Aula 11 · Árvore de Segmentos Preguiçosa, Decomposição em Raiz Quadrada, Algoritmo de Mo e Decomposição Pesado-Leve

Desafios de Programação

Fernando Kiotheka Victor Alflen

UFPR

17/08/2022

Árvore de Segmentos Preguiçosa

Revisão breve de árvore de segmentos

- Árvore binária
- Cada vértice corresponde a um segmento de uma sequência
- $\mathcal{O}(n)$ espaço
- Operações e consultas em $\mathcal{O}(\lg n)$

Isso funciona bem para operações realizadas sobre um elemento de cada vez, mas para operar sobre segmentos arbitrários precisamos modificar a árvore um pouco.

Adição em segmento

Dado um vetor V, realizaremos operações de dois tipos:

- **Modificação:** adicionar um valor X a todos os números no intervalo $[L..R]$
- **Consulta:** consultar o valor de um número no índice I

Ideia de solução

Vamos guardar nos vértices da árvore quanto deve ser adicionado aos valores no segmento que cada um deles representa, mantendo a complexidade de $\mathcal{O}(\lg n)$.
Em consultas, basta buscar o elemento na árvore somando todos os valores ao longo do caminho.

Implementação da árvore de segmentos

Código `stsegadd.h`

```
vector<ll> t (4*N);
void add_inclusive(int l, int r, int d,
        int ti=1, int tl=1, int tr=N) {
    if (l > r) { return; }
    if (l == tl && r == tr) { t[ti] += d; return; }
    int tm = (tl + tr) / 2;
    add_inclusive(l, min(r, tm), d, ti*2, tl, tm);
    add_inclusive(max(l, tm+1), r, d, ti*2+1, tm+1, tr);
}
ll get(int i, int ti=1, int tl=1, int tr=N) {
    if (tl == tr) { return t[ti]; }
    int tm = (tl + tr) / 2;
    if (i <= tm) { return t[ti] + get(i, ti*2, tl, tm); }
    else { return t[ti] + get(i, ti*2+1, tm+1, tr); }
}
```

Implementação da construção da árvore de segmentos

Código `stsegaddbuild.h`

```
void build(vector<int>& src,
        int ti=1, int tl=1, int tr=N) {
    if (tl == tr) {
        if (tl < src.size()) { t[ti] = src[tl]; }
        return;
    }
    int tm = (tl+tr)/2;
    build(src, ti*2, tl, tm);
    build(src, ti*2+1, tm+1, tr);
    t[ti] = 0;
}
```

Utilização da adição em segmento

Código stsegadd.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 2e5+15;
#include "stsegadd.h"
#include "stsegaddbuild.h"
int main() {
    int n; cin >> n; vector<int> src (n+1);
    for (int i = 1; i <= n; i++) { cin >> src[i]; }
    build(src);
    char op; int a, b, d, i; string s;
    while (cin >> op)
        if (op == '+') {
            cin >> a >> b >> d; add_inclusive(a, b, d);
        } else if (op == 'q') {
            cin >> i; cout << get(i) << "\n";
        } else getline(cin, s);
}
```

Utilização da adição em segmento (continuado)

Entrada	Saída
5 1 2 0 0 0 + 3 5 3 # 1 2 3 3 3 q 1 q 2 q 3 + 1 2 1 # 2 3 3 3 3 q 1 + 1 3 -2 # 0 1 1 0 3 q 1 q 2 q 3	1 2 3 2 0 1 1

Atribuição em segmento

Dado um vetor V, realizaremos operações de dois tipos:

- **Modificação:** atribuir um valor X a todos os elementos no intervalo $[L..R]$
- **Consulta:** consultar o valor de um número no índice I

Ideia de solução

- Guardar em todos os vértices uma informação adicional: se ele está totalmente preenchido com o mesmo valor
- Em vez de atualizar todos os vértices cobertos pelo intervalo $[L..R]$ da operação, atualizamos apenas alguns
- Só passamos a informação à frente quando um vértice for dividido

Ideia de solução

- Se realizarmos uma atribuição sobre o intervalo $[0..N-1]$, só a raiz da árvore será alterada
- Em seguida, realizando outra atribuição sobre o intervalo $[0..N/2]$, a raiz da árvore deve "empurrar" a informação para seus dois nós filhos

Implementação da propagação

Código `stsegatrpush.h`

```
void push(int ti) {
    if (mark[ti]) {
        t[ti*2] = t[ti*2+1] = t[ti];
        mark[ti*2] = mark[ti*2+1] = true;
        mark[ti] = false;
    }
}
```

Implementação da árvore de segmentos

Código `stsegatr.h`

```
vector<int> t (4*N); vector<bool> mark (4*N);
#include "stsegatrpush.h"
void set_inclusive(int l, int r, int v,
        int ti=1, int tl=1, int tr=N) {
    if (l > r) { return; }
    if (l == tl && tr == r) {
        t[ti] = v; mark[ti] = true; return; }
    push(ti); int tm = (tl + tr) / 2;
    set_inclusive(l, min(r, tm), v, ti*2, tl, tm);
    set_inclusive(max(l, tm+1), r, v, ti*2+1, tm+1, tr);
}
int get(int i, int ti=1, int tl=1, int tr=N) {
    if (tl == tr) { return t[ti]; }
    push(ti); int tm = (tl + tr) / 2;
    if (i <= tm) { return get(i, ti*2, tl, tm); }
    else { return get(i, ti*2+1, tm+1, tr); }
}
```

Utilização da atribuição em segmento

Código `stsegatr.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 2e5+15;
#include "stsegatr.h"
#include "stsegaddbuild.h"
int main() {
    int n; cin >> n; vector<int> src (n+1);
    for (int i = 1; i <= n; i++) { cin >> src[i]; }
    build(src);
    char op; int a, b, v, i; string s;
    while (cin >> op)
        if (op == '=') {
            cin >> a >> b >> v; set_inclusive(a, b, v);
        } else if (op == 'q') {
            cin >> i; cout << get(i) << "\n";
        } else getline(cin, s);
}
```

