Aula 15 · Jogos Combinatórios Imparciais, Decomposição em Raiz Quadrada e Algoritmo de Mo

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

28/06/2023

Jogos Combinatórios Imparciais

Jogo imparcial

- Jogo finito.
- Jogo em turnos alternantes entre dois jogadores.
  - Jogo dos quadradinhos não é alternante: Se você fizer um quadrado você pode repetir sua jogada.
- Ações simétricas (a mesma pessoa ia ter as mesmas jogadas).
  - No Xadrez, um jogador só pode mexer as peças brancas e o outro só pode mexer as peças pretas.

Esvazie a pilha

Considere um jogo imparcial onde

- Existe uma pilha com N pedras.
- Os jogadores alternam jogadas entre si.
- Em cada jogada, o jogador da vez escolhe tirar 1, 2 ou 3 pedras da pilha (desde que a pilha tenha um número suficiente de pedras)
- Perde o jogador que não puder tirar pedra nenhuma (isto é, ganha aquele que tirar a última pedra)

Posições vencedoras e posições perdedoras

- Posição perdedora só pode mudar de estado para posição vencedora.
- Posição vencedora pode mudar o estado para alguma posição perdedora.
- No jogo anterior, posições com $X \pmod{3+1} = 0$ pedras são perdedoras e todas as outras são vencedoras.

O jogo de Nim

Consideremos agora o seguinte jogo:

- Há N pilhas de pedras de tamanho variável.
- Dois jogadores alternam jogadas entre si.
- Em cada jogada, o jogador da vez escolhe uma das pilhas e remove dela quantas pedras (no mínimo uma) quiser (desde que a pilha tenha o número de pedras a ser removido).
- Perde quem não tiver jogadas (ganha quem esvaziar a última pilha)

Chamamos este jogo de Jogo de Nim.

Olhando para as pilhas de Nim

Quais são as posições perdedoras e vencedoras no jogo de Nim?

- Posição perdedora: Xor dos tamanhos de pilhas vale 0
- Posição vencedora: Qualquer outra

Observação importante: todos os jogos são independentes (mudar uma pilha não muda nada no estado das outras).

Provando posições perdedoras

Posição é perdedora se e somente se o xor das pilhas é 0.

- Vamos provar que de uma posição perdedora $g = 0$, só podemos ir para uma posição vencedora $g' \neq 0$.
- Se o xor de todas as pilhas é 0,

$$g = p_1 \oplus p_2 \oplus p_3 \oplus \ldots \oplus p_n = 0$$
$$p_2 \oplus p_3 \oplus \ldots \oplus p_n = p_1$$

- Sem perda de generalidade, podemos pegar uma das pilhas, por exemplo a pilha p_1 e reduzir a unidades ($1 \leq a \leq p_1$).

$$\begin{aligned} g' &= (p_1 - a) \oplus p_2 \oplus p_3 \oplus \ldots \oplus p_n \\ &= (p_1 - a) \oplus p_1 \end{aligned}$$

- Como em xor o único inverso de p_1 é p_1 e $p_1 \neq (p_1 - a)$, concluímos que $g' \neq 0$.

Provando posições vencedoras

Agora temos que provar que é possível ir de uma posição onde $g \neq 0$ para uma posição onde $g' = 0$.

- Seja p a posição do bit mais significativo em g.
- Existe um número ímpar de pilhas onde o bit p está ligado, escolhemos uma pilha u para jogar.
- Nossa jogada será $g' = g \oplus u \oplus t = 0$. Isto porque u é o inverso de u em xor, e t será o nosso novo u.
- Então $t = g \oplus u$, para cumprir a equação.
- Sabemos que $t < u$ pois vamos alterar apenas os bits menores ou iguais a p de u.

Então é possível partir de qualquer posição com soma xor $g \neq 0$ para uma com soma xor $g = 0$ (ou seja, ir de uma posição vencedora para uma perdedora).

