ESCOLA DE PRIMAVERA DA MARATONA SBC DE PROGRAMAÇÃO

PROMOÇÃO:

CEFET-MG

UFMG

UFV

UNIFEI

USP Universidade de São Paulo

UFU

APOIO:

beecrowd

SBC Sociedade Brasileira de Computação

Grupo de Computação Competitiva

MENOR ANCESTRAL COMUM

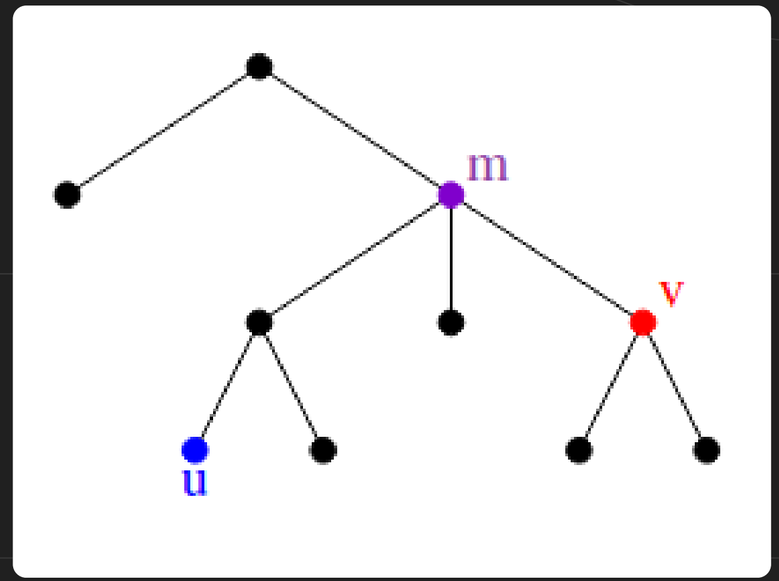
Por: *Eloy Ribeiro Maciel*

CONTEÚDOS

01 - CONCEITO

Dado um vértice ***v*** de uma árvore ***T*** de raiz ***r***, existe um único caminho conectando ***r*** e ***v***. Qualquer vértice contido nesse caminho é considerado ancestral de ***v***. Dado dois vértices ***u*** e ***v*** de ***T***, o **menor ancestral comum (Lowest Common Ancestor - LCA)** desses vértices é o vértice mais distante da raiz e que pertence ao conjunto de ancestrais de ***u*** e ***v***.

No exemplo ao lado ***m*** é o menor ancestral comum de ***u*** e ***v***.



02 - APLICAÇÕES

Uma das principais utilidades de descobrir o LCA de dois vértices está calcular a menor distância entre esses dois vértices. Isso ocorre porque só precisamos saber as profundidades de ***u***, ***v*** e do ***lca(u, v)***. Com esses dados é possível descobrir a distância entre ***u*** e ***v*** em O(1).

Inclusive, a fórmula para calcular a distância tendo o lca(u, v) é:

$$\text{profundidade}(u) + \text{profundidade}(v) - 2 \cdot lca(u, v)$$

Outras possíveis aplicações estão aliadas ao uso de Árvore de Sufixos e são:

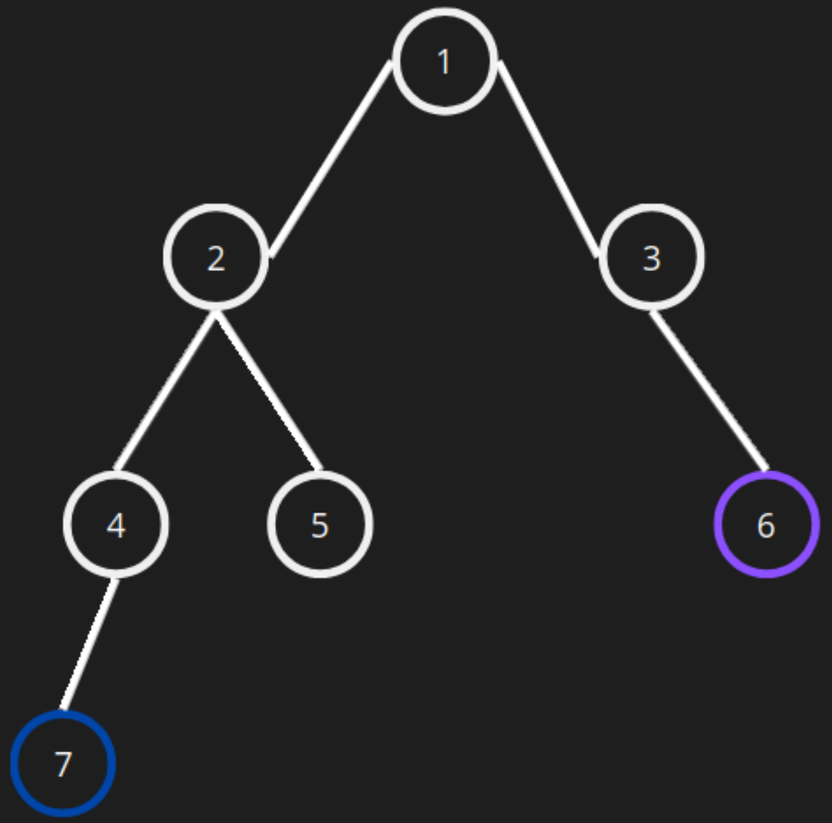
- Encontrar palíndromos maximais/repetições em um texto;
- buscas de padrões em textos admitindo erros em um ou nos dois textos comparados;
- Encontrar textos em que um padrão aparece ao menos uma vez.