Utilização da atribuição em segmento (continuado)

Entrada	Saída
5 1 2 0 0 0 = 4 5 3 # 1 2 0 3 3 q 1 q 2 q 3 q 4 = 1 2 1 # 1 1 0 3 3 q 1 = 2 3 2 # 1 2 2 3 3 q 1 q 2 q 4	1 2 0 3 1 1 2 3

Adição em segmento com consulta de máximo

Dado um vetor V, realizaremos operações de dois tipos:

- **Modificação:** adicionar um valor X a todos os elementos no intervalo $[L..R]$
- **Consulta:** consultar o valor máximo em um intervalo $[L..R]$

Ideia de solução

- Cada vértice da árvore guarda o valor máximo no segmento que representa e os adendos que não foram propagados ("empurrados" para os filhos)
- Realizamos a propagação tanto ao consultar quanto ao modificar

Implementação da propagação

Código `lstpush.h`

```
void push(int ti, int tl, int tm, int tr) {
    if (sety[ti] != -1) {
        t[ti*2] = sety[ti] * FACTOR(tm - tl + 1);
        lazy[ti*2] = 0; sety[ti*2] = sety[ti];
        t[ti*2+1] = sety[ti] * FACTOR(tr - (tm+1) + 1);
        lazy[ti*2+1] = 0; sety[ti*2+1] = sety[ti];
        sety[ti] = -1;
    }
    t[ti*2] += lazy[ti] * FACTOR(tm - tl + 1);
    lazy[ti*2] += lazy[ti];
    t[ti*2+1] += lazy[ti] * FACTOR(tr - (tm+1) + 1);
    lazy[ti*2+1] += lazy[ti];
    lazy[ti] = 0;
}
```

Implementação da árvore de segmentos preguiçosa

Código `lstaddset.h`

```
void set_inclusive(int l, int r, int d,
        int ti=1, int tl=1, int tr=N) {
    if (l > r) { return; }
    if (l == tl && tr == r) {
        t[ti] = ll(d) * FACTOR(tr - tl + 1);
        sety[ti] = d; lazy[ti] = 0; return; }
    int tm = (tl + tr) / 2; push(ti, tl, tm, tr);
    set_inclusive(l, min(r, tm), d, ti*2, tl, tm);
    set_inclusive(max(l, tm+1), r, d, ti*2+1, tm+1, tr);
    t[ti] = OP(t[ti*2], t[ti*2+1]);
}
void add_inclusive(int l, int r, int d,
        int ti=1, int tl=1, int tr=N) {
    if (l > r) { return; }
    if (l == tl && tr == r) {
        t[ti] += ll(d) * FACTOR(tr - tl + 1); lazy[ti] += d; return; }
    int tm = (tl + tr) / 2; push(ti, tl, tm, tr);
    add_inclusive(l, min(r, tm), d, ti*2, tl, tm);
    add_inclusive(max(l, tm+1), r, d, ti*2+1, tm+1, tr);
    t[ti] = OP(t[ti*2], t[ti*2+1]);
}
```

Implementação da árvore de segmentos preguiçosa

Código `lst.h`

```
vector<ll> t (4*N), lazy (4*N), sety (4*N, -1);
#include "lstpush.h"
#include "lstaddset.h"
ll op_inclusive(int l, int r,
        int ti=1, int tl=1, int tr=N) {
    if (l > r) { return NEUTRAL; }
    if (l <= tl && tr <= r) { return t[ti]; }
    int tm = (tl + tr) / 2; push(ti, tl, tm, tr);
    return OP(op_inclusive(l, min(r, tm), ti*2, tl, tm),
              op_inclusive(max(l, tm+1), r, ti*2+1, tm+1, tr));
}
```

Implementação da construção

Código `lstbuild.h`

```
void build(vector<int>& src,
        int ti=1, int tl=1, int tr=N) {
    if (tl == tr) {
        if (tl < src.size()) { t[ti] = src[tl]; }
        return;
    }
    int tm = (tl + tr) / 2;
    build(src, ti*2, tl, tm);
    build(src, ti*2+1, tm+1, tr);
    t[ti] = OP(t[ti*2], t[ti*2+1]);
}
```

Máximo na árvore de segmentos preguiçosa

Código `lst.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 1e5+15;
#define NEUTRAL 0
#define FACTOR(sz) 1
#define OP(X, Y) max(X, Y)
#include "lst.h"
#include "lstbuild.h"
int main() {
    int n; cin >> n; vector<int> src (n+1);
    for (int i = 1; i <= n; i++) { cin >> src[i]; }
    build(src);
    char op; int a, b, d; string s;
    while (cin >> op)
        if (op == '+') {
            cin >> a >> b >> d; add_inclusive(a, b, d);
        } else if (op == 'q') {
            cin >> a >> b; cout << op_inclusive(a, b) << "\n";
        } else getline(cin, s);
}
```

Máximo na árvore de segmentos preguiçosa (continuado)

Entrada	Saída
5 1 2 0 0 0 + 4 5 3 # 1 2 0 3 3 q 1 1 q 1 2 q 1 3 q 2 3 q 2 4 + 1 2 1 # 2 3 0 3 3 q 1 3 q 1 4 q 3 3 + 2 3 2 # 2 5 2 3 3 q 1 3 q 1 5	1 2 2 2 3 3 3 0 5 5

Aplicação da árvore de segmentos preguiçosa iterativa

Código `lstitapply.h`

```
ll apply(int ti, dlta d, int sz) {
    if (d.set != -1) {
        t[ti] = ll(d.set) * FACTOR(sz);
        delta[ti] = { 0, d.set };
    }
    if (d.add != 0) {
        t[ti] += ll(d.add) * FACTOR(sz);
        delta[ti].add += d.add;
    }
    return t[ti];
}
```

Empurra/Puxa da árvore de segmentos preguiçosa iterativa

Código `lstitpushpull.h`