Resolvendo o Nim

Código `nim.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; while (cin >> n) {
        int g = 0;
        vector<int> s (n);
        for (int i = 0; i < n; i++) {
            cin >> s[i]; g ^= s[i]; }
        if (g == 0) { cout << "-1\n"; continue; }
        int msb = (1<<__lg(g));
        for (int i = 0; i < n; i++) if (s[i]&msb) {
            int t = s[i]^g;
            cout << s[i]-t << " " << i+1 << "\n";
            break;
        }
    }
}
```

Exemplo de um jogo de Nim

Entrada	Saída
3 3 4 5	2 1
3 1 4 2	1 2
3 1 2 2	1 1
3 0 1 2	1 3
3 0 0 1	1 3
3 0 0 0	-1

Pilha 1	Pilha 2	Pilha 3	Movimento
3	4	5	Início do jogo
1 (-2)	4	5	Computador tira 2 da pilha 1
1	4	2 (-3)	Humano tira 3 da pilha 3
1	3 (-1)	2	Computador tira 1 da pilha 2
1	2 (-1)	2	Humano tira 1 da pilha 2
0 (-1)	2	2	Computador tira 1 da pilha 1
0	1 (-1)	2	Humano tira 1 da pilha 2
0	1	1 (-1)	Computador tira 1 da pilha 3
0	0 (-1)	1	Humano tira 1 da pilha 2
0	0	0 (-1)	Computador tira 1 da pilha 3 e ganha

Nim k-Slow

- Jogador escolhe uma pilha e tira $0 < P \leq K$ elementos dela.
- Posição é a soma xor dos tamanhos das pilhas $\pmod{k+1}$:

$$(a_1 \mod (k+1)) \oplus (a_2 \mod (k+1)) \oplus \ldots \oplus (a_n \mod (k+1))$$

Atenção que $(a_1 \oplus a_2 \oplus \ldots \oplus a_n) \pmod{k+1}$ **não é igual**.

$$\begin{aligned}
(6 \mod 5) \oplus (1 \mod 5) &= 1 \oplus 1 = 0 \\
(6 \oplus 1) &= 7 = 2 \pmod 5 \\
(6 \mod 7) \oplus (19 \mod 7) &= 6 \oplus 5 = 3 \\
(6 \oplus 19) &= 21 = 0 \pmod 7 \\
(6 \mod 5) \oplus (4 \mod 5) &= 1 \oplus 4 = 5 \\
(6 \oplus 4) &= 2 \pmod 5
\end{aligned}$$

Para alguns módulos específicos, o módulo pode ser comutativo com o xor. Para módulo $m = 2^n$, o módulo é equivalente a aplicação de máscara `& (m-1)`, que é comutativa com xor.

Nim monotônico

- Estado só é válido se a i-ésima pilha for menor ou tão alta quanto a $i+1$-ésima (sequência não decrescente).
- Fora isso podemos tirar um número arbitrário de pedras de uma pilha escolhida como no jogo original.
- Para um estado $A = (a_1, a_2, \ldots, a_{2k})$ (número par de pilhas), mapeamos A para um estado $B = (b_1, b_2, \ldots, b_k)$ onde $b_i = a_{2i} - a_{2i-1}$ (estado de diferença).
- Caso o estado A seja ímpar, colocamos mais uma posição 0 no seu início para torná-lo par.
- O estado A é uma posição perdedora no Nim monotônico se, e somente se, B é uma posição perdedora no Nim incrementado

Nim incrementado

- Agora vamos permitir adicionar pedras à pilha, e não só removê-las.
- É importante manter o jogo acíclico (finito), e portanto as regras (independente de quais sejam) devem garantir que os incrementos respeitem isso.
- Isso não muda a definição de posições vencedoras e perdedoras porque na estratégia vencedora é inútil adicionar pedras (se o primeiro jogador adiciona X pedras a uma pilha, o segundo pode remover X pedras da mesma pilha, desfazendo a jogada).

Função mex

- Mínimo excludente.
- Primeiro número natural que não pertence ao conjunto.

$$\text{mex}\{0, 1, 3, 5\} = 2.$$
$$\text{mex}\{1, 3, 5\} = 0.$$

Teorema de Sprague-Grundy

- Vamos encontrar para qualquer estado de um jogo imparcial um estado correspondente em um jogo de Nim incrementado.
- Para um estado e_i a partir do qual podem ser acessados os estados num conjunto E_i, atribuímos a e_i um jogo de Nim com pilha de tamanho X, onde X é o **número de Grundy** ou **nímero** de e_i ($G(e_i)$):

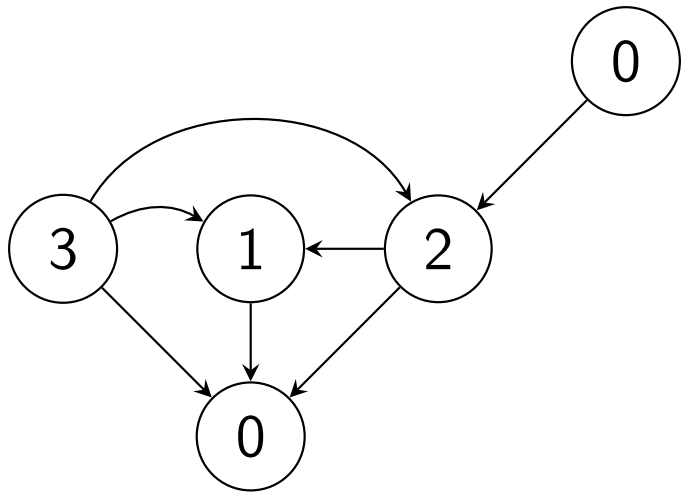
$$G(e_i) = \text{mex}(E_i)$$

- O nímero de um estado perdedor é 0, e o nímero de um estado vencedor é positivo.
- Por conta do uso da função **mex** que se fez necessário o uso do Nim incrementado.

Número de Grundy com subjogos

- Consideremos um jogo que consista de subjogos: a cada turno, o jogador da vez escolhe um subjogo e realiza uma jogada nele. O jogo acaba quando não há movimentos possíveis.
- Neste caso, o número de Grundy de um estado é o xor dos níneros dos subjogos.

DAG gerado de estados



Implementação do mex usando set

Código `mexset.h`