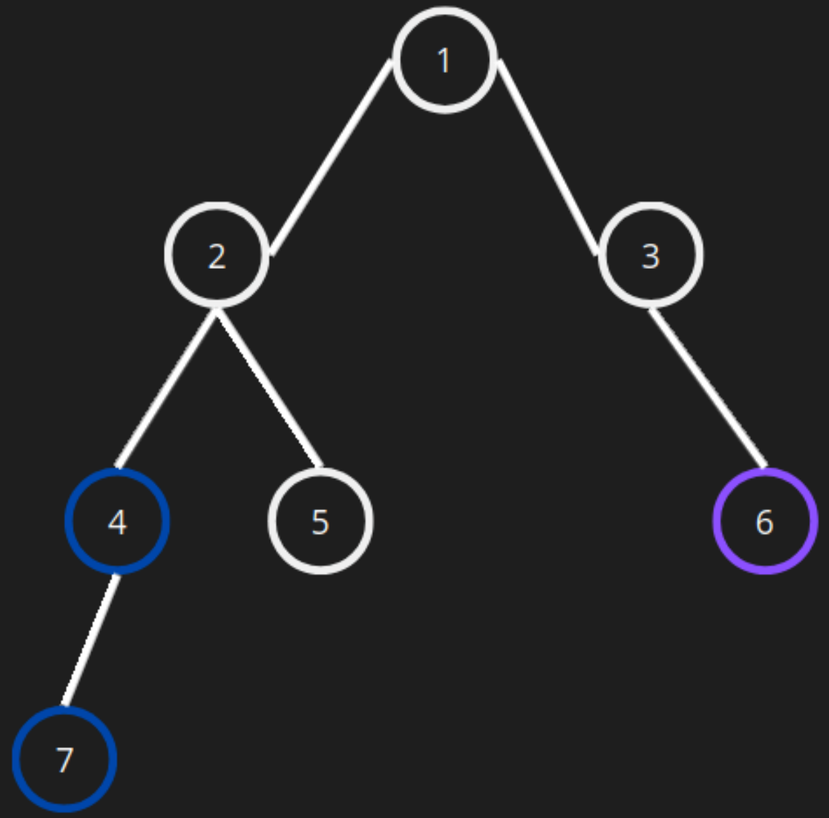
03 - SOLUÇÃO INGÊNUA

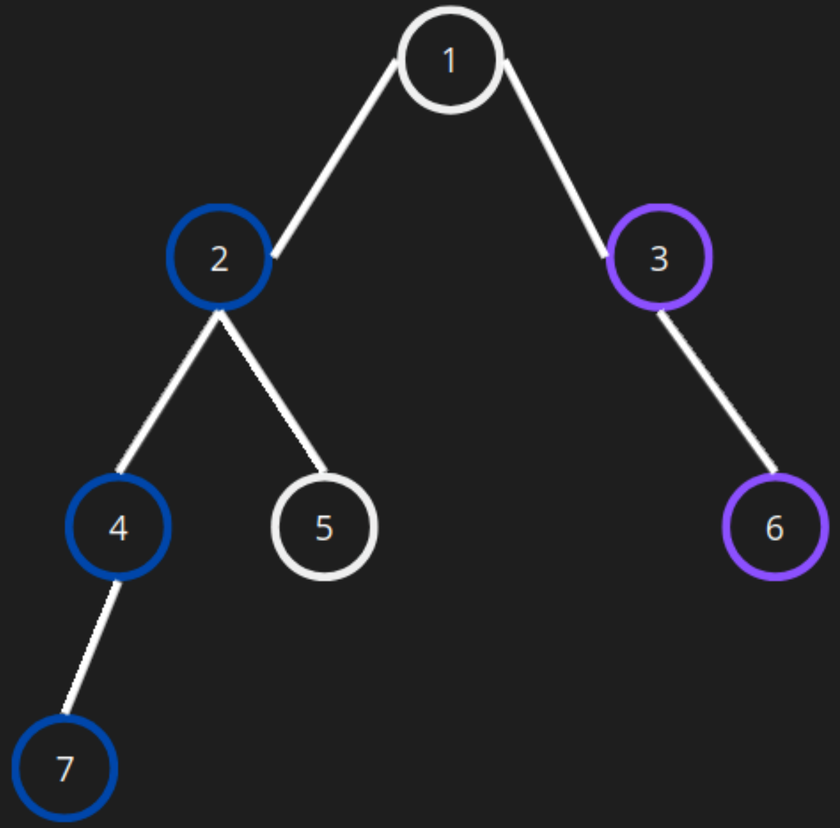
A primeira solução é passar pelos ancestrais de ambos os vértices até encontrarmos algum vértice já visitado, ou seja, um vértice comum.

