Dicas

Na prova

Olhar os balões pelo score do site para não confundir e considerar o tempo que as questões foram feitas, além do número de questões feitas.

Boilerplate

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

using namespace std;

void setup() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
}

const int INF = 1e9 + 7;
const ll LINF = 1e18 + 7;
const double PI = acos(-1.0);
const double EPS = 1e-9;

int main() {
    setup();
    return 0;
}
```

```cpp
// Para imprimir um double 'd' com 9 casas decimais
double d = 0.31415926535;
cout << fixed << setprecision(9) << d << endl;
// Saída: 0.314159265
```

Técnicas de Resolução de Problemas

Análise de Restrições

O programa demora ~1 segundo para ~ 4 × 10 **8 operações. Com isso, prestar atenção nos limites de tempos das questões para não dar TLE (Time Limit Exceeded).

- **$N \le 20 \implies O(2^N \cdot N)$** (Exponencial, DP com bitmask, backtracking)
- **$N \le 500 \implies O(N^3)$** (Floyd-Warshall, DP cúbico)
- **$N \le 5000 \implies O(N^2)$** (DP quadrático, grafos densos)
- **$N \le 10^5 \implies O(N \log N)$** (Ordenação, estruturas de dados baseadas em árvore)
- **$N \le 10^7 \implies O(N)$** (Processamento linear, Two Pointers)

- **$N > 10^7 \implies O(\log N)$ ou $O(1)$** (Busca Binária, fórmulas matemáticas)

Técnicas

- **Brute Force:** Considere a solução mais simples.
- **Busca Binária na Resposta:** Se você pode verificar se uma resposta X é válida, e se X for válida, qualquer valor "menor" (ou "maior") que X também for, use busca binária para encontrar o melhor X. A função de checagem vira `F F F...T T T...`.
- **Greedy:** O que acontece se processarmos os itens em ordem crescente/decrescente de algum valor? Muitas vezes, isso revela uma estrutura ótima.
- **Programação Dinâmica:** A solução para o problema de tamanho N depende das soluções para problemas de tamanho menor.
- **Grafos:** Modele o problema como nós e arestas.
- **SegTree/BIT:** Problemas de queries em ranges.
- **Two Pointers/Sliding Window:** Ótimo para encontrar sub-arrays/substrings que satisfazem uma condição.
- **Sets, Maps, Priority Queue, Multiset**: add in structure while in order
- **Prefix**: arr[i]++, arr[j]--, arr[i] += arr[i-1]

Passar vetor por referência

Exemplo:

Faça:

```
int get_size(vector<int> &a) {
  return (int)a.size();
}
```

em vez de:

```
int get_size(vector<int> a) {
  return (int)a.size();
}
```

Pois passando o vetor no segundo caso requer $O(n)$ O(n) (vai fazer uma cópia do vetor) e pode dar erro de memória se o vetor for muito grande e estivermos fazendo várias chamadas recursivas, enquanto passando por referência (primeiro caso) requer $O(1)$ O(1).

map e unordered_map

Podemos iterar pelo map (ordenado) e unordered_map (não ordenado) com um iterador ou:

```
for (auto &it: map) (pair<>)
```

Porém, tome cuidado, pois ao checar um elemento i que não existe no map ainda:

```
if (mp[i])
```

o elemento i é adicionado no map e nesse caso também passaríamos por ele. Para fazer isso sem adicionar o elemento i no map podemos fazer:

```
if (mp.find(i) != mp.end()) {
  // código
}
```

Tome cuidado com o uso de unordered_map, pois o número de colisões pode fazer com que o lookup fique O(n), enquanto map O(log n)

ordered_set / unordered_set

Estrutura	Inserção/Deleção
set	O(log n)
unordered_set	O(1) (em média)

lower_bound / upper_bound

lower_bound: retorna um iterador para o menor elemento maior ou igual a algum elemento k. $O(\log n)$

upper_bound: retorna um iterador para o menor elemento estritamente maior que algum elemento k. $O(\log n)$

Funciona para set, map, vetor.

Exemplo:

```
  1 1 1 1 1 1 2
  ^           ^   para 1
  lb          ub
```

```
vector<int> v = {10, 20, 20, 20, 30, 40};