```
struct mex {
    set<int> tomex;

    void insert_mex(int x) {
        tomex.insert(x);
    }
    int get_reset_mex() {
        int m = 0;
        for (auto &j : tomex) {
            if (j > m) { tomex.clear(); return m; }
            m++;
        }
        tomex.clear();
        return m;
    }
};
```

Implementação do mex usando um vetor

Código `mex.h`

```
struct mex {
    vector<bool> tomex;
    int gt = 0;

    mex(int n) : tomex(n) {};

    void insert_mex(int x) {
        tomex[x] = 1;
        gt = max(gt, x);
    }
    int get_reset_mex() {
        int m;
        for (m = 0; tomex[m]; m++);
        fill(tomex.begin(), tomex.begin()+gt+1, 0);
        return m;
    }
};
```

O jogo de Grundy

- Existe uma pilha de n pedras e dois jogadores se movem alternadamente.
- Em cada jogada, um jogador escolhe uma pilha e divide ela em duas pilhas não vazias que tem um número diferente de moedas.
- O jogador que fizer a última jogada ganha o jogo.
- Esse jogo é um pouquinho mais complexo, e precisa que a gente use números de Grundy para resolver.

Implementação do jogo de Grundy

Código grundy.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6+15;
#include "mexset.h"
vector<int> g (N, -1);
int main() {
    mex m (N);
    for (int n = 0; n <= 1226; n++) {
        if (n == 1 || n == 2) { g[n] = 0; continue; }
        for (int i = 1; i <= n/2; i++) if (i != n-i)
            m.insert_mex(g[i] ^ g[n-i]);
        g[n] = m.get_reset_mex();
    }
    int t; cin >> t; while (t--) {
        int n; cin >> n;
        if (n >= 1226)
            cout << "first\n";
        else
            cout << (g[n] != 0 ? "first" : "second") << "\n";
    }
}
```

Implementação do jogo de Grundy (continuado)

Entrada	Saída
3 6 7 8	first second first

Um exercício em procura de padrões

- Por que quando $N \geq 1226$, o primeiro jogador sempre ganha?
- Não sei, mas isso é verdade até pelo menos 10^6.
- É uma conjectura em aberto se isso é verdade sempre.
- Muitos jogos imparciais em geral chegam a padrões que se repetem, então as vezes tem como roubar com uma solução matemática (como já vimos).