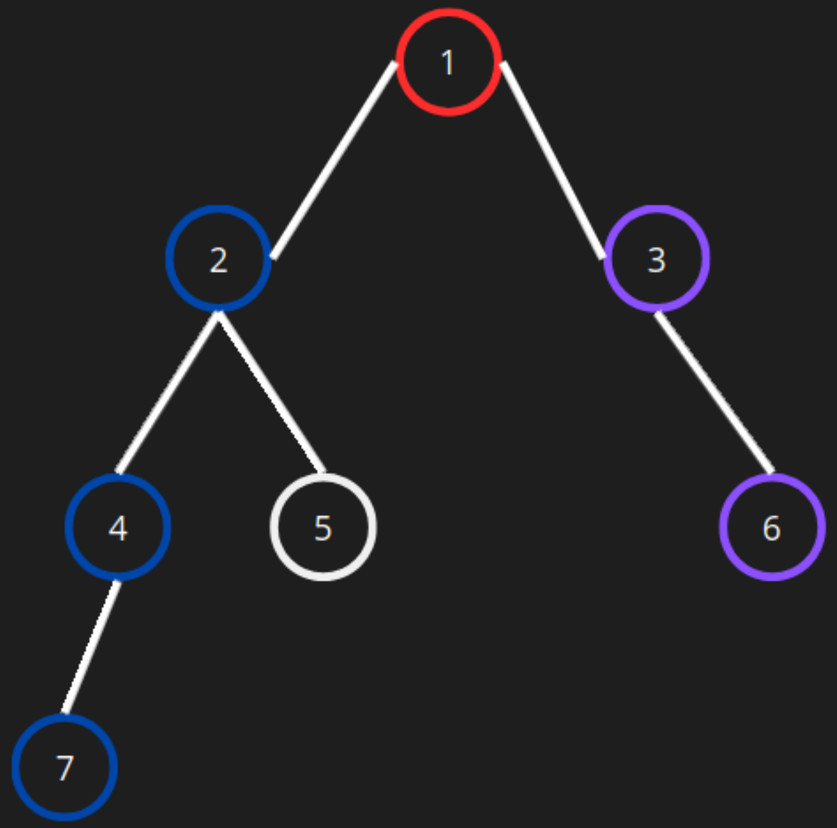
Para isso, é necessário um pré-processamento que registra a profundidade e o pai de cada vértice, esse pré-processamento tem uma complexidade O(n), algo que será similar para todas as soluções.

Já as consultas também possuem complexidade O(n), algo que não é bom. A ideia do algoritmo é sempre passar para o pai do vértice mais baixo ou de ambos os vértices até que se encontre um vértice comum.