// Encontra o primeiro elemento >= 20
auto it_lb = lower_bound(v.begin(), v.end(), 20);
// it_lb aponta para o elemento no índice 1 (valor 20)
int idx_lb = it_lb - v.begin(); // idx_lb = 1

// Encontra o primeiro elemento > 20
auto it_ub = upper_bound(v.begin(), v.end(), 20);
```

```
// it_ub aponta para o elemento no índice 4 (valor 30)
int idx_ub = it_ub - v.begin(); // idx_ub = 4

// Se o elemento não existe, aponta para onde ele seria inserido
it_lb = lower_bound(v.begin(), v.end(), 25);
// it_lb aponta para o elemento no índice 4 (valor 30)
```

```
set<int> s = {10, 20, 30, 40};

// Encontra o primeiro elemento >= 25
auto it_lb = s.lower_bound(25);
// it_lb aponta para o elemento 30
if (it_lb != s.end()) {
    cout << "Lower bound de 25 é " << *it_lb << endl; // Saída: 30
}

// Encontra o primeiro elemento > 40
auto it_ub = s.upper_bound(40);
if (it_ub == s.end()) {
    cout << "Não há elementos maiores que 40" << endl;
}
```

Otimizadores IO

```
ios::sync_with_stdio(false);
cin.tie(NULL);
```

Ao usá-los, não misture estilos de print do C e C++ (não utilize cout e printf no mesmo código nesse caso).

statisticsTree (sweepline)

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<class T> using ordered_set = tree<T, null_type, less<T>,
rb_tree_tag, tree_order_statistics_node_update>;

// Uso:
// ordered_set<int> s;
// s.insert(x);
// s.find_by_order(k); // Retorna iterador para o k-ésimo elemento (0-
indexed)
// s.order_of_key(x);  // Retorna o número de elementos estritamente
menores que x
```

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// note que está ordenando do menor para o maior e só tem 1 elemento de
cada
// para mais de um elemento coloque pair<int, int> ou algo assim
typedef tree<
    int,
    null_type,
    greater<int>,
    rb_tree_tag,
    tree_order_statistics_node_update>
statisticsTree;
```

Uso:

```
auto f = [&](vector<pair<int, int>> lines) {
    sort(lines.begin(), lines.end());
    statisticsTree ord;
    for (auto &p: lines) {
        // verifica qual posição p.second está
        max_slices += ord.order_of_key(p.second);
        ord.insert(p.second);
    }
    return;
};
```

Segment Tree

Útil para queries em range.

Exemplo:

```
template <class T> class MinSegmentTree {
private:
    const T DEFAULT = numeric_limits<T>().max();
    vector<T> segTree;
    int len;

public:
    MinSegmentTree (int segTree_len) {
        len = segTree_len;
        segTree = vector<T>(len * 2, DEFAULT);
    }

    void set(int idx, T val) {
        idx += len;
```

```
        segTree[idx] = val;

        // update (ignoring 0)
        for (; idx > 1; idx >>= 1) {
            segTree[idx >> 1] = min(segTree[idx], segTree[idx ^ 1]);
        }
    }

    T range_min(int l, int r) {
        T min_ = DEFAULT;

        for (l += len, r += len; l < r; l >>= 1, r >>= 1) {
            if (l & 1) min_ = min(min_, segTree[l++]);
            if (r & 1) min_ = min(min_, segTree[--r]);
        }
        return min_;
    }
};
```

```
template <class T> class SumSegmentTree {

  private:

    const T DEFAULT = 0;
    int len;
    vector<T> segtree;

    T combine(const T &a, const T &b) { return a + b; }


    // [at left, at right]
    // by at*2 we go to the left, so the new range is
    // [at left, mid]