```
void pull(int i) {
    for (int s = __builtin_ctz(i)+1; s < L; s++) {
        int ti = i >> s;
        t[ti] = OP(t[2*ti], t[2*ti+1]);
    }
}
void push(int i) {
    int sz = 1 << (L-1);
    for (int s = L; s > 0; s--, sz /= 2) {
        int ti = i >> s;
        apply(2*ti, delta[ti], sz);
        apply(2*ti+1, delta[ti], sz);
        delta[ti] = {};
    }
}
```

Aplicação da árvore de segmentos preguiçosa iterativa

Código `lstitapplyinc.h`

```
void apply_inclusive(int l, int r,
        char op = '\0', ll x = 0) {
    r++;
    dlta d;
    if (op == '+') { d.add = x; }
    if (op == '=') { d.set = x; }
    int tl = l += N, tr = r += N, sz = 1;
    push(tl); push(tr);
    for (; l < r; l /= 2, r /= 2, sz *= 2) {
        if (l & 1) { apply(l++, d, sz); }
        if (r & 1) { apply(--r, d, sz); }
    }
    pull(tl); pull(tr);
}
void add_inclusive(int l, int r, ll d) {
    apply_inclusive(l, r, '+', d);
}
```

Árvore de segmentos preguiçosa iterativa

Código `lstit.h`

```
const int L = ceil(log2(N));
struct dlta { int add = 0, set = -1; };
vector<ll> t (2*N); vector<dlta> delta (2*N);
#include "lstitapply.h"
#include "lstitpushpull.h"
#include "lstitapplyinc.h"
ll op_inclusive(int l, int r) {
    r++;
    int tl = l += N, tr = r += N, sz = 1;
    push(tl); push(tr);
    ll ans = NEUTRAL;
    for (; l < r; l /= 2, r /= 2, sz *= 2) {
        if (l & 1) { ans = OP(ans, apply(l++, dlta(), sz)); }
        if (r & 1) { ans = OP(ans, apply(--r, dlta(), sz)); }
    }
    pull(tl); pull(tr);
    return ans;
}
```

Construção da árvore de segmentos preguiçosa iterativa

Código `lstitbuild.h`

```
void build(vector<int>& src) {
    for (int i = 1; i < src.size(); i++)
        t[N+i] = src[i];
    for (int ti = N-1; ti > 0; ti--)
        t[ti] = OP(t[2*ti], t[2*ti+1]);
}
```

Máximo na árvore de segmentos preguiçosa iterativa

Código `lstit.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 2e5+15;
#define NEUTRAL 0
#define FACTOR(sz) 1
#define OP(X, Y) max(X, Y)
#include "lstit.h"
#include "lstitbuild.h"
int main() {
    int n; cin >> n; vector<int> src (n+1);
    for (int i = 1; i <= n; i++) { cin >> src[i]; }
    build(src);
    char op; int a, b, d; string s;
    while (cin >> op)
        if (op == '+') {
            cin >> a >> b >> d; add_inclusive(a, b, d);
        } else if (op == 'q') {
            cin >> a >> b; cout << op_inclusive(a, b) << "\n";
        } else getline(cin, s);
}
```

Máximo na árvore de segmentos preguiçosa iterativa (continuado)

Entrada	Saída
5 1 2 0 0 0	1
+ 4 5 3	2
# 1 2 0 3 3	2
q 1 1	2
q 1 2	3
q 1 3	3
q 2 3	3
q 2 4	0
+ 1 2 1	5
# 2 3 0 3 3	5
q 1 3	
q 1 4	
q 3 3	
+ 2 3 2	
# 2 5 2 3 3	
q 1 3	
q 1 5	

Somas na árvore de segmentos preguiçosa

- Para fazer somas, é necessário mudar o `FACTOR(sz)` para `sz` para somar corretamente no nó de forma preguiçosa.
- Isso significa que em nós colocamos o valor certo em nós mais acima onde atualizar a sua soma significa somar multiplicando pelo número de nós que aquele nó representa.
- No caso do mínimo e do máximo isso não era necessário pois o máximo e o mínimo não acumulavam, apenas pegavam um dos valores.