1
2
3
4
5
6
7


1
2
3
4
5
6
7


1
2
3
4
5
6
7


1
2
3
4
5
6
7

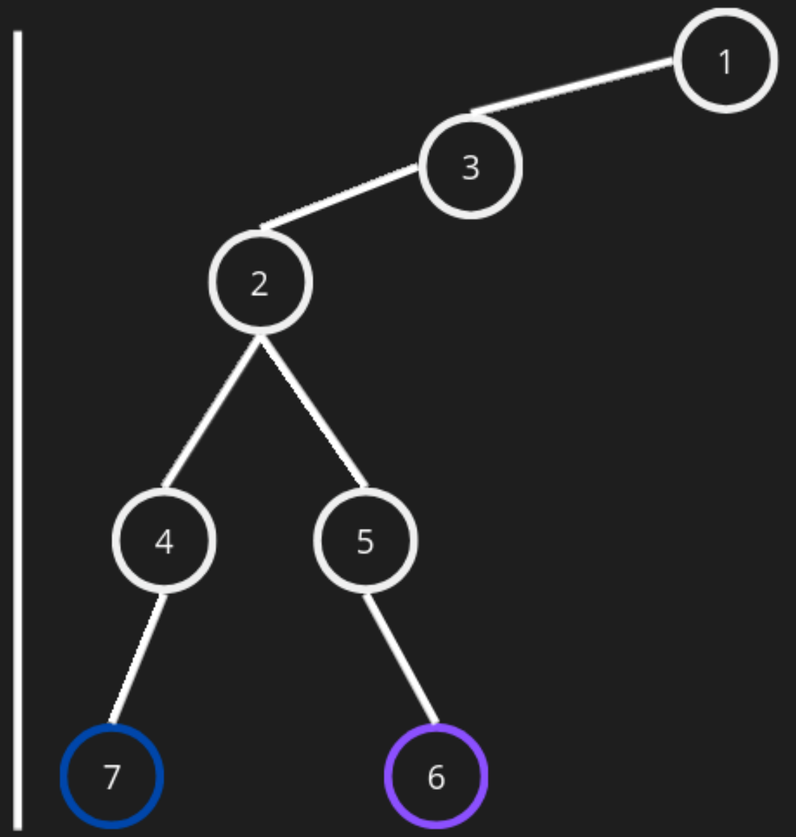
ALGORITMO INGÊNUO

```cpp
void dfs(int v, int d) {
    vis[v] = 1;
    depth[v] = d;

    for (int u : adj[v]) if (!vis[u]) {
        parent[u] = v;
        dfs(u, d + 1);
    }
}

int lca(int v, int u) {
    if(depth[u] < depth[v]) swap(u, v);

    while(depth[v] < depth[u]) v = parent[v];
    while(v != u) {
        v = parent[v];
        u = parent[u];
    }

    return v;
}
```

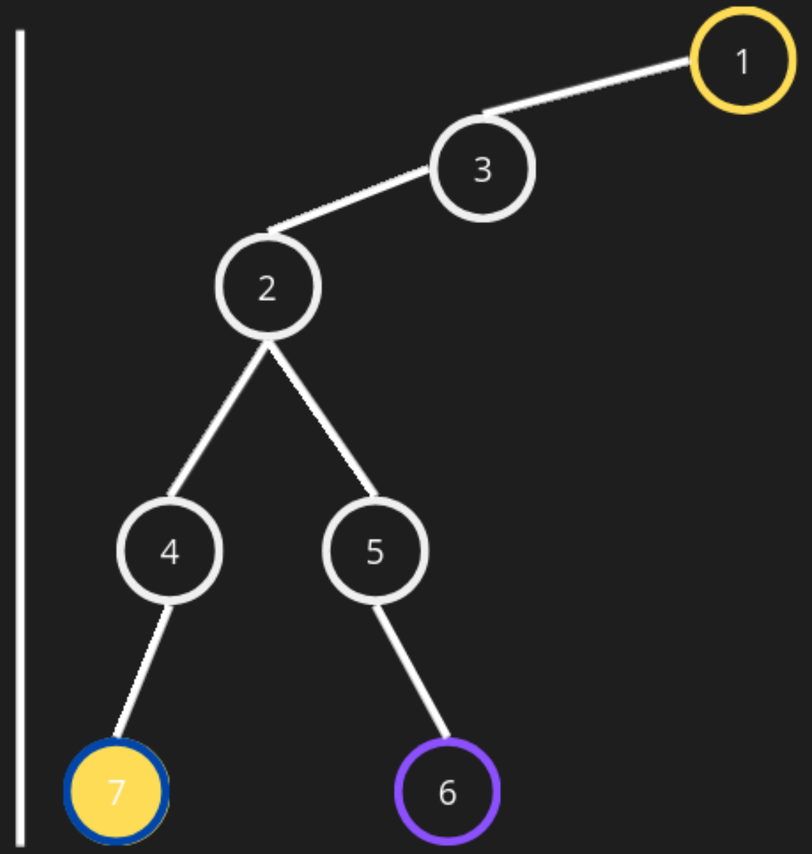
04 - ESCALADA BINÁRIA

A solução da escalada binária é semelhante à uma busca binária. Dado todos os antecessores de dois vértices v e u, queremos encontrar, primeiro, o vértice que é o último ancestral não comum, ou o ancestral não comum mais próximo da raiz. Sabendo esse vértice, teremos que o pai dele é o ancestral comum mais baixo.

Qual o lca de 7 e 8?