Misère

- Quando o número de Grundy é zero, você está no estado de derrota.
- Tem alguns outros jogos onde isso é o contrário, o estado de zero é vitória. Isso é chamado de *misère*.
- Em alguns casos é só inverter o jogo, em outros, não é tão trivial.
- No Nim normal a estratégia é a mesma com apenas uma situação onde se muda a jogada: Quando a jogada iria deixar apenas pilhas de tamanho 1. Nesse caso, a estratégia certa é deixar um número ímpar de pilhas de tamanho 1.

Implementação do Misère

Código misere.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    int t; cin >> t;
    while (t--) {
        int n; cin >> n;
        bool all_ones = true;
        int res = 0;
        for (int i = 0; i < n; i++) {
            int x; cin >> x; res ^= x;
            if (x != 1) { all_ones = false; }
        }
        if (all_ones) { res = n % 2 == 0; }
        cout << (res != 0 ? "first" : "second") << "\n";
    }
}
```

Implementação do Misère (continuado)

Entrada	Saída
4 2 1 1 3 2 1 3 3 1 1 1 4 1 1 1 1	first second second first

Nim de escada

- Temos um vetor com N números.
- Podemos mover X de uma posição do vetor da direita para esquerda (soma X na esquerda, diminui X na direita).
- Como modelar isso como um jogo de Nim?

Nim de escada

- Podemos desconsiderar as posições ímpares porque o segundo jogador pode sempre "desfazer" a sua jogada.
- Então precisamos apenas das posição pares (que são as válidas).

Implementação do Nim de escada

Código `stair.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int t; cin >> t;
    while (t--) {
        int n; cin >> n;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int p; cin >> p;
            if (i % 2) { ans ^= p; }
        }
        cout << (ans == 0 ? "second" : "first") << "\n";
    }
}
```

Implementação do Nim de escada (continuado)

Entrada	Saída
3 3 0 2 1 4 1 1 1 1 2 5 3	first second first

Nim de escada só que reescrito

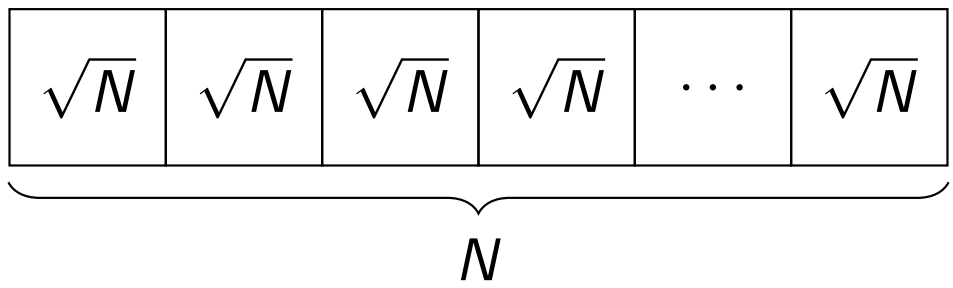
Esses problemas acabam sendo só o problema do Nim de escada:

- Pegar um valor do filho para um pai em um árvore, altura par é válida, altura ímpar inválida.
- Mover pedras para a esquerda, pulando pedras que já estão colocadas no tabuleiro.

Decomposição em Raiz Quadrada

O que vou decompor? Para quê?

- A decomposição em raiz quadrada é um termo genérico para a ideia de fragmentar uma sequência em blocos de tamanho $\sqrt{N}$.
- Por que $\sqrt{N}$? Porque são $\sqrt{N}$ blocos de tamanho $\sqrt{N}$, oras!
- Em outras palavras, $\sqrt{N} = \frac{N}{\sqrt{N}}$.



- Isso nos dá a possibilidade de iterar não só de um em um, mas também de bloco em bloco.
- A complexidade de $\mathcal{O}(\sqrt{N})$ é melhor que $\mathcal{O}(N)$ (melhor seria se fosse $\mathcal{O}(\lg N)$, mas é o que tem).
- O tamanho ótimo dos blocos pode não ser *exatamente* $\sqrt{n}$, mudar um pouco esse valor por vezes dá resultados melhores.

