Bibliotecas e Funções C++ para Competição - Guia Detalhado

Inclusões Básicas e Configurações

```
// Inclui TODAS as bibliotecas padrão C++ - útil para competições
#include <bits/stdc++.h>
using namespace std;

// Acelera entrada/saída - CRUCIAL para competições
ios_base::sync_with_stdio(false);
cin.tie(NULL);
cout.tie(NULL);

// Por que usar?
// - sync_with_stdio(false): desvincula cin/cout de scanf/printf
// - tie(NULL): desvincula cin de cout (evita flush automático)
// Resultado: I/O 3-5x mais rápido
```

Typedefs e Macros Úteis

```
// Atalhos para tipos comuns
typedef long long ll;           // 64-bit integers
typedef long double ld;         // Precisão decimal
typedef vector<int> vi;         // Vector de inteiros
typedef vector<ll> vll;         // Vector de long long
typedef pair<int, int> pii;     // Par de inteiros
typedef vector<pii> vii;        // Vector de pares

// Macros para loops
#define rep(i, a, b) for(int i = a; i < (b); i++)
#define repr(i, a, b) for(int i = a; i >= (b); i--)
#define sz(x) (int)(x).size()   // Tamanho de container
#define all(x) begin(x), end(x) // Iteradores begin/end
```

Funções de Array/Vector Explicadas

Ordenação e Busca

```
vector<int> v = {5, 2, 8, 1, 9};

// sort() - Ordena em ordem crescente
sort(v.begin(), v.end());        // v = {1, 2, 5, 8, 9}

// sort() com função comparadora
sort(v.begin(), v.end(), greater<int>()); // v = {9, 8, 5, 2, 1}

// Ordenar em ordem decrescente alternativa
sort(v.rbegin(), v.rend());      // v = {9, 8, 5, 2, 1}

// lower_bound() - Primeiro elemento NÃO MENOR que x
// Requer que o array esteja ordenado!
auto it = lower_bound(v.begin(), v.end(), 5);
// Retorna iterator para primeiro elemento >= 5
if (it != v.end()) {
    int index = it - v.begin();  // Índice do elemento
    int value = *it;             // Valor do elemento
}

// upper_bound() - Primeiro elemento MAIOR que x
auto it = upper_bound(v.begin(), v.end(), 5);
// Retorna iterator para primeiro elemento > 5

// binary_search() - Verifica se elemento existe
bool exists = binary_search(v.begin(), v.end(), 5);
```

Operações em Arrays

```
vector<int> v = {1, 2, 3, 4, 5};

// reverse() - Inverte a ordem dos elementos
reverse(v.begin(), v.end());     // v = {5, 4, 3, 2, 1}

// max_element() - Iterator para o maior elemento
auto max_it = max_element(v.begin(), v.end());
int max_val = *max_it;           // Valor máximo = 5
int max_index = max_it - v.begin(); // Índice do máximo

// min_element() - Iterator para o menor elemento
auto min_it = min_element(v.begin(), v.end());

// accumulate() - Soma todos os elementos
int sum = accumulate(v.begin(), v.end(), 0); // Soma = 15
// O terceiro parâmetro é o valor inicial

// count() - Conta ocorrências de um valor
int count_3 = count(v.begin(), v.end(), 3); // count_3 = 1

// find() - Encontra primeira ocorrência
auto it = find(v.begin(), v.end(), 3);
if (it != v.end()) {
    // Elemento encontrado
}
```

Funções Matemáticas Detalhadas

```cpp
#include <cmath>
#include <numeric>

// GCD - Maior Divisor Comum (Algoritmo de Euclides)
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// LCM - Mínimo Múltiplo Comum
int lcm(int a, int b) {
    return (a / gcd(a, b)) * b;  // Evita overflow
}

// Exemplo de uso:
int a = 12, b = 18;
cout << gcd(a, b) << endl;  // 6
cout << lcm(a, b) << endl;  // 36

// Potência modular (útil para números grandes)
ll modpow(ll base, ll exp, ll mod) {
    ll result = 1;
    base %= mod;
    while (exp > 0) {
        if (exp & 1)  // Se exp é ímpar
            result = (result * base) % mod;
        base = (base * base) % mod;
        exp >>= 1;    // exp = exp / 2
    }
    return result;
}

// Exemplo: 3^13 mod 1000000007
ll result = modpow(3, 13, 1000000007);

// Combinações com fatorial pré-computado
vector<ll> fact(1000001), inv_fact(1000001);

void precompute_factorials(int n, ll mod) {
    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (fact[i-1] * i) % mod;
    }

    // Inverso modular usando Fermat
    inv_fact[n] = modpow(fact[n], mod-2, mod);
    for (int i = n-1; i >= 0; i--) {
        inv_fact[i] = (inv_fact[i+1] * (i+1)) % mod;
    }
}

ll nCr(int n, int r, ll mod) {
    if (r < 0 || r > n) return 0;
    return (fact[n] * inv_fact[r] % mod) * inv_fact[n-r] % mod;
}
```