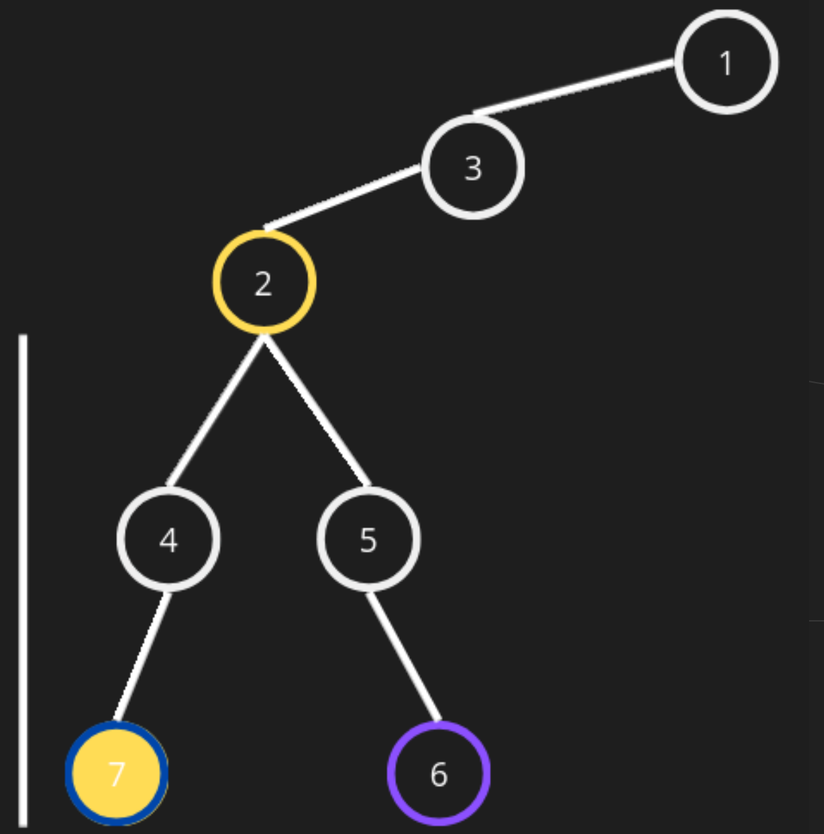


Esquerda = 1 (menor profundidade)
Direita = 7



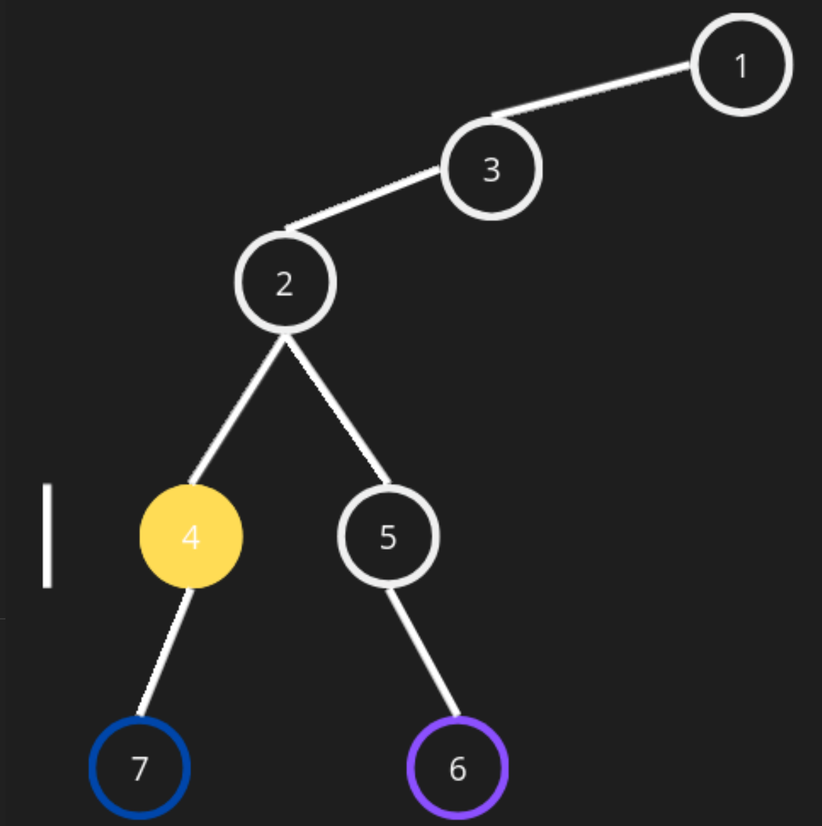
Esquerda = 1 (menor profundidade)
Meio = 2
Direita = 7

2 será a nova esquerda



Esquerda = 2 (menor profundidade)
Meio = 4
Direita = 7

4 é o último ancestral não comum e o pai dele é a resposta do lca dos vértices 7 e 8, ou seja, é o vértice 2