Soma na árvore de segmentos preguiçosa

Código `lstsum.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 1e5+15;
#define NEUTRAL 0
#define FACTOR(sz) (sz)
#define OP(X, Y) (X + Y)
#include "lst.h"
#include "lstbuild.h"
int main() {
    int n; cin >> n; vector<int> src (n+1);
    for (int i = 1; i <= n; i++) { cin >> src[i]; }
    build(src);
    char op; int a, b, d; string s;
    while (cin >> op)
        if (op == '+') {
            cin >> a >> b >> d; add_inclusive(a, b, d);
        } else if (op == 'q') {
            cin >> a >> b; cout << op_inclusive(a, b) << "\n";
        } else getline(cin, s);
}
```

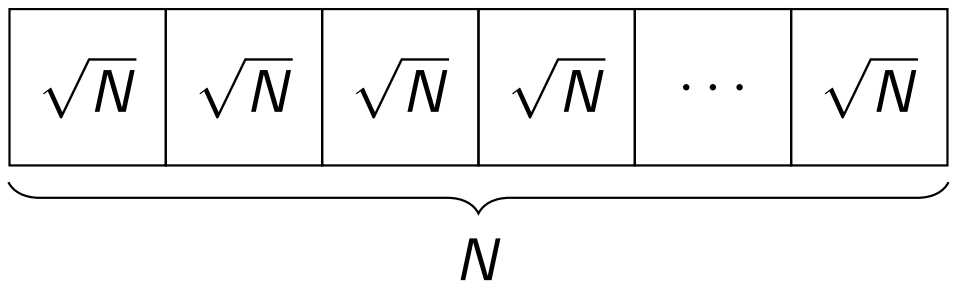
Soma na árvore de segmentos preguiçosa (continuado)

Entrada	Saída
5 1 2 0 0 0 + 4 5 3 # 1 2 0 3 3 q 1 1 q 1 2 q 1 3 q 2 3 q 2 4 + 1 2 1 # 2 3 0 3 3 q 1 3 q 1 4 q 3 3 + 2 3 2 # 2 5 2 3 3 q 1 3 q 1 5	1 3 3 2 5 5 8 0 9 15

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`

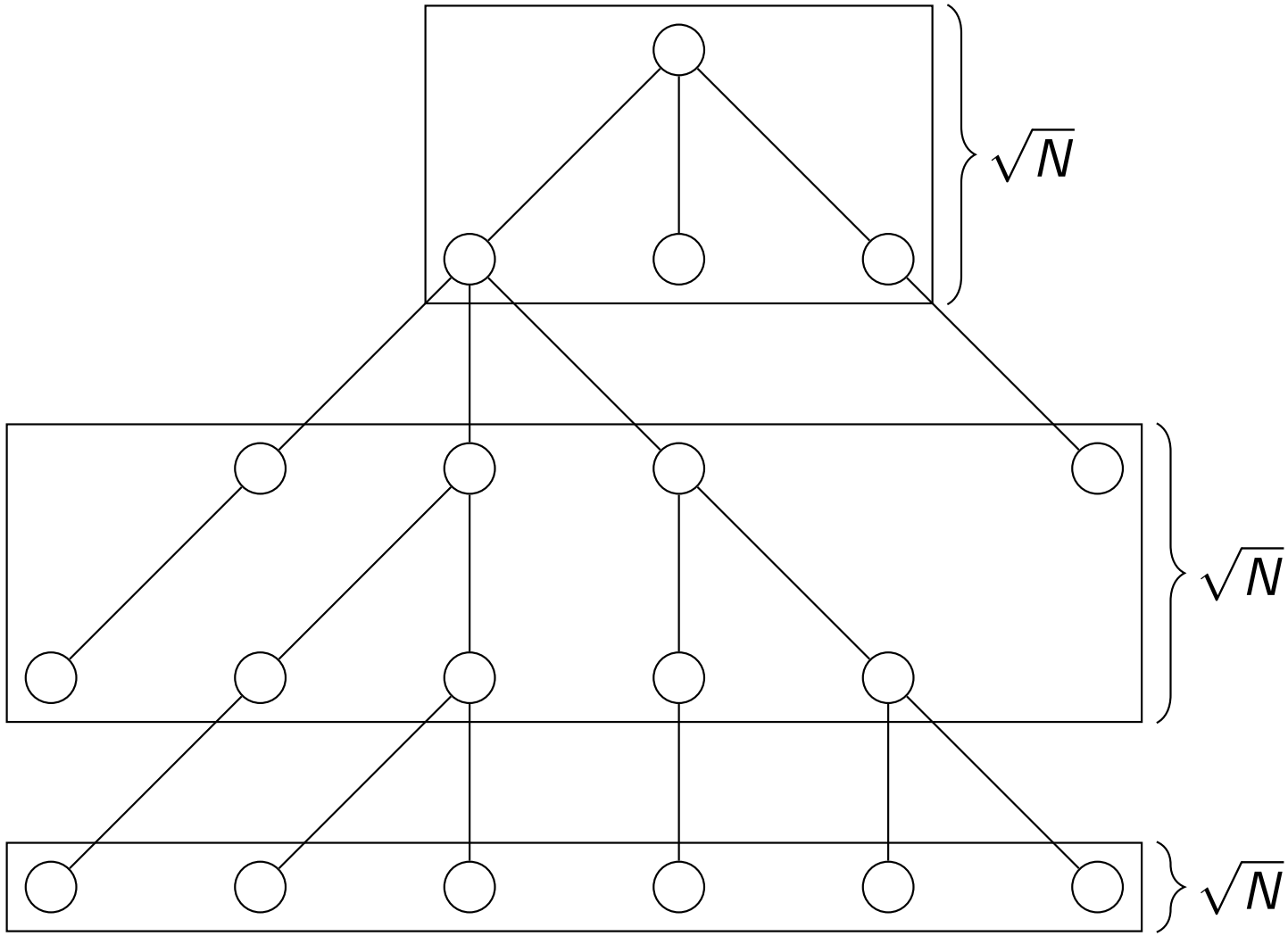
```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15, B = sqrt(ceil(N));
struct qry { int l, r, ix; };
#include "mode.h"
#include "mo.h"
int main() {
    int n, q; cin >> n >> q;
    for (int i = 0; i < n; i++) { cin >> v[i]; }
    vector<qry> qs;
    for (int i = 0; i < q; i++) {
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