Manipulação de Strings

```
string s = "Hello World";

// Conversões
int num = 123;
string str_num = to_string(num);    // "123"
int back_to_num = stoi(str_num);    // 123

// Operações com strings
s.substr(6, 5);        // "World" (posição, tamanho)
s.find("World");       // Retorna posição (6)
s.replace(6, 5, "CPP"); // "Hello CPP"
s.erase(5, 6);         // "Hello"

// Transformações
transform(s.begin(), s.end(), s.begin(), ::toupper);
// "HELLO WORLD"

// String stream para parsing
string data = "123 456 789";
stringstream ss(data);
int a, b, c;
ss >> a >> b >> c;     // a=123, b=456, c=789

// Dividir string por delimitador
vector<string> split(string s, char delimiter) {
    vector<string> tokens;
    string token;
    stringstream ss(s);
    while (getline(ss, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}
```

Estruturas de Dados Avançadas

Union-Find (Disjoint Set Union)

```
class DSU {
private:
    vector<int> parent, rank, size;

public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        size.resize(n, 1);
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    // Find com path compression
    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }

    // Union by rank
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) return;

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
            size[rootY] += size[rootX];
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
            size[rootX] += size[rootY];
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
            size[rootX] += size[rootY];
        }
    }

    // Tamanho do conjunto
    int getSize(int x) {
        return size[find(x)];
    }

    // Verifica se estão no mesmo conjunto
    bool connected(int x, int y) {
        return find(x) == find(y);
    }
};
```

Segment Tree para Soma de Range

```
class SegmentTree {
private:
    vector<int> tree;
    int n;

    void build(vector<int>& nums, int node, int start, int end) {
        if (start == end) {
            tree[node] = nums[start];
        } else {
            int mid = (start + end) / 2;
            build(nums, 2*node, start, mid);
            build(nums, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    void update(int node, int start, int end, int idx, int val) {
        if (start == end) {
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;
            if (idx <= mid) {
                update(2*node, start, mid, idx, val);
            } else {
                update(2*node+1, mid+1, end, idx, val);
            }
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    int query(int node, int start, int end, int l, int r) {
        if (r < start || l > end) return 0;
        if (l <= start && end <= r) return tree[node];

        int mid = (start + end) / 2;
        int left_sum = query(2*node, start, mid, l, r);
        int right_sum = query(2*node+1, mid+1, end, l, r);
        return left_sum + right_sum;
    }

public:
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        tree.resize(4 * n);
        build(nums, 1, 0, n-1);
    }

    void update(int idx, int val) {
        update(1, 0, n-1, idx, val);
    }

    int query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

Algoritmos de Grafos

BFS com Distância

```
vector<int> bfs(int start, vector<vector<int>>& graph) {
    int n = graph.size();
    vector<int> dist(n, -1);  // -1 significa não visitado
    queue<int> q;

    dist[start] = 0;
    q.push(start);

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        for (int neighbor : graph[current]) {
            if (dist[neighbor] == -1) {
                dist[neighbor] = dist[current] + 1;
                q.push(neighbor);
            }
        }
    }
    return dist;
}
```

DFS para Componentes Conectados

```
void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited) {
    visited[node] = true;

    for (int neighbor : graph[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, graph, visited);
        }
    }
}

int countConnectedComponents(int n, vector<vector<int>>& graph) {
    vector<bool> visited(n, false);
    int components = 0;

    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            components++;
            dfs(i, graph, visited);
        }
    }
    return components;
}
```

Funções Úteis Adicionais

Geração de Permutações

```
vector<int> v = {1, 2, 3};

