P.push_back(i);
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> K;
    for(int i=1; i<=M; i++){
        int u, v; cin >> u >> v >> C[i];
        G[u].emplace_back(v, i);
        G[v].emplace_back(u, i);
        D[u]++; D[v]++;
    }
    for(int i=1; i<=N; i++) if(D[i] % 2 != 0) Die();
    for(int i=1; i<=N; i++) if(D[i] != 0) { DFS(i); break; }
    if(P.size() != 2 * M) Die();

    auto prv = [](int x){ return x ? x-1 : 2*M-1; };
    auto nxt = [](int x){ return x+1 < 2*M ? x+1 : 0; };

    for(int i=0; i<P.size(); i+=2){
        bool flag = false;
        int x = P[i], e1 = P[prv(i)], e2 = P[nxt(i)];
        if(C[e1] != C[e2]) continue;
        for(int j=0; j<P.size(); j+=2){
            int y = P[j], e3 = P[prv(j)], e4 = P[nxt(j)];
            if(x != y || C[e1] == C[e3] || C[e1] == C[e4]) continue;
            reverse(P.begin()+min(i,j), P.begin()+max(i,j)+1);
            flag = true; break;
        }
        if(!flag) Die();
    }

    cout << P[0] << "\n";
    for(int i=1; i<P.size(); i+=2) cout << P[i] << " ";
}
```

K. Karamell

입력으로 주어진 N개의 수의 합을 S라고 하면, N개의 수를 합이 $S/2$인 두 그룹으로 나눈 다음 적당한 순서로 출력하면 문제를 해결할 수 있습니다.

합이 $S/2$인 그룹은 동적 계획법을 이용하면 어렵지 않게 구할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, S, A[111], D[111][10101], P[111][10101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    S = accumulate(A+1, A+N+1, 0);

    D[0][0] = 1;
```

```
    for(int i=1; i<=N; i++){
        for(int j=0; j<=S; j++) if(D[i-1][j]) D[i][j] = 1, P[i][j] = 0;
        for(int j=A[i]; j<=S; j++) if(D[i-1][j-A[i]]) D[i][j] = 1, P[i][j] = 1;
    }
    if(S % 2 != 0 || D[N][S/2] == 0){ cout << -1; return 0; }

    vector<int> X, Y;
    for(int i=N, j=S/2; i; i--){
        if(P[i][j]) X.push_back(A[i]), j -= A[i];
        else Y.push_back(A[i]);
    }

    int x = 0, y = 0;
    for(int i=1; i<=N; i++){
        if(x <= y) cout << X.back() << " ", x += X.back(), X.pop_back();
        else cout << Y.back() << " ", y += Y.back(), Y.pop_back();
    }
}
```

L. Lecographically Maximum

비트마다 교환할 수 있으므로, 각 자리에 켜져 있는 비트를 모두 앞으로 몰아주면 사전 순으로 최대인 수열을 만들 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, C[33];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++){
        int t; cin >> t;
        for(int j=0; j<30; j++) if(t >> j & 1) C[j]++;
    }
    for(int i=1; i<=N; i++){
        int now = 0;
        for(int j=0; j<30; j++) if(C[j]) now |= 1 << j, C[j]--;
        cout << now << " ";
    }
}
```