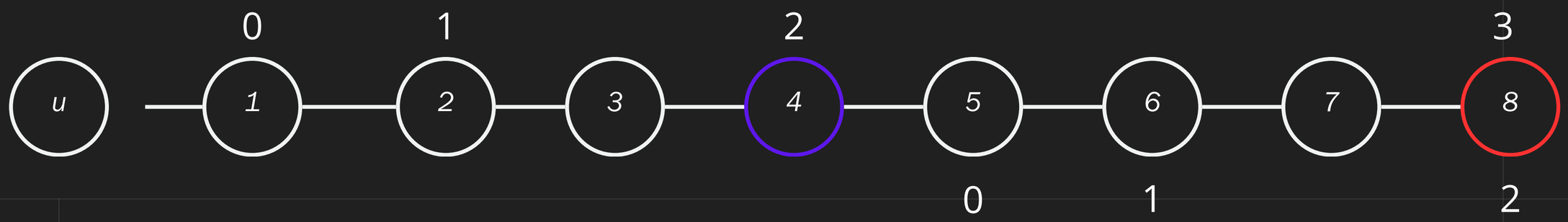


Todavia, salvar todos os ancestrais de todos os vértices é muito caro em termos de memória, já percorrer todos eles é muito caro em termos de tempo. Para contornar isso usamos Programação Dinâmica para construir uma tabela onde salvaremos apenas os antecessores em distâncias logaritmicas de cada vértice.

A relação de construção da tabela é a seguinte:

$$pai(u, 0) = p[u]$$
$$pai(u, k) = pai(pai(u, k-1), k-1)$$

Abaixo mostro como o 8° pai de u é o 4° pai do 4° pai de u



Na construção da árvore é necessário apenas implementar essa pd em complemento à dfs. Feito o pré-processamento, cada consulta será semelhante a solução ingênua, mas tomada em intervalos logarítmicos. Todavia buscamos o último ancestral não comum de ambos os vértices, após isso retornamos o pai desse vértice encontrado.

O pré-processamento é feito em O(nlog n) já o processamento é feito em O(log n), algo bem menor que a solução ingênua considerando que temos apenas um pré-processamento para várias consultas.

ALGORITMO DA ESCALADA

```cpp
class EscaladaBinaria {
private:
    int n, log_max;
    vector<int> depth, vis;
    vector<vector<int>> adj_list, parents;

    void dfs(int v, int p) {
        parents[v][0] = p;
        vis[v] = 1;

        for(int k {1}; k <= log_max; k++)
            parents[v][k] = parents[parents[v][k-1]][k-1];

        for (int u : adj_list[v]) if (!vis[u]) {
            depth[u] = depth[v] + 1;
            dfs(u, v);
        }
    }
```

```cpp
public:
    EscaladaBinaria(vector<vector<int>> _adj_list, int root) {
        n = _adj_list.size();
        log_max = log2(n);
        adj_list = _adj_list;
        vis = vector<int>(n, 0);
        depth = vector<int>(n, 0);

        parents = vector<vector<int>>(n, vector<int>(log_max + 1, -1));

        dfs(root, root);
    }

    int lca(int v, int u) {
        if(depth[u] < depth[v]) swap(u, v);

        for (int i {log_max}; i >= 0; i--) {
            if ((depth[v] - (1 << i)) >= depth[u])
                v = parents[v][i];
        }

        if (u == v) return v;

        for (int i {log_max}; i >= 0; i--) {
            if (parents[v][i] != parents[u][i]) {
                v = parents[v][i];
                u = parents[u][i];

            }
        }

        return parents[v][0];
    }
};
```

05 - REDUÇÃO PARA RMQ

Uma estratégia interessante para lidar com o problema é transforma-lo em um problema menor (reduzi-lo) e com soluções melhores. Seguindo essa ideia, é possível transformar uma consulta de um lca em uma consulta de valor minimo em um intervalo (Range Minimum Query - RMQ), problema com solução possivel mais rápida.

O problema de RMQ busca, em um vetor de valores, o valor mínimo entre dois índices válidos. Há algumas estratégias para resolver esse problema, dentre elas está usar uma Árvore de Segmentos e usar Sparse Table, sendo que essa segunda permite resolver em O(1).

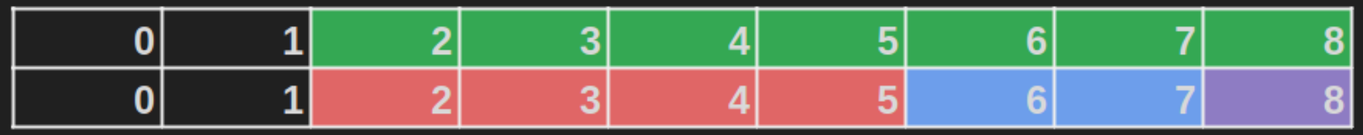
06 - SPARSE TABLE

Temos que qualquer número inteiro não negativo pode ser representado pela soma de potências decrescentes de 2 (base binária).