    // by at*2+1 we go to the right, so the new range is
    // [mid+1, at right]
    void build(const vector<T> &arr, int at, int at_left, int at_right) {

        if (at_left == at_right) {
            segtree[at] = arr[at_left];
            return;
        }

        int mid = (at_left + at_right) / 2;

        build(arr, 2 * at, at_left, mid);
        build(arr, 2 * at + 1, mid + 1, at_right);

        segtree[at] = combine(segtree[2 * at], segtree[2 * at + 1]);
    }
```

```
    void set(int ind, T val, int at, int at_left, int at_right) {

        if (at_left == at_right) {
            segtree[at] = val;
            return;
        }

        int mid = (at_left + at_right) / 2;

        if (ind <= mid) {
            set(ind, val, 2 * at, at_left, mid);
        } else {
            set(ind, val, 2 * at + 1, mid + 1, at_right);
        }

        segtree[at] = combine(segtree[2 * at], segtree[2 * at + 1]);
    }

    T get(int ind, int at, int at_left, int at_right) {

        if (at_left == at_right) {
            return segtree[at];
        }

        int mid = at_left + (at_right - at_left) / 2;

        if (ind <= mid) {
            return get(ind, 2 * at, at_left, mid);
        } else {
            return get(ind, 2 * at + 1, mid + 1, at_right);
        }
    }


    T range_sum(int start, int end, int at, int at_left, int at_right) {
        if (at_right < start || end < at_left) { return DEFAULT; }
        if (start <= at_left && at_right <= end) { return segtree[at]; }

        int mid = (at_left + at_right) / 2;

        T left_res = range_sum(start, end, 2 * at, at_left, mid);
        T right_res = range_sum(start, end, 2 * at + 1, mid + 1, at_right);

        return combine(left_res, right_res);
    }


  public:
    SumSegmentTree(int len) : len(len) { segtree = vector<T>(len * 4,
DEFAULT); };

    SumSegmentTree(const vector<T> &arr) : len(arr.size()) {
        segtree = vector<T>(len * 4, DEFAULT);
```

```
        build(arr, 1, 0, len - 1);
    }


    void set(int ind, T val) {
        // at 1 since we start at the root
        set(ind, val, 1, 0, len - 1);
    }

    T get(int ind) {
        return get(ind, 1, 0, len-1);
    }

    T range_sum(int start, int end) { return range_sum(start, end, 1, 0,
len - 1); }
};
```

Fenwick Tree (BIT)

Para queries de soma em prefixo e updates em um ponto ($O(\log N)$).

```
template <class T> class FenwickTree {
private:
    vector<T> bit;
    int size;
public:
    FenwickTree(int sz) : size(sz), bit(sz + 1, 0) {}

    void update(int idx, T delta) {
        for (++idx; idx <= size; idx += idx & -idx) bit[idx] += delta;
    }

    T query(int idx) {
        T sum = 0;
        for (++idx; idx > 0; idx -= idx & -idx) sum += bit[idx];
        return sum;
    }

    T query_range(int l, int r) {
        return query(r) - query(l - 1);
    }
};
```

Bitset

Para utilizar operações de bits em uma string de bits com mais de 64 bits podemos usar o bitset. Ele suporta as bitwise operations e podem ser usados como vetor (por exemplo, utilizar [] para acessar, além de ser possível passar para string)

```
bitset<size> variable_name;
```

Grafos

- **Representação:** Lista de adjacência `vector<pii> adj[N];`
- **BFS (Busca em Largura):** Caminho mínimo em grafos com peso 1.
- **DFS (Busca em Profundidade):** Conectividade, ciclos, ordenação topológica.
- **Dijkstra:** Caminho mínimo com pesos não-negativos. Use com `priority_queue`.
- **Kruskal:** Árvore Geradora Mínima (MST). Use com DSU.

Dijkstra (Caminho Mínimo com Pesos Não-Negativos)

Usa uma `priority_queue` para encontrar o caminho mais curto de um nó de origem para todos os outros.

- **Complexidade:** $O(E \log V)$
- **Representação:** Lista de adjacência `vector<pii> adj[N];` onde `pii` é `{peso, vizinho}`.

```
vector<ll> dijkstra(int start, int n, const vector<vector<pii>>& adj) {
    vector<ll> dist(n + 1, LINF);
    dist[start] = 0;

    // Fila de prioridade: {distância, nó} - min-heap
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<pair<ll,
int>>> pq;
    pq.push({0, start});

    while (!pq.empty()) {
        ll d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // Otimização: se já achamos um caminho melhor, ignora
        if (d > dist[u]) {
            continue;
        }

        for (auto& edge : adj[u]) {
            int v = edge.second;
            int weight = edge.first;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Kruskal (Árvore Geradora Mínima - MST)

Encontra um subconjunto de arestas que conecta todos os vértices com o menor custo total possível.

