Graph Algorithms Book

NachiketPatil
RollNo : 2020101018

Index

No.	Topic	Page No.
1	$Topological\ Sorting$	1
	$\rightarrow Using\ DFS$	1
	$\rightarrow Kahn's\ algorithm$	3
2	$Dijkstra$	6
3	$Floyd - Warshall$	9
4	$Minimum\ Spanning\ Tree$	11
	$\rightarrow Prim's\ algorithm$	11
	$\rightarrow Kruskal's\ algorithm$	13
5	$Lowest\ Common\ Ancestor\ (using\ RMQ)$	16
6	$Lowest\ Common\ Ancestor\ (Binary\ Lifting)$	19

Topological Sorting

Topological Sorting for a Directed Acyclic Graph (*DAG*) is a linear ordering of vertices, such that for every edge from vertex *u* to *v*, *u* comes before *v*. It is a graph traversal in which every vertex *a* is visited only after all its dependencies are visited. For example, topological sorting may be used to represent an order of tasks to be performed, if every vertex represents a task and the edges represent a constraint that one task must be performed before another.

Using DFS:

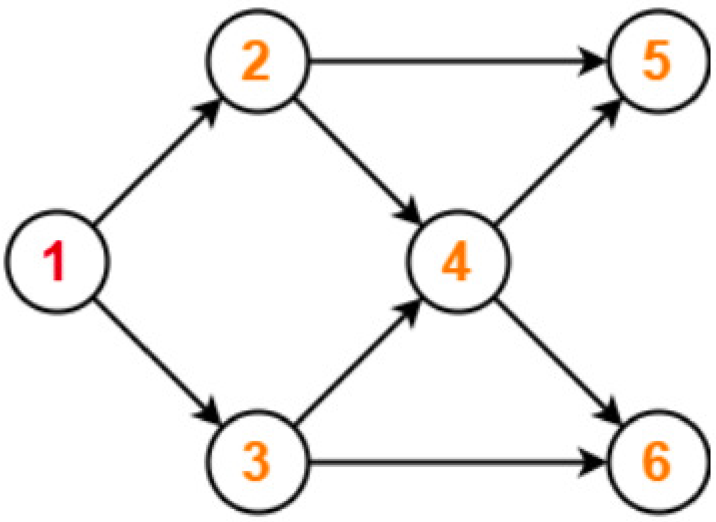
DFS is somewhat modified to get a topological ordering of the vertices. This algorithm is started from a vertex and then the method is recursively called for all its adjacent vertices. A temporary stack is used. For a vertex, after all its adjacent vertices, only then, it is pushed in the stack. Later, the stack is emptied by popping the vertices one by one, which is, ultimately, the topological sorting of the DAG. Consider an edge uv directed from u to v. If the vertex v occurs before u in the topological ordering, the graph is not an DAG.

Below is a implementation of the same in `topoSortUsingDFS.cpp`. It uses the `class Graph` as a graph from `graph.hpp` mentioned before.

```
// topoSortUsingDFS.cpp

#include "graph.hpp"

/**
 * @brief Helps topologicalSort by recursively applying DFS. Reversed topological
ordering
 * is stored in order.
 *
 * @param G The Graph Object.
 * @param u The vertex on which DFS is to be performed.
 * @param order The vector container in which reverse topological ordering is stored.
 * @param visited The vector boolean container to keep track of visited vertices.
 */
void
topoSortUtil(const Graph &G, Vertex u, std::vector<Vertex> &order, std::vector<bool>
&visited) {
    visited[u] = true;
    const std::vector<Vertex> & adj = G.getAdj(u);
    for (Vertex v : adj) {
        if (not visited[v])
            topoSortUtil(G, v, order, visited);
    }
    order.push_back(u);
}


/**
 * @brief Prints a topological ordering of the vertices if the Graph
 * is an DAG. Prints an error message if the Graph is not an DAG.
 *
 * @param G The Graph object
 * @param err Default = "IMPOSSIBLE". The error message to be prited
 * if the graph is not an DAG.
 */
void
topologicalSort(const Graph &G, const std::string& err = "IMPOSSIBLE") {
    int V = G.getV();
    int pos[V + 1]={};
    std::vector<Vertex> topologicalOrdering;
    topologicalOrdering.reserve(V);
    std::vector<bool> visited(V + 1);

    for (Vertex i = Vertex(1); i <= V; ++i) {
        if (not visited[i])
            topoSortUtil(G, i, topologicalOrdering, visited);
    }

    std::reverse(topologicalOrdering.begin(), topologicalOrdering.end());

    int p = 0;
    for (Vertex u : topologicalOrdering) {
        pos[u] = p; ++p;
    }

    for (Vertex i = Vertex(1); i <= V; ++i) {
        const std::vector<Vertex> &adj = G.getAdj(i);
        for (Vertex v : adj) {
            if (pos[v] < pos[i]) {
                std::cout << err << "\n";
                return;
            }
        }
    }

    for (int u : topologicalOrdering) {
        std::cout << u << ' ';
    }
    std::cout << "\n";
}
```

```
#include "graph.hpp"

void topologicalSort(const Graph &G, const std::string& err = "IMPOSSIBLE");

int main() {
    Graph G(6, true);
    G.addEdge(1, 2);
    G.addEdge(1, 3);
    G.addEdge(2, 4);
    G.addEdge(2, 5);
    G.addEdge(3, 4);
    G.addEdge(3, 6);
    G.addEdge(4, 5);
    G.addEdge(4, 6);
    topologicalSort(G);
    return 0;
}
```