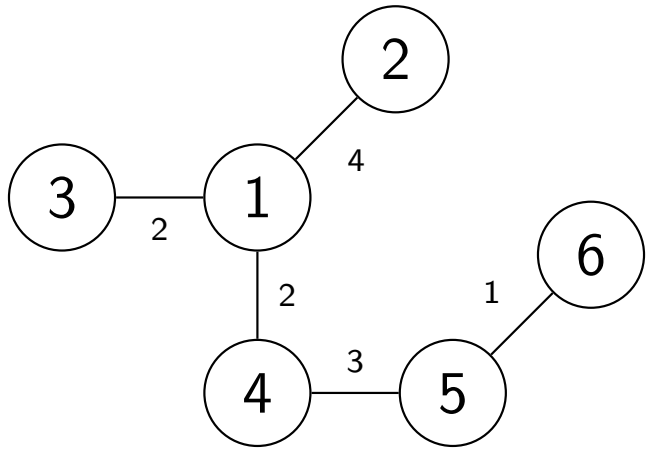
```
int stdt_op(int a, int b) {
    int ans = NEUTRAL;
    if (!(depth[a]/B > depth[b]/B)) { swap(a, b); }
    while (depth[a]/B > depth[b]/B) {
        ans = OP(ans, bweiop[a]); a = bup[a]; }
    if (!(depth[a] > depth[b])) { swap(a, b); }
    while (depth[a] > depth[b]) {
        ans = OP(ans, weiop[a]);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

Decomposição em Cadeias

Decomposição em cadeias

Decomposição em Cadeias (ou Decomposição Pesado-Leve) é uma técnica que resolve muitos problemas de consulta em árvore.

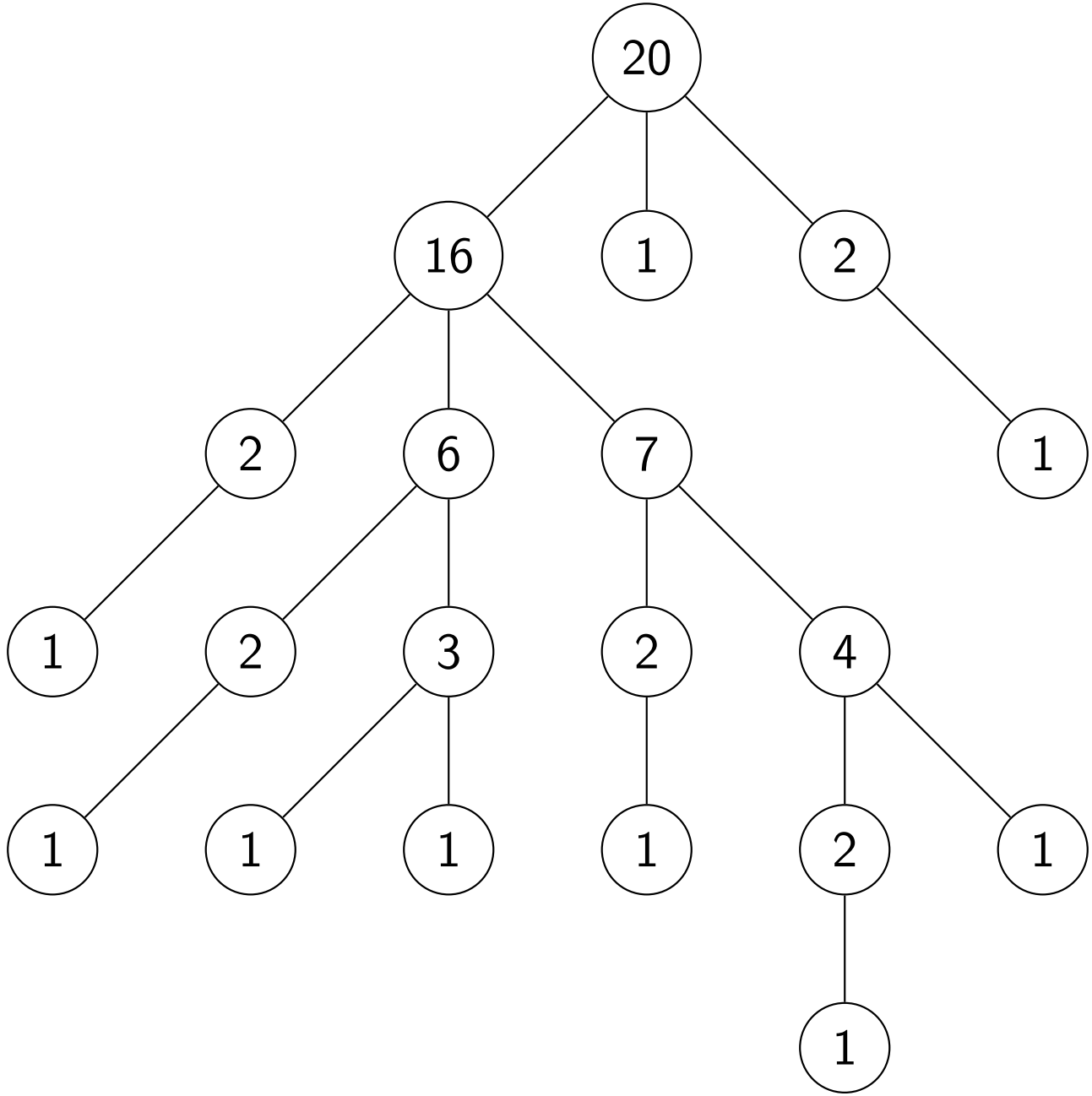
- Dividir a árvore inteira em várias cadeias disjuntas de maneira que qualquer nó pode chegar na raiz atravessando $\mathcal{O}(\lg N)$ cadeias.
- Consultas do tipo "soma no caminho de a até b" viram "soma de todas as cadeias no caminho de a até b".
- Que cadeias disjuntas são essas?

Arestas pesadas e leves