// Gera todas as permutações em ordem lexicográfica
do {
    // Processa permutação atual
    for (int x : v) cout << x << " ";
    cout << endl;
} while (next_permutation(v.begin(), v.end()));

// Para gerar todas as permutações, o array deve estar ordenado!
```

Operações com Bits

```
// Manipulação de bits
int x = 13;  // 1101 em binário

// Contar bits setados
int count_bits = __builtin_popcount(x);  // 3

// Checar se é potência de 2
bool is_power_of_2 = (x & (x-1)) == 0;

// Máscaras de bits
int set_bit = x | (1 << 2);    // Seta bit 2: 1101 → 1101 (já setado)
int clear_bit = x & ~(1 << 0); // Limpa bit 0: 1101 → 1100
int toggle_bit = x ^ (1 << 1); // Alterna bit 1: 1101 → 1111
```

Tempo de Execução

```
#include <chrono>

auto start = chrono::high_resolution_clock::now();

// Seu código aqui

auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "Tempo: " << duration.count() << "ms" << endl;
```

Template Completo para Competições

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

typedef long long ll;
typedef long double ld;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int, int> pii;
typedef vector<pii> vii;

#define fastio ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
#define rep(i, a, b) for(int i = a; i < (b); i++)
#define sz(x) (int)(x).size()
#define all(x) begin(x), end(x)
#define rall(x) rbegin(x), rend(x)
#define debug(x) cerr << #x << " = " << x << endl

const int MOD = 1e9 + 7;
const int INF = 1e9;
const ll LLINF = 1e18;

void solve() {
    // Sua solução aqui
}

int main() {
    fastio;

    int t = 1;
    cin >> t;
    while (t--) {
        solve();
    }

    return 0;
}
```

Estruturas de Dados Avançadas para Competição - Explicação Detalhada

1. Union-Find (Disjoint Set Union) - Explicação Completa

Conceito e Aplicações

```cpp
/*
O DSU gerencia uma coleção de conjuntos disjuntos (sem elementos em comum)
Aplicações:
- Verificar conectividade em grafos dinâmicos
- Componentes conectadas
- Kruskal's algorithm (MST)
- Detecção de ciclos
*/
```

Implementação Comentada Passo a Passo

```cpp
class DSU {
private:
    vector<int> parent;  // parent[i] = pai do elemento i
    vector<int> rank;    // rank[i] = altura da árvore (otimização)
    vector<int> size;    // size[i] = tamanho do conjunto (opcional)
```

```
public:
    // Inicializa n conjuntos (cada elemento é seu próprio pai)
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);     // Todas as árvores começam com altura 0
        size.resize(n, 1);     // Cada conjunto tem tamanho 1 inicialmente

        // Cada elemento é seu próprio representante inicialmente
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // FIND com Path Compression
    // Encontra o representante (raiz) do conjunto
    int find(int x) {
        if (parent[x] != x) {
            // Path compression: achatamos a árvore durante a busca
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // UNION by Rank
    // Une dois conjuntos
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        // Se já estão no mesmo conjunto, não faz nada
        if (rootX == rootY) return;

        // Union by Rank: anexa a árvore menor à maior
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
            size[rootY] += size[rootX];
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
            size[rootX] += size[rootY];
        } else {
            // Mesmo rank: escolhe um como pai e incrementa seu rank
            parent[rootY] = rootX;
            rank[rootX]++;
            size[rootX] += size[rootY];
        }
    }

    // Funções utilitárias
    bool connected(int x, int y) {
        return find(x) == find(y);
    }

    int getSize(int x) {
        return size[find(x)];
    }

    int getSets() {
        int count = 0;
        for (int i = 0; i < parent.size(); i++) {
            if (parent[i] == i) count++;
        }
        return count;
    }
};
```

Exemplo de Uso Prático

```
// Problema: Verificar se dois nós estão conectados em um grafo dinâmico
int main() {
    int n = 5;  // 5 elementos: 0,1,2,3,4
    DSU dsu(n);

    // Conectar elementos
    dsu.unite(0, 1);  // Conjunto: {0,1}
    dsu.unite(2, 3);  // Conjunto: {2,3}
    dsu.unite(1, 2);  // Conjunto: {0,1,2,3}

    cout << dsu.connected(0, 3) << endl;  // true (1)
    cout << dsu.connected(0, 4) << endl;  // false (0)
    cout << dsu.getSize(0) << endl;       // 4
    cout << dsu.getSets() << endl;        // 2 ({0,1,2,3} e {4})

    return 0;
}
```