The example graph used in the `main.cpp` file is:



After compiling and running,

```
$ g++ topoSortUsingDFS.cpp main.cpp -std=gnu++17
$ ./a.out
1 3 2 4 6 5
```

- Time Complexity: $O(V+E)$
- Space Complexity: $O(V)$

Since this algorithm is a modification of the DFS algorithm with an extra temporary stack, the time complexity is the same. It needs extra space for the stack.

Kahn's Algorithm:

Kahn's algorithm is based on including vertices with no incoming edges and eliminating vertices with no outgoing edges.

In this algorithm, a queue is maintained. Initially, the queue is empty. Also, a count of in-degree of all vertices is maintained. For every outgoing edge, the in-degree of the destination vertex is incremented.

Then, all vertices with no incoming edge or with in-degree equal to 0 is enqueued. Then, until the queue is not empty, following steps are performed:

- A vertex from the queue is dequeued.
- For every such vertex, the in-degrees' of its adjacent vertices are decremented.
- After decrementing, if the in-degree becomes equal to 0, the adjacent vertex is enqueued in the queue.

Below is a implementation of the same in `topoSortUsingKahn.cpp`. It uses the `class Graph` as a parameter from `graph.hpp` mentioned before.

```
// topoSortUsingKahn.cpp

#include "graph.hpp"

void
topologicalSort(const Graph &G, const std::string& err = "IMPOSSIBLE") {
    int V = G.getV();
    Vertex inDegree[V + 1]={};
    std::queue<Vertex> Q;
    std::vector<Vertex> topologicalOrdering;
    topologicalOrdering.reserve(V);

    // Count incoming edges for all vertices
    for (Vertex u = Vertex(1); u <= V; ++u) {
        const std::vector<Vertex> & adj = G.getAdj(u);
        for (Vertex v : adj) {
            inDegree[v]++;
        }
    }

    for (Vertex u = Vertex(1); u <= V; ++u) {
        // Enqueue the vertex with 0 in-degree
        if (inDegree[u] == 0)
            Q.push(u);
    }

    while (not Q.empty()) {
        Vertex u = Q.front();
        Q.pop();

        topologicalOrdering.push_back(u);

        const std::vector<Vertex> & adj = G.getAdj(u);
        for (Vertex v : adj) {

            // Decrement in-degree of adjacent vertex
            --inDegree[v];

            // Enqueue the vertex with 0 in-degree
            if (inDegree[v] == 0)
                Q.push(v);
        }
    }

    if (Vertex(topologicalOrdering.size()) != V) {
        std::cout << err << "\n";
        return;
    }

    for (int u : topologicalOrdering) {
        std::cout << u << ' ';
    }
    std::cout << "\n";
}
```

The same `main.cpp` is used and the same example diagram shown before in the section *Using DFS* of *Topological Sort*.

After compiling and running `main.cpp`,

```
$ g++ topoSortUsingKahn.cpp main.cpp -std=gnu++17
$ ./a.out
1 2 3 4 5 6
```

- Time Complexity: $O(V + E)$
- Space Complexity: $O(V)$

Dijkstra

Given a directed or undirected weighted graph with V vertices and E edges with no negative cycles, the problem is to find the lengths of the shortest paths from a starting vertex s to all other vertices, and find the shortest paths themselves. This problem is called *single-source shortest paths problem*.

We maintain an array $d[]$ where for every vertex u, $d[u]$ is the length of the shortest path from s to u. We initialize $d[u] = \infty \; \forall \; u \in V - \{s\}$ where ∞ is some large number. Also, $d[s] = 0$.

We also maintain an array $p[]$ which keeps track of immediate predecessor, i.e., $p[v]$ would store the vertex which is predecessor of v. This array would be helpful in restoring the path from source vertex s to any other vertex t.