Problema da Soma em Intervalo (de novo!)

Dado um vetor de tamanho N, ache Q somas de intervalo $[L, R]$.

- Podemos resolver usando árvores em $\mathcal{O}(Q \lg N)$.
- Vamos resolver usando decomposição em raiz quadrada em $\mathcal{O}(Q\sqrt{N})$ (o que é pior!).

Ideia:

- Dividir o vetor em blocos de tamanho $\lceil \sqrt{N} \rceil$
- Guardar a resposta da soma para cada bloco
- Para responder consultas, juntamos as respostas:
  - Blocos cobertos: Somamos o valor deles. Serão no máximo $\mathcal{O}(\sqrt{N})$ blocos.
  - Blocos parcialmente cobertos: Calculamos "manualmente". Serão 2 blocos de tamanho $\mathcal{O}(\sqrt{N})$

Implementação da Soma em Intervalo

Código `sumsqrt.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
int main() { int n, q; cin >> n >> q;
    int b = ceil(sqrt(n));
    vector<ll> v (n), s (n/b + 1);
    for (int i = 0; i < n; i++) {
        cin >> v[i]; s[i/b] += v[i]; }
    while (q--) { int l, r; cin >> l >> r; l--; r--;
        int bl = l/b, br = r/b;
        ll ans = 0;
        if (l/b == r/b) {
            for (int i = l; i <= r; i++) { ans += v[i]; }
            cout << ans << "\n"; continue; }
        for (int i = l; i <= (bl+1)*b-1; i++) { ans += v[i]; }
        for (int bi = bl+1; bi <= br-1; bi++) { ans += s[bi]; }
        for (int i = br*b; i <= r; i++) { ans += v[i]; }
        cout << ans << "\n";
    }
}
```

Implementação da Soma em Intervalo (continuado)

Entrada	Saída
8 4 3 2 4 5 1 1 5 3 2 4 5 6 1 8 3 3	11 2 24 4

Problema da Remoção em Lista

É dado uma lista de tamanho N, remova-os nas posições dadas.

$$\left[{}_0 2 {}_1 6 {}_2 1 {}_3\right]$$

Removendo o elemento na posição 1-indexada 2:

$$\left[{}_0 2 {}_1 4 {}_2\right]$$

Removendo o elemento na posição 1-indexada 1:

$$\left[{}_0 4 {}_1\right]$$

- Sabemos resolver em $O(N \lg N)$ usando a Árvore de Fenwick.
- Que tal $O(N\sqrt{N})$ usando decomposição em raiz quadrada?

Implementação da Remoção em Lista

Código `remove.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5+15, B = ceil(sqrt(N));
list<int> l; vector<int> bsz (N/B + 1);
vector<list<int>::iterator> bbegin (N/B + 1);
int main() {
    int n; cin >> n;
    for (int i = 0; i < n; i++) {
        int x; cin >> x; auto ins = l.insert(l.end(), x);
        if (i % B == 0) { bbegin[i/B] = ins; }
        bsz[i/B]++; }
    while (n--) { int i; cin >> i; i--;
        auto it = bbegin[i/B]; advance(it, i%B);
        if (i % B == 0) { bbegin[i/B]++; }
        cout << *it << " "; l.erase(it);
        int bi = i/B;
        while (bsz[bi+1] > 0) { bbegin[bi+1]++; bi++; }
        bsz[bi]--; }
    cout << "\n";
}
```

Implementação da Remoção em Lista (continuado)

Entrada	Saída
5 2 6 1 4 2 3 1 3 1 1	1 2 2 6 4

Algoritmo de Mo

- Na decomposição em raiz quadrada, pré-computamos os valores de cada bloco (e unimo-nos ao responder consultas).
- A união pode ser custosa demais para certos problemas.
- Podemos contornar isso respondendo às consultas de maneira *offline*.

Problema do Elemento Mais Frequente no Intervalo

É dado um vetor de tamanho N, ache qual a frequência do elemento mais frequente do intervalo $[L, R]$.