2. Segment Tree - Explicação Visual

Conceito da Árvore de Segmentos

```
 Array: [2, 4, 1, 7, 3, 5, 6, 2]
Índices: 0  1  2  3  4  5  6  7

Árvore de Segmentos (para soma):
Nível 0:                  [0-7: 30]
Nível 1:        [0-3: 14]          [4-7: 16]
Nível 2:   [0-1: 6]  [2-3: 8]   [4-5: 8]  [6-7: 8]
Nível 3: [0:2][1:4] [2:1][3:7] [4:3][5:5] [6:6][7:2]
```

Implementação Comentada

```
 class SegmentTree {
private:
    vector<int> tree;  // Array que representa a árvore
    int n;             // Tamanho do array original

    // Constrói a árvore recursivamente
    void build(vector<int>& nums, int node, int start, int end) {
        // node: índice atual na árvore
        // start, end: intervalo no array original que este nó representa

        if (start == end) {
            // Nó folha: armazena o valor do array original
            tree[node] = nums[start];
        } else {
            int mid = (start + end) / 2;

            // Constroi subárvore esquerda (filho 2*node)
            build(nums, 2*node, start, mid);
            // Constroi subárvore direita (filho 2*node + 1)
            build(nums, 2*node + 1, mid + 1, end);

            // Combina resultados dos filhos (soma neste caso)
            tree[node] = tree[2*node] + tree[2*node + 1];
        }
    }

    // Atualiza um elemento
    void update(int node, int start, int end, int idx, int val) {
```

```
    void update(int node, int start, int end, int idx, int val) {
        if (start == end) {
            // Encontrou o elemento: atualiza
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;

            if (idx <= mid) {
                // Elemento está na subárvore esquerda
                update(2*node, start, mid, idx, val);
            } else {
                // Elemento está na subárvore direita
                update(2*node + 1, mid + 1, end, idx, val);
            }

            // Atualiza o nó atual com os novos valores dos filhos
            tree[node] = tree[2*node] + tree[2*node + 1];
        }
    }

    // Consulta um intervalo [l, r]
    int query(int node, int start, int end, int l, int r) {
        if (r < start || l > end) {
            // Intervalo [l,r] completamente fora de [start,end]
            return 0;  // Elemento neutro da operação (0 para soma)
        }

        if (l <= start && end <= r) {
            // Intervalo [start,end] completamente dentro de [l,r]
            return tree[node];
        }

        // Intervalo [start,end] parcialmente sobreposto com [l,r]
        int mid = (start + end) / 2;
        int left_sum = query(2*node, start, mid, l, r);
        int right_sum = query(2*node + 1, mid + 1, end, l, r);

        return left_sum + right_sum;
    }

public:
    SegmentTree(vector<int>& nums) {
        n = nums.size();
        tree.resize(4 * n);  // Tamanho seguro: 4 * n
        build(nums, 1, 0, n-1);
    }

    void update(int idx, int val) {
        update(1, 0, n-1, idx, val);
    }

    int query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

Exemplo Prático com Debug

```
// Demonstração passo a passo
int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11};
    SegmentTree seg(arr);

    cout << "Array original: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    // Consultas
    cout << "Soma [0,2]: " << seg.query(0, 2) << endl;  // 1+3+5 = 9
    cout << "Soma [1,4]: " << seg.query(1, 4) << endl;  // 3+5+7+9 = 24

    // Atualização
    cout << "\nAtualizando índice 2 de 5 para 10..." << endl;
    seg.update(2, 10);

    cout << "Nova soma [0,2]: " << seg.query(0, 2) << endl;  // 1+3+10 = 14
    cout << "Nova soma [1,4]: " << seg.query(1, 4) << endl;  // 3+10+7+9 = 29

    return 0;
}
```

3. Fenwick Tree (Binary Indexed Tree)

Conceito Baseado em Bits

```
/*
Fenwick Tree é mais simples que Segment Tree para:
- Soma de prefixo (prefix sum)
- Atualizações de ponto