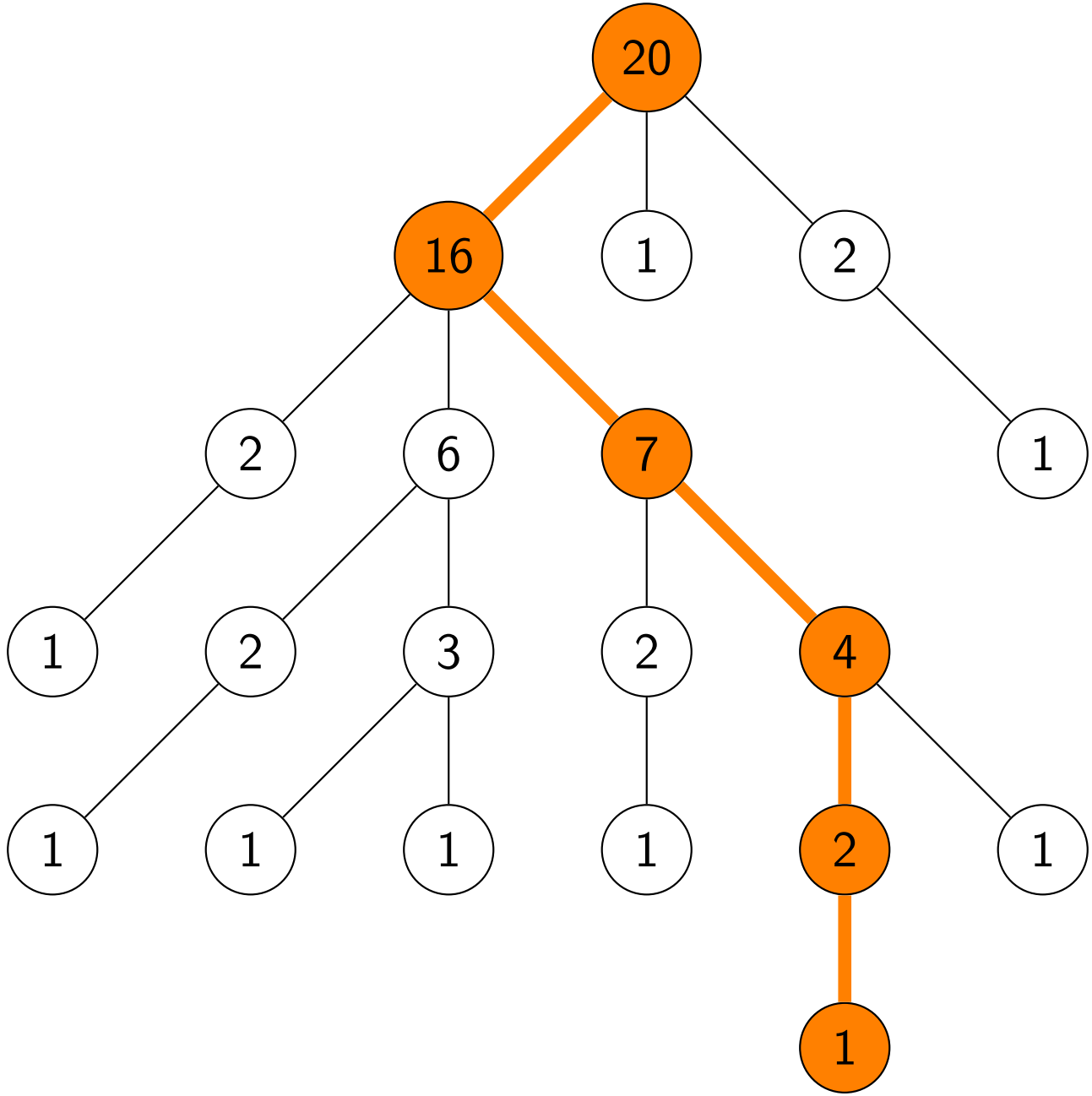
- Seja $S(u)$ a quantidade de vértices na subárvore do vértice u (incluindo u).
- Chamaremos a aresta (u, v) de **pesada** se $S(v)$ for maior que todas as outras arestas. Caso múltiplas satisfaçam esse requisito, escolha uma.
- Todas as outras arestas (u, v) serão ditas **leves**.
- Indo da raiz até qualquer vértice, passamos por no máximo $\mathcal{O}(\lg N)$ arestas leves.
- Isto porque andar para uma aresta leve reduz o tamanho da subárvore que vamos considerar **pela metade**:

$$S(v) < \frac{S(u)}{2}$$

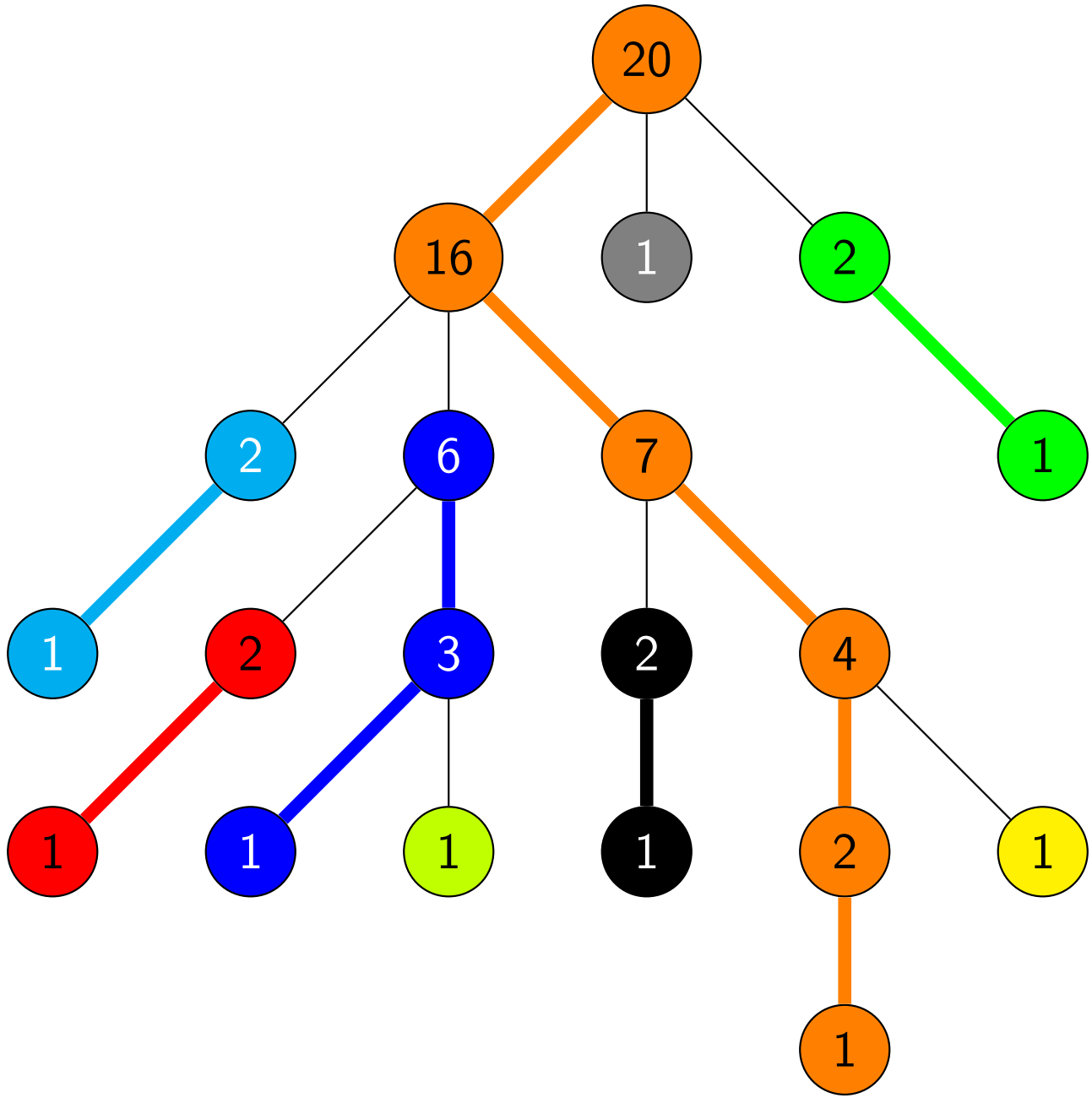
As cadeias pesadas



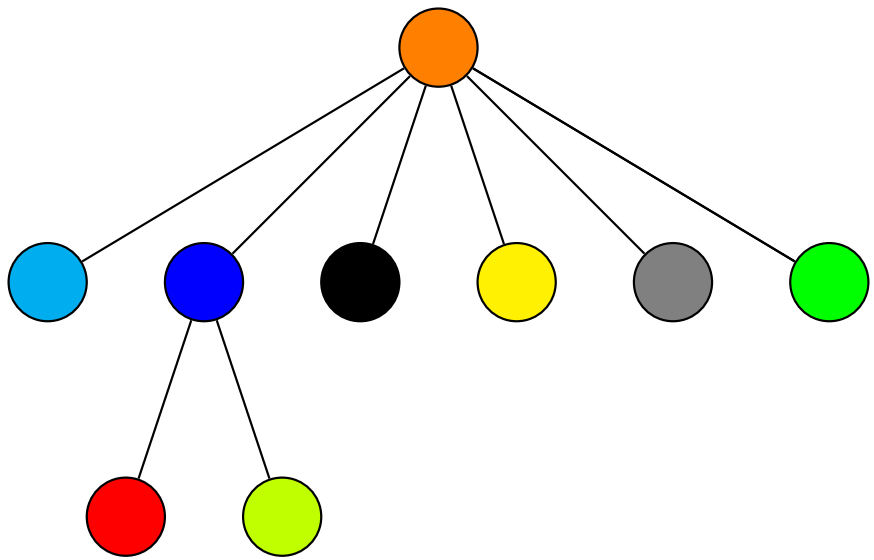
As cadeias pesadas



As cadeias pesadas



As cadeias pesadas (comprimidas)



Obtendo as cadeias pesadas

- Primeiro preencheremos o $S(u)$ de cada vértice. Faremos isso usando uma busca em profundidade.
- Nessa busca já podemos estabelecer para cada vértice a aresta que é pesada, então vamos guardar qual o filho que é pesado para cada u.