- **Complexidade:** $O(E \log E)$
- **Dependência:** Estrutura DSU (Union-Find).
- **Representação:** Vetor de arestas `{peso, u, v}`.

```
struct DSU {
    vi parent;
    DSU(int n) {
        parent.resize(n + 1);
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int i) {
        if (parent[i] == i) return i;
        return parent[i] = find(parent[i]); // Path compression
    }
    void unite(int i, int j) {
        int root_i = find(i);
        int root_j = find(j);
        if (root_i != root_j) {
            parent[root_i] = root_j;
        }
    }
};

// Arestas do grafo
vector<tuple<int, int, int>> edges; // {peso, u, v}

ll kruskal(int n) {
    // 1. Ordena as arestas pelo peso
    sort(edges.begin(), edges.end());

    // 2. Inicializa DSU
    DSU dsu(n);

    ll mst_cost = 0;
    int edges_count = 0;

    for (const auto& edge : edges) {
        auto [weight, u, v] = edge;

        // 3. Se u e v não estão no mesmo componente
        if (dsu.find(u) != dsu.find(v)) {
            mst_cost += weight;
            dsu.unite(u, v);
            edges_count++;
            // Otimização: para quando a MST estiver completa
            if (edges_count == n - 1) break;
        }
    }
```

```
    // Opcional: checar se o grafo é conexo
    if (edges_count < n - 1) return -1; // Não foi possível formar a MST

    return mst_cost;
}
```

Topological sort

```
// n = número de vértices
// adj = lista de adjacência do grafo direcionado
vector<int> topological_sort(int n, const vector<vector<int>>& adj) {
    vector<int> in_degree(n + 1, 0);
    for (int u = 1; u <= n; ++u) {
        for (int v : adj[u]) {
            in_degree[v]++;
        }
    }

    queue<int> q;
    for (int i = 1; i <= n; ++i) {
        if (in_degree[i] == 0) {
            q.push(i);
        }
    }

    vector<int> result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        for (int v : adj[u]) {
            in_degree[v]--;
            if (in_degree[v] == 0) {
                q.push(v);
            }
        }
    }

    // Se result.size() < n, o grafo tem um ciclo
    if (result.size() < n) {
        return {}; // Retorna vetor vazio para indicar ciclo
    }

    return result;
}
```

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

bool dfs(int pupil, vector<int> &colors, vector<vector<int>> &graph) {
    // cant be the same color as the pupil
    int friend_color = colors[pupil] == 1 ? 2 : 1;

    for (int &f: graph[pupil]) {
        if (colors[f] == 0) {
            // just change it
            colors[f] = friend_color;
            if (!dfs(f, colors, graph)) {
                return false;
            }
        } else if (colors[f] != friend_color) {
            // can't have
            return false;
        }
    }
    return true;
}

int main() {
    int n, friendships;
    cin >> n >> friendships;

    vector<int> colors(n);
    vector<vector<int>> graph(n);
    for (int i = 0; i < friendships; i++) {
        int f1, f2;
        cin >> f1 >> f2;
        graph[f1 - 1].push_back(f2 - 1);
        graph[f2 - 1].push_back(f1 - 1);
    }

    for (int i = 0; i < n; i++) {
        if (colors[i] == 0) {
            // put in the group 1
            colors[i] = 1;
            if (!dfs(i, colors, graph)) {
                cout << "IMPOSSIBLE\n";
                return 0;
            }
        }
    }

    for (int c: colors) {
        cout << c << " ";
    }
    cout << "\n";
}
```

Matemática

- **GCD e LCM**

```
ll gcd(ll a, ll b) { return b == 0 ? a : gcd(b, a % b); }
ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b; }
```

- **Exponenciação Modular** Calcula $(base^{exp}) \pmod M$ em $O(\log exp)$.