Índices são baseados em representação binária:
Índice 1: 001 - Responsável por [1,1]
Índice 2: 010 - Responsável por [1,2]
Índice 3: 011 - Responsável por [3,3]
Índice 4: 100 - Responsável por [1,4]
*/
```

Implementação Eficiente

```
class FenwickTree {
private:
    vector<int> bit;  // Binary Indexed Tree
    int n;

    // Obtém o valor do último bit setado (LSB)
    int lsb(int i) {
        return i & -i;
    }

public:
    FenwickTree(int size) {
        n = size;
        bit.resize(n + 1, 0);  // Índices de 1 a n
    }

    // Constrói a BIT a partir de um array
    FenwickTree(vector<int>& arr) {
        n = arr.size();
        bit.resize(n + 1, 0);

        for (int i = 0; i < n; i++) {
            update(i, arr[i]);
        }
    }

    // Atualiza: adiciona 'val' na posição 'index' (0-based)
    void update(int index, int val) {
        // Converte para 1-based
        int i = index + 1;

        while (i <= n) {
            bit[i] += val;
            i += lsb(i);  // Move para o próximo nó responsável
        }
    }

    // Query: soma do prefixo [0, index]
    int prefixSum(int index) {
        int sum = 0;
        int i = index + 1;  // Converte para 1-based

        while (i > 0) {
            sum += bit[i];
            i -= lsb(i);  // Move para o nó anterior
        }
        return sum;
    }

    // Query: soma do intervalo [l, r]
    int rangeSum(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }
};
```

Comparação Fenwick vs Segment Tree

```
/*
FENWICK TREE (Vantagens):
- Menor constante de tempo
- Código mais simples
- Menor uso de memória
- Melhor para soma de prefixo

SEGMENT TREE (Vantagens):
- Mais genérica (qualquer operação associativa)
- Suporta range updates
- Mais flexível para queries complexas

ESCOLHA:
- Use Fenwick para soma/prefix sum
- Use Segment Tree para outras operações (min, max, gcd, etc.)
*/
```

4. Trie (Árvore de Prefixos)

Estrutura para Strings

```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord;
    int count;  // Quantas palavras terminam aqui

    TrieNode() : isEndOfWord(false), count(0) {}
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    // Insere uma palavra
    void insert(string word) {
        TrieNode* node = root;

        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                node->children[c] = new TrieNode();
            }
            node = node->children[c];
        }

        node->isEndOfWord = true;
        node->count++;
    }

    // Busca uma palavra (retorna se existe)
    bool search(string word) {
        TrieNode* node = root;

        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }
```

```
        }

        return node->isEndOfWord;
    }

    // Verifica se existe palavra com dado prefixo
    bool startsWith(string prefix) {
        TrieNode* node = root;

        for (char c : prefix) {
            if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }

        return true;
    }

    // Conta quantas palavras têm dado prefixo
    int countWordsWithPrefix(string prefix) {
        TrieNode* node = root;

        // Navega até o final do prefixo
        for (char c : prefix) {
            if (node->children.find(c) == node->children.end()) {
                return 0;
            }
            node = node->children[c];
        }

        // Conta todas as palavras nesta subárvore
        return countWordsInSubtree(node);
    }

private:
    int countWordsInSubtree(TrieNode* node) {
        if (!node) return 0;

        int count = 0;
        if (node->isEndOfWord) {
            count += node->count;
        }

        for (auto& child : node->children) {
            count += countWordsInSubtree(child.second);
        }

        return count;
    }
};
```

Exemplo de Uso do Trie

```
int main() {
    Trie trie;

    // Inserindo palavras
    trie.insert("apple");
    trie.insert("app");
    trie.insert("application");
    trie.insert("banana");

    // Buscas
    cout << trie.search("app") << endl;      // true
    cout << trie.search("apple") << endl;    // true
    cout << trie.search("appl") << endl;     // false

    // Prefixos
    cout << trie.startsWith("app") << endl;  // true
    cout << trie.countWordsWithPrefix("app") << endl; // 3

    return 0;
}
```

5. Segment Tree com Lazy Propagation

Para Range Updates

```
class LazySegmentTree {
private:
    vector<int> tree, lazy;
    int n;

    void build(vector<int>& nums, int node, int start, int end) {
        if (start == end) {
            tree[node] = nums[start];
        } else {
            int mid = (start + end) / 2;
            build(nums, 2*node, start, mid);
            build(nums, 2*node+1, mid+1, end);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    // Propaga atualizações pendentes
    void propagate(int node, int start, int end) {
        if (lazy[node] != 0) {
            // Aplica a atualização lazy
            tree[node] += (end - start + 1) * lazy[node];