- Ex: 13 = 1101 (2) = 8+4+1

E podemos representar qualquer intervalo como a união de intervalos menores com comprimento em potências de 2.

- Intervalo [2,8] = [2,5] U [6,7] U [8,8]
- |[2,5]| = 4
- |[6,7]| = 2
- |[8,8]| = 1



A ideia por trás de uma sparse table é pré-calcular os resultados de uma função aplicada a todos os intervalos com tamanho em potência de 2. Para intervalos distintos, consultamos dividindo em intervalos com valores já conhecidos. Como para um número n há no máximo log n potências de 2 somando, um intervalo de tamanho n será dividido no máximo em log n subintervalos.

Para implementar essa estrutura, usaremos uma tabela ***st*** onde a posição ***st[i][j]*** armazena o valor do intervalo ***[i, i + 2^(j-1)]***. Ai está a mágica dessa estrutura, para um vetor de tamanho ***n*** teremos uma tabela ***n*log n***, não uma com tamanho ***n*n***.

Para construção, usaremos a seguinte relação, onde a função f é a função que desejamos aplicar (soma, mínimo, máximo, etc):

$$st(i,j) = \begin{cases} f(v[i]) & \text{se } j = 0 \\ f(st(i, j-1), st(i + 2^{j-1}, j-1)) & c.c. \end{cases}$$

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

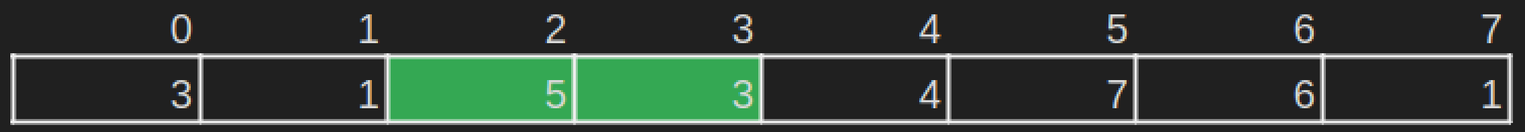
	0	1	2	3
0	3			
1	1			
2	5			
3	3			
4	4			
5	7			
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1			
2	5			
3	3			
4	4			
5	7			
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1	6		
2	5			
3	3			
4	4			
5	7			
6	6			
7	1			



	0	1	2	3
0	3	4		
1	1	6		
2	5	8		
3	3			
4	4			
5	7			
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1	6		
2	5	8		
3	3	7		
4	4			
5	7			
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1	6		
2	5	8		
3	3	7		
4	4	11		
5	7			
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1	6		
2	5	8		
3	3	7		
4	4	11		
5	7	13		
6	6			
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4		
1	1	6		
2	5	8		
3	3	7		
4	4	11		
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	
1	1	6		
2	5	8		
3	3	7		
4	4	11		
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	
1	1	6	13	
2	5	8		
3	3	7		
4	4	11		
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	
1	1	6	13	
2	5	8	19	
3	3	7		
4	4	11		
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	
1	1	6	13	
2	5	8	19	
3	3	7	20	
4	4	11		
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	
1	1	6	13	
2	5	8	19	
3	3	7	20	
4	4	11	18	
5	7	13		
6	6	7		
7	1			

0	1	2	3	4	5	6	7
3	1	5	3	4	7	6	1

	0	1	2	3
0	3	4	12	30
1	1	6	13	
2	5	8	19	
3	3	7	20	
4	4	11	18	
5	7	13		
6	6	7		
7	1			

Para realizar uma consulta no intervalo [L,R}, encontramos a primeira potência que tem um comprimento igual ou menor que o intervalo (começamos da maior possível), ao encontrar processamos a parte encontrada (aplicamos f com o valor salvo anteriormente e salvamos) e continuamos para o intervalo que restou.

Essa consulta apresentada é realizada em tempo O(log n), como você deve ter notado na explicação, mencionei a possibilidade de realizar ela em O(1). Isso é possível quando a função f não sofre influencia com as sobreposições de subsequências, ou seja, quando não precisamos ter conjuntos disjuntos. Esse é o caso de consultar um RMQ uma vez que a função minimo é idempotente (f(x,x)=x)

Nesses casos a relação da pesquisa é: $min(st[L][j], st[R - 2^j + 1][j])$

Onde $j = \lfloor log_2(R - L + 1) \rfloor$

```cpp
class SparseTable {
private:
    int n, k;
    vector<vector<int>> tab;
    vector<int> log;
    function<int(int, int)> f;

    void computar() {
        for(int j{1}; j <= k; j++) {
            for(int i{0}; i + (1 << j) <= n; i++)
                tab[i][j] = f(tab[i][j-1], tab[i + (1 << (j - 1))][j - 1]);
        }
    }