```
ll binpow(ll base, ll exp, ll mod) {
    ll res = 1;
    base %= mod;
    while (exp > 0) {
        if (exp % 2 == 1) res = (res * base) % mod;
        base = (base * base) % mod;
        exp /= 2;
    }
    return res;
}
```

- **Crivo de Eratóstenes (Sieve)** Encontra todos os primos até N em $O(N \log \log N)$.

```
vector<bool> is_prime(N + 1, true);
is_prime[0] = is_prime[1] = false;
for (int p = 2; p * p <= N; p++) {
    if (is_prime[p]) {
        for (int i = p * p; i <= N; i += p)
            is_prime[i] = false;
    }
}
```

Strings

- **Hashing (Rolling Hash):** Técnica para comparar substrings em $O(1)$ após um pré-processamento $O(N)$. Útil para encontrar todas as ocorrências de um padrão, palíndromos, etc.

Monotonic Increasing/Decreasing Stack

Um stack que mantém os números ordenados, enquanto percorre pela arr

Binary search

Try to make an array of with a monotonic function: [T, T, T, ..., T, F, F, ..., F] (get the last true)

```
int last_true(int lo, int hi, function<bool(int)> f) {
    // if none of the values in the range work, return lo - 1
```

```
    lo--;
    while (lo < hi) {
        // find the middle of the current range (rounding up)
        int mid = lo + (hi - lo + 1) / 2;
        if (f(mid)) {
            // if mid works, then all numbers smaller than mid also work
            lo = mid;
        } else {
            // if mid does not work, greater values would not work either
            hi = mid - 1;
        }
    }
    return lo;
}
```

[F, F, F, ..., F, T, T, ..., T] (get the first true)

```
ll first_true(ll lo, ll hi, function<bool(ll)> f) {
    hi++;

    while (lo < hi) {
        ll mid = lo + (hi - lo) / 2;

        if (f(mid)) {
            hi = mid;
        } else {
            lo = mid +1;
        }
    }
    return lo;
}

double first_true(double lo, double hi, function<bool(double)> f) {

    while (hi - lo > 1e-6) {
        double mid = lo + (hi - lo) / 2;

        if (f(mid)) {
            hi = mid;
        } else {
            lo = mid;
        }
    }
    return lo;
}
```

Programação Dinâmica

- **Estado da DP:** Defina `dp[i]...` como "a melhor resposta para o subproblema considerando os primeiros `i` elementos".

- **Recorrência:** Encontre a relação entre `dp[i]` e os estados anteriores (`dp[i-1]`, `dp[i-2]`, etc.).
- **Padrões Clássicos:**
  - **Knapsack (Mochila):** `dp[i][w]` = maior valor usando os primeiros `i` itens com capacidade `w`.
  - **LIS (Longest Increasing Subsequence):** `dp[i]` = tamanho da LIS que termina no elemento `i`. Pode ser otimizado para $O(N \log N)$.
  - **LCS (Longest Common Subsequence):** `dp[i][j]` = tamanho da LCS entre as strings `s1[0..i]` e `s2[0..j]`.
  - **DP em Grids/Matrizes:** `dp[i][j]` = resposta para o sub-grid terminando em `(i, j)`.
  - **DP com Bitmask:** Use uma máscara de bits para representar subconjuntos de itens como estado. `dp[mask]` = resposta para o subconjunto de itens representados por `mask`.
  - **Intervalo:** `dp[i][j]` de i até j, considere todos os j para esse i, etc.

```
// Coin example
// many of same coin
    vector<int> dp(target + 1, 0);
    dp[0] = 1;
    for (int i = 1; i <= target; i++) {
        for (int c: coins) {
            if (i - c >= 0) {
                dp[i] += dp[i - c];
                dp[i] %= (int) (1e9 + 7);
            }
        }
    }

// only one use coin
    vector<int> dp(target + 1, 0);
    dp[0] = 1;
    // first add every coin of one type to dont make duplicates
    // note that we not adding in a ordered way, but it works the same way
    // because we want only the number of ways
    for (int &curr_coin: coins) {
        for (int i = 1; i <= target; i++) {
            if (i - curr_coin >= 0) {
                dp[i] += dp[i - curr_coin];
                dp[i] %= mod;
            }
        }
    }
    cout << dp[target] << endl;
```