            // Propaga para filhos se não for folha
            if (start != end) {
                lazy[2*node] += lazy[node];
                lazy[2*node+1] += lazy[node];
            }

            lazy[node] = 0;  // Limpa o lazy
        }
    }

    // Range update: adiciona 'val' a todos em [l, r]
    void update(int node, int start, int end, int l, int r, int val) {
        propagate(node, start, end);

        if (r < start || l > end) return;

        if (l <= start && end <= r) {
```

```
        if (l <= start && end <= r) {
            // [start,end] completamente dentro de [l,r]
            lazy[node] += val;
            propagate(node, start, end);
        } else {
            int mid = (start + end) / 2;
            update(2*node, start, mid, l, r, val);
            update(2*node+1, mid+1, end, l, r, val);
            tree[node] = tree[2*node] + tree[2*node+1];
        }
    }

    int query(int node, int start, int end, int l, int r) {
        propagate(node, start, end);

        if (r < start || l > end) return 0;
        if (l <= start && end <= r) return tree[node];

        int mid = (start + end) / 2;
        int left_sum = query(2*node, start, mid, l, r);
        int right_sum = query(2*node+1, mid+1, end, l, r);
        return left_sum + right_sum;
    }

public:
    LazySegmentTree(vector<int>& nums) {
        n = nums.size();
        tree.resize(4 * n);
        lazy.resize(4 * n, 0);
        build(nums, 1, 0, n-1);
    }

    void rangeUpdate(int l, int r, int val) {
        update(1, 0, n-1, l, r, val);
    }

    int rangeQuery(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

Exemplo de Lazy Propagation

```
int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    LazySegmentTree seg(arr);

    cout << "Original: ";
    for (int i = 0; i < 5; i++)
        cout << seg.rangeQuery(i, i) << " ";
    cout << endl;

    // Adiciona 10 aos elementos [1,3]
    seg.rangeUpdate(1, 3, 10);

    cout << "Após update [1,3] +10: ";
    for (int i = 0; i < 5; i++)
        cout << seg.rangeQuery(i, i) << " ";
    cout << endl;  // 1, 12, 13, 14, 5

    cout << "Soma [0,4]: " << seg.rangeQuery(0, 4) << endl;  // 45

    return 0;
}
```

6. Heap Customizado (Priority Queue)

Max Heap e Min Heap Personalizados

```
// Comparador para Max Heap
struct MaxCompare {
    bool operator()(int a, int b) {
        return a < b;  // Para max heap, elemento maior tem maior prioridade
    }
};

// Comparador para Min Heap
struct MinCompare {
    bool operator()(int a, int b) {
        return a > b;  // Para min heap, elemento menor tem maior prioridade
    }
};

// Heap para pares (valor, índice)
struct PairCompare {
    bool operator()(pair<int, int> a, pair<int, int> b) {
        return a.first < b.first;  // Max heap baseado no primeiro elemento
    }
};

// Exemplos de uso:
priority_queue<int> maxHeap;  // Max heap padrão
priority_queue<int, vector<int>, greater<int>> minHeap;  // Min heap

// Heap customizado
priority_queue<int, vector<int>, MaxCompare> customMaxHeap;
priority_queue<pair<int, int>, vector<pair<int, int>>, PairCompare> pairHeap;
```

Quando Usar Cada Estrutura

Estrutura	Complexidade	Melhor Para	Código
DSU	O(α(n))	Conectividade dinâmica	Grafos, componentes
Segment Tree	O(log n)	Range queries complexas	Min, Max, GCD, etc.
Fenwick Tree	O(log n)	Prefix sum, point updates	Soma, contagem
Trie	O(L)	Strings, prefixos	Dicionários, autocomplete
Lazy Segment Tree	O(log n)	Range updates	Adicionar a intervalos

Essas estruturas cobrem 95% dos problemas de competição! Pratique cada uma com problemas específicos para dominá-las completamente.

Este guia cobre as funções mais essenciais com explicações detalhadas. Pratique cada função individualmente para entender completamente seu comportamento!