```

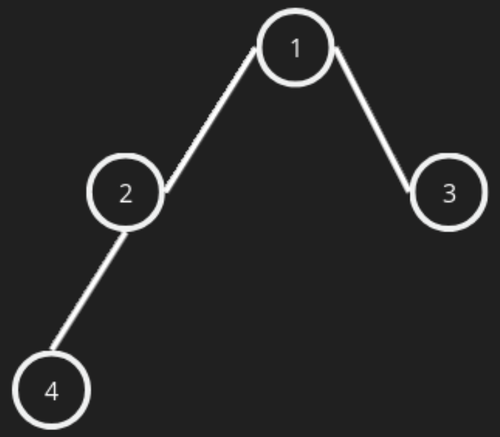
```cpp
public:
    SparseTable() {}

    SparseTable(vector<int> &arr, function<int(int, int)> _f) {
        n = arr.size();

        f = _f;

        log = vector<int>(n + 1);
        log[1] = 0;
        for(int i = 2; i < n + 1; ++i) log[i] = log[i / 2] + 1;

        k = log[n];

        tab = vector<vector<int>>(n, vector<int>(k + 1));
        for(int i = 0; i < n; ++i) tab[i][0] = arr[i];

        computar();
    }

    int consultar(int left, int right) {
        int j = log[right - left + 1];
        return f(tab[left][j], tab[right - (1 << j) + 1][j]);
    }
};
```

07 - ALGORIMO COM RMQ

Com a sparse table pronta, precisamos fazer um pré-processamento da árvore salva um tour de Euler, a profundidade de cada nó e primeira vez que passamos por ele.

Com isso note que para calcular o lca(v,u) entre dois vértices basta calcular o vértice de menor altura no tour de Euler que está entre a primeira visita de v e u. Para isso usamos uma sparse table com consultas em O(1).

Construir um tour de Euler é salvar em um vetor o vertice atual sempre, independente se está voltando para ele ou passando pela primeira vez.



Vértices	1	2	3	4
Profundidade	0	1	1	2
Primeira	0	1	5	2

	0	1	2	3	4	5	6
Tour de Euler	1	2	4	2	1	3	1

```cpp
class LCATree {
private:
    int n;
    vector<int> vis, depth, first, euler_tour;
    SparseTable sp;

    void dfs(int v, int d, vector<vector<int>> &adj) {
        vis[v] = 1;
        depth[v] = d;
        first[v] = euler_tour.size();
        euler_tour.push_back(v);

        for(auto u : adj[v]) {
            if(!vis[u]) {
                dfs(u, d + 1, adj);
                euler_tour.push_back(v);
            }
        }
    }

    int lca(int v, int u) {
        int left = first[v], right = first[u];

        if(left > right)
            swap(left, right);

        return sp.consultar(left, right);
    }

    int rmq(int v, int u) {
        int lca_v_u = lca(v, u);
        return depth[v] + depth[u] - 2 * depth[lca_v_u];
    }

```

```cpp
public:
    LCATree(vector<vector<int>> &g) {
        int n = g.size();
        vis = vector<int>(n, 0);
        depth = vector<int>(n);
        first = vector<int>(n);
        euler_tour = vector<int>();

        for(int i = 0; i < n; ++i) if(!vis[i]) dfs(i, 0, g);

        auto f = [&](int a, int b){
            if(depth[a] < depth[b]) return a;
            return b;
        };

        sp = SparseTable(euler_tour, f);
    }

    int query(int v, int u) {
        return rmq(v, u);
    }
};
```

08 - PROBLEMA

Agora que conhecemos o tema podemos praticar com problemas.
Para servir de exemplo, irei resolver o probelema 2470 do beecrowd.

- Problema: https://judge.beecrowd.com/pt/problems/view/2470.
- Link da solução: https://gist.github.com/El0y-C0SM0/1f099b8e91a71825a1ad2860b1425ea7

09 - FONTES

- O problema do Menor Ancestral Comum, USP, disponivel em: https://bccdev.ime.usp.br/tccs/2005/daniel/poster.pdf
- Menor Ancestral Comum (LCA - Lowest Common Ancestor), disponivel em: https://youtu.be/bs1ohR0Kdyw?si=SnF46VElXXXxjTpo
- Sparse Table, disponivel em: https://youtu.be/inAZoc5K9jo?si=lNS6-dx6R-MKyTYB

OBRIGADO PELA ATENÇÃO

Grupo de Computação Competitiva

GCC.CEFETMG

gcc-cefetmg.github.io/Site-GCC

@gcc.cefetmg