- Esse é o problema de achar a "moda" em um intervalo.
- Tem uma página da Wikipédia muito boa do problema: `https://en.wikipedia.org/wiki/Range_mode_query`.
- Em suma, não usando $\mathcal{O}(N^2)$ de memória, a melhor solução é com consultas em $\mathcal{O}(\sqrt{N})$.
- Uma das formas é usando o Algoritmo de Mo.

Ideia do algoritmo de Mo

Ordenamos as consultas por índice e construímos a resposta caminhando pelo vetor

- Ordenamos as consultas por bloco: primeiro responderemos todas que começam no bloco 0, depois todas que começam no bloco 1, e assim em diante
- Além disso, entre as consultas que começam no mesmo bloco, faremos as que terminam em maior índice primeiro
- Usaremos uma única estrutura de dados para manter informação sobre o intervalo (inicialmente vazia)
- Estendemos ou reduzimos o intervalo de acordo com a consulta atual adicionando ou removendo elementos da estrutura

Para o problema de encontrar a soma do intervalo, por exemplo, a estrutura seria simplesmente um inteiro inicializado em 0 e adicionaríamos e removeríamos elementos com operações de adição e subtração.

Complexidade

- Ordenar as Q consultas: $\mathcal{O}(Q \log Q)$
- Remover/adicionar um elemento à direita em um bloco: $\mathcal{O}(N)$
- Combinando todos os blocos[1]: $\mathcal{O}(N\sqrt{N})$
- Remover/adicionar um elemento à esquerda em um bloco: $\mathcal{O}(\sqrt{N})$
- Combinando todos os blocos: $\mathcal{O}(\sqrt{N}Q)$
- Combinando tudo com as operações de adicionar e remover ($\mathcal{O}(F)$) da estrutura de dados: $\mathcal{O}(F(N+Q)\sqrt{N})$

[1] Assumindo um bloco de tamanho $\sqrt{N}$

Implementação do Algoritmo de Mo

Código `mo.h`

```
vector<int> mo(vector<qry> qs) {
    vector<int> ans (qs.size());
    sort(qs.begin(), qs.end(), [](qry a, qry b) {
        if (a.l/B != b.l/B) {
            return make_pair(a.l, a.r) < make_pair(b.l, b.r);
        }
        return (a.l/B) % 2 ? a.r < b.r : a.r > b.r;
    });
    int l = 0, r = -1;
    for (qry q : qs) {
        while (l > q.l) { l--; ins(l, 'l'); }
        while (r < q.r) { r++; ins(r, 'r'); }
        while (l < q.l) { rem(l, 'l'); l++; }
        while (r > q.r) { rem(r, 'r'); r--; }
        ans[q.ix] = get_ans();
    }
    return ans;
}
```

Implementação da Moda no Intervalo

Código `mode.h`

```
vector<int> v (N), freqfreq (N), freq (N);
int ans = 0;
void ins(int i, char dir) {
    freqfreq[freq[v[i]]]--;
    freq[v[i]]++;
    freqfreq[freq[v[i]]]++;
    if (freqfreq[ans+1]) { ans++; }
}
void rem(int i, char dir) {
    freqfreq[freq[v[i]]]--;
    freq[v[i]]--;
    freqfreq[freq[v[i]]]++;
    if (ans > 0 && freqfreq[ans] == 0) { ans--; }
}
int get_ans() {
    return ans;
}
```

Implementação da Moda no Intervalo (continuado)