- Em seguida, faremos outra busca em profundidade, agora montando as cadeias. Manteremos um vetor que diz qual o elemento cabeça de cada cadeia, e continuaremos a cadeia, montando uma sequência de índices referente a aquela cadeia.
- Depois de terminarmos uma cadeia, chamaremos a função recursivamente para os filhos leves criarem suas próprias cadeias.

Preenchendo os filhos pesados

Código `hldfill.h`

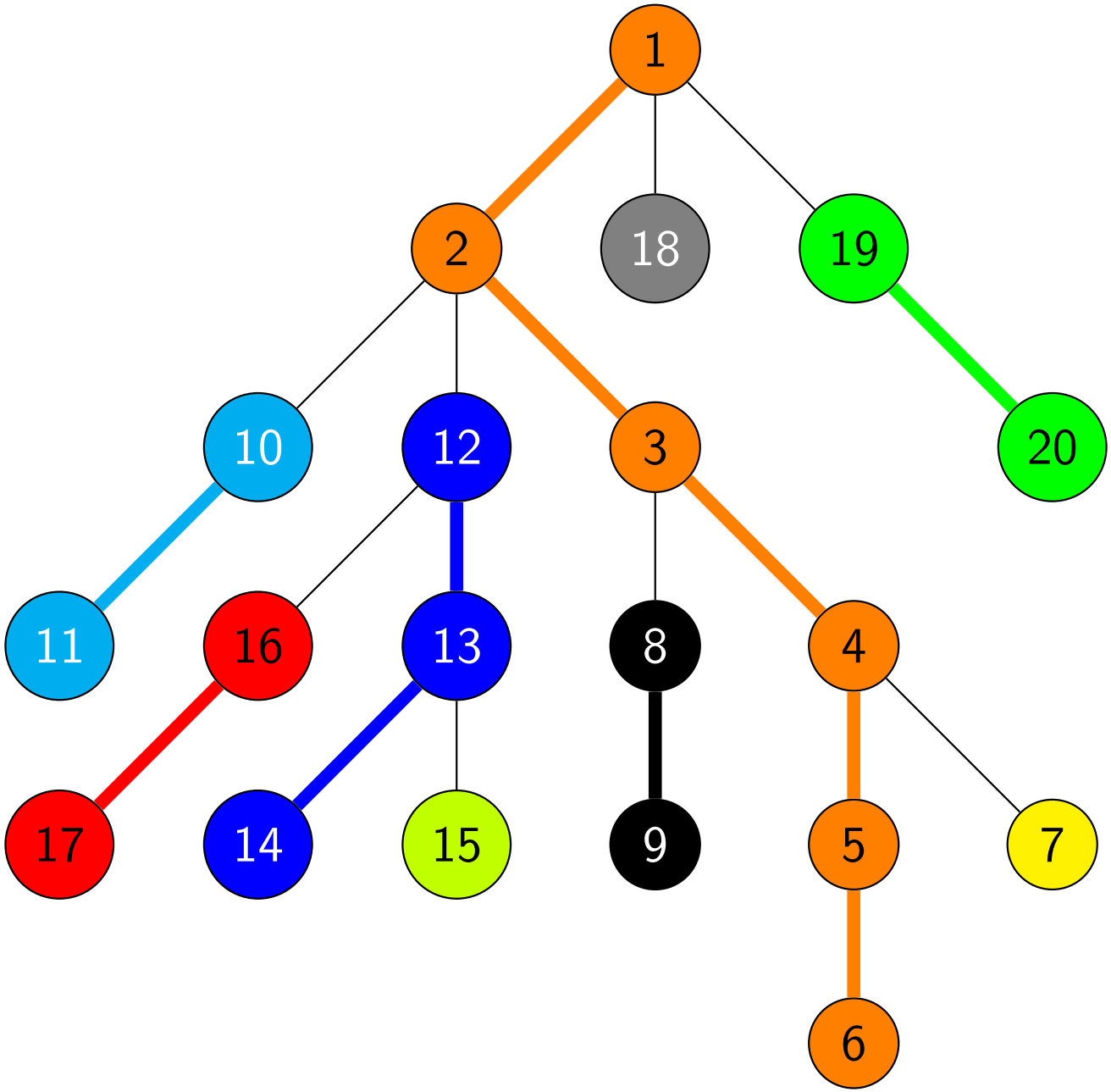
```
vector<int> heavy (N, -1), par (N, -1);
vector<int> depth (N), wei (N);
int hld_fill /* O(V + E) */ (int u, int w) {
    int s = 1, maxs = 0;
    wei[u] = w;
    for (auto [v, w] : g[u]) if (v != par[u]) {
        par[v] = u;
        depth[v] = depth[u] + 1;
        int cs = hld_fill(v, w);
        s += cs;
        if (cs > maxs) {
            maxs = cs;
            heavy[u] = v;
        }
    }
    return s;
}
```

Montando as cadeias

Código `hld.h`