Código `mode.cpp`

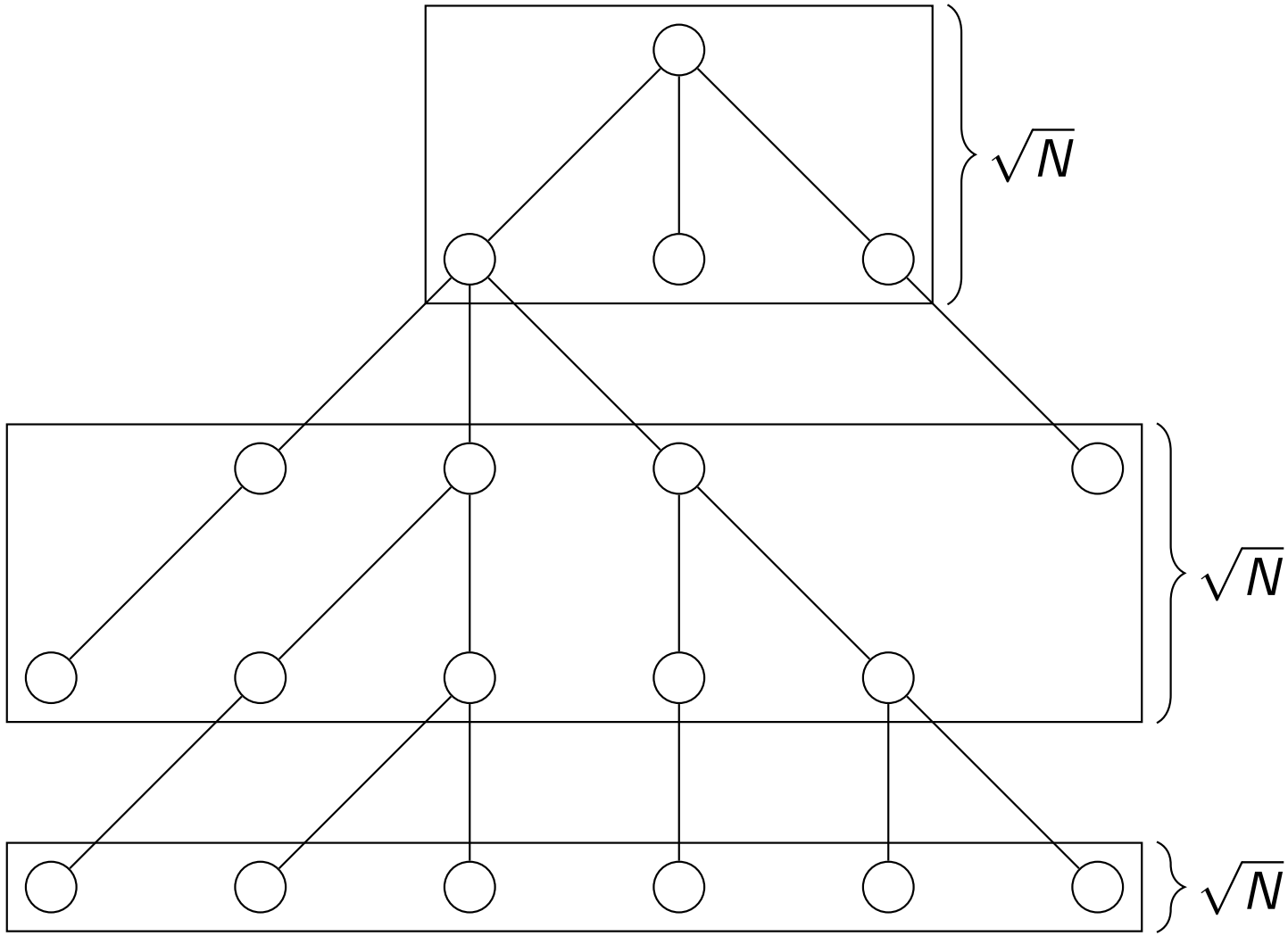
```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15, B = sqrt(ceil(N));
struct qry { int l, r, ix; };
#include "mode.h"
#include "mo.h"
int main() {
    int n, q; cin >> n >> q;
    for (int i = 0; i < n; i++) { cin >> v[i]; }
    vector<qry> qs;
    for (int i = 0; i < q; i++) {
        int l, r; cin >> l >> r; l--; r--;
        qs.push_back({ .l = l, .r = r, .ix = i });
    }
    vector<int> ans = mo(qs);
    for (int i = 0; i < q; i++) {
        cout << ans[i] << "\n";
    }
}
```

Implementação da Moda no Intervalo (continuado)

Entrada	Saída
5 3 1 2 1 3 3 1 3 2 3 1 5	2 1 2

Decomposição em raiz quadrada em árvore

- Dividindo uma árvore enraizada em blocos $\mathcal{O}(\sqrt{N})$.