```
vector<int> hds (N), ixs (N), origin (N);
int cix = 1;
void hld /* O(n lg n) */ (int u, int h) {
    hds[u] = h;
    origin[cix] = wei[u];
    ixs[u] = cix++;
    if (heavy[u] != -1)
        hld(heavy[u], h); // continue chain
    for (auto [v, w] : g[u])
        if (v != par[u] && v != heavy[u])
            hld(v, v); // new chain
}
```

Os índices das cadeias pesadas



E agora?

- Temos esse vetor colorido:

$$\left[1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\right]$$

Algumas propriedades:

- A subárvore de um vértice está em um intervalo contíguo.
- Caminhos na árvore são compostos de $\mathcal{O}(\lg N)$ intervalos.

Qual o próximo passo então? Usar uma estrutura de dados que trabalha bem com intervalos.

- Vamos usar a Árvore de Segmentos Preguiçosa.
- Então atualização e consulta em intervalos em $\mathcal{O}(\lg N)$.
- Operações em caminho acabam com complexidade $\mathcal{O}(\lg^2 N)$.
- Note que se você pré-computar a soma total dos intervalos e não atualizá-los, a complexidade pode ser de $\mathcal{O}(\lg N)$.

Fazendo operações nas cadeias

Código `hldop.h`

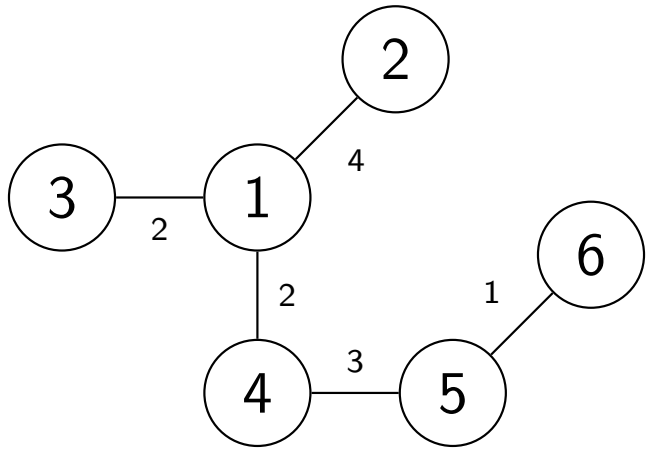
```
// O(lg^2 n)
ll hld_op(int u, int v) {
    ll ans = 0;
    for (; hds[u] != hds[v]; v = par[hds[v]]) {
        if (depth[hds[u]] > depth[hds[v]]) { swap(u, v); }
        ans = OP(ans,
            op_inclusive(ixs[hds[v]], ixs[v]));
    }
    if (depth[u] > depth[v]) { swap(u, v); }
    // Remove +1 if values are associated with vertices
    return OP(ans, op_inclusive(ixs[u] + 1, ixs[v]));
}
```

Maior peso entre dois vértices de uma árvore

Código `hldmax.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using ii = pair<int, int>;
const int N = 1e5+15; vector<vector<ii>> g (N);
#define NEUTRAL 0
#define OP(X, Y) max(X, Y)
#define FACTOR 1
#include "lst.h"
#include "lstbuild.h"
#include "hldfill.h"
#include "hld.h"
#include "hldop.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)); }
    hld_fill(0, 0); hld(0, 0); build(origin);
    while (q--) {
        int u, v; cin >> u >> v; u--; v--;
        cout << hld_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.
- Existem versões mais avançadas da Árvore de Segmentos! Variações populares incluem:
  - Árvore de Segmentos Beats (do anime "Angel Beats").
  - Árvore de Segmentos Persistente.
- Bons estudos!