Implementação da decomposição de árvore

Código `stdt.h`

```
vector<int> depth (N);
vector<int> up (N); vector<int> weiop (N);
vector<int> bup (N); vector<int> bweiop (N);
void stdt_decompose(int u, int p, int w) {
    up[u] = p; weiop[u] = w;
    depth[u] = depth[p] + 1;
    bup[u] = depth[u] % B ? bup[p] : p;
    bweiop[u] = OP(depth[u] % B ? bweiop[p] : NEUTRAL, w);
    for (auto [v, w] : g[u]) if (v != p)
        stdt_decompose(v, u, w);
}
```

Menor ancestral comum

Código `stdtlca.h`

```
int std_lca(int a, int b) {
    if (!(depth[a]/B > depth[b]/B)) { swap(a, b); }
    while (depth[a]/B > depth[b]/B) { a = bup[a]; }
    if (!(depth[a] > depth[b])) { swap(a, b); }
    while (depth[a] > depth[b]) { a = up[a]; }
    while (a != b) { a = up[a]; b = up[b]; }
    return a;
}
```

Operação no caminho

Código `stdtop.h`

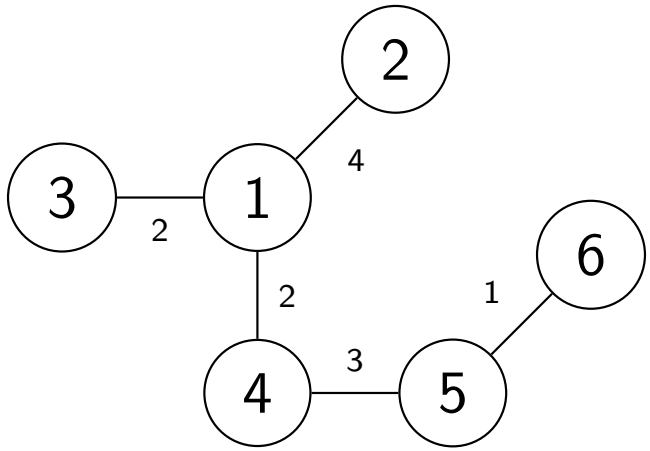
```
int stdt_op(int a, int b) {
    int ans = NEUTRAL;
    if (!(depth[a]/B > depth[b]/B)) { swap(a, b); }
    while (depth[a]/B > depth[b]/B) {
        ans = OP(ans, bweiop[a]); a = bup[a]; }
    if (!(depth[a] > depth[b])) { swap(a, b); }
    while (depth[a] > depth[b]) {
        ans = OP(ans, weiop[a]); a = up[a]; }
    while (a != b) {
        ans = OP(ans, OP(weiop[a], weiop[b]));
        a = up[a]; b = up[b];
    }
    return ans;
}
```

Maior peso entre dois vértices de uma árvore

Código `stdtmax.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ii = pair<int, int>;
const int N = 1e5+15, B = ceil(sqrt(N));
vector<vector<ii>> g (N);
#define NEUTRAL 0
#define OP(X, Y) max(X, Y)
#include "stdt.h"
#include "stdtop.h"
int main() {
    int n, m, q; cin >> n >> m >> q;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        g[u].push_back(ii(v, w)); g[v].push_back(ii(u, w));
    }
    stdt_decompose(0, 0, 0);
    while (q--) {
        int u, v; cin >> u >> v; u--; v--;
        cout << stdt_op(u, v) << "\n";
    }
}
```

Exemplo de maior peso entre dois vértices de uma árvore



Entrada	Saída
6 5 5 3 1 2 1 4 2 2 1 4 4 5 3 5 6 1 1 6 2 6 3 2 3 4 5 6	3 4 4 2 1

E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Existem outras decomposições como a Decomposição Centróide que resolvem outros problemas.
- A Decomposição Pesado-Leve é extremamente versátil porém, e serve para resolver muitos outros problemas.
- Mais variações de Nim: `https://www.youtube.com/watch?v=_99F4as2V6c`.
- Mais informações sobre k-Slow Nim e Nim monotônico: `https://arxiv.org/pdf/1705.06774.pdf`.
- Jogos em grafos arbitrários: `https://cp-algorithms.com/game_theory/games_on_graphs.html`.
- Bons estudos!