The algorithm runs for almost V iterations.
At each iteration, an unmarked vertex u with least value $d[u]$ is extracted and marked.
From vertex u, relaxations are performed on all vertices adjacent to it. That is, for each adjacent vertex v, we minimize the value of $d[v]$. If the current edge length is len, then

$$d[v] = min(d[v], len + d[u])$$

On every relaxation we perform, we update the value of $p[v]$ as:

$$p[v] = u$$

The time for extracting an unmarked vertex u with least value $d[u]$ is naively $O(V)$ if we just loop through all vertices. The step of extraction is performed V times for every vertex. Since the time for relaxation of an edge is simply $O(1)$ and we perform the step of relaxation E times for every edge. The time complexity of this algorithm would be $O(V^2 + E)$.

We can use data structures such as AVL trees, binary heap, etc. to solve this problem of extracting minimum in $O(\log V)$. For example, we can make a compromise and use a binary heap for both operations of extraction (in $O(\log V)$) and updating the value after relaxation (in $O(\log V)$).
Then, the total complexity would be,
$O(V \log V + E \log V) = O((V + E) \log V) \approx O(E \log V)$.

Here is an implementation for the algorithm using the `graph.hpp` header introduced before. The file `dijkstra.hpp` has `struct Dijkstra` inherited publicly from `class weightedGraph` in `graph.hpp`.

```cpp
// dijkstra.hpp

#ifndef DIJKSTRA_HPP
#define DIJKSTRA_HPP

#include "graph.hpp"

struct Dijkstra : public weightedGraph {
    using ll = long long int;
    const ll INF = 0x3f3f3f3f3f3f3f3f;

    ll *d;
    Vertex *p;
    /**
     * @brief Constructs a new Dijkstra object.
     *
     * @param _V Number of vertices.
     * @param _directed Default = false. specify true if graph is directed.
     */
    Dijkstra(Vertex _V, bool _directed = false) : weightedGraph(_V, _directed) {
        d = new ll[V + 1]();
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from a weighted Graph object.
     *
     * @param G The weighted Graph Object to be copied.
     */
    Dijkstra(weightedGraph G) : weightedGraph(G) {
        d = new ll[V + 1];
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from an old one.
     * It calls the weightedGraph copy constructor.
     *
     * @param copy The old Dijkstra object to be copied.
     */
    Dijkstra(const Dijkstra &copy) : weightedGraph(copy) {
        memcpy(d, copy.d, sizeof(ll) * (V + 1));
    }


    /**
     * @brief Destroys the Dijkstra object.
     *
     */
    ~Dijkstra() {
        delete []d;
    }

    /**
     * @brief finds the lengths of shortest paths from the source vertex to all
     * vertices and stores in Dijkstra::d[].
     *
     * @param s Default = 1. The source vertex.
     */
    void solveShortestPaths(Vertex s = 1) {
        using length = std::pair<ll, Vertex>;
        std::priority_queue<length, std::vector<length>, std::greater<length>>
                                                                            pq;
        memset(d, 0x3f, sizeof(ll) * (V + 1)); // initializing d[] to INF
        d[s] = 0LL;
        p[s] = -1;
        pq.push(length(d[s], s));
        while (not pq.empty()) {
            Vertex u = pq.top().second;
            ll dist = pq.top().first;
            pq.pop();

            if (dist > d[u])
                continue;

            for (Edge e : adj[u]) {
                auto[v, len] = e;
                if (len + d[u] < d[v]) {
                    d[v] = len + d[u];
                    p[v] = u;
                    pq.push(length(d[v], v));
                }
            }
        }
    }

    /**
     * @brief prints a path from start vertex s to destination vertex t. Also
     * returns a vector container having the path.
     *
     * @param s Start vertex s.
     * @param t Destination vertex t.
     * @return std::vector<Vertex> Vector Container having the correct order
     * of vertices in its path.
     */
    std::vector<Vertex> printPath(Vertex s, Vertex t) {
        std::vector<Vertex> path;
        path.reserve(V);
        for (Vertex u = t; u != s; u = p[u])
            path.push_back(u);
        path.push_back(s);
        std::reverse(path.begin(), path.end());
        for (int u : path) {
            std::cout << u << ' ';
        }
        std::cout << "\n";
        return path;
    }

};
```

The C++ STL `std::priority_queue` uses a binary heap. Therefore, the push and pop operations would be $O(\log V)$.
Hence, the above function `solveShortestPaths(s)` runs in $O(E \log V)$.

The process of printing is also very simple. The idea is to store the parent of destination vertex u, $p[u]$ in a container and assign u as $p[u]$. The function `printPath(s, t)` runs in $O(V)$.

Floyd-Warshall Algorithm

Given a directed or an undirected graph with V vertices, the task is to find d_{ij}, the length of the shortest path between each pair of vertices i and j. The algorithm also detects negative cycle. If there exists a vertex u such that $d_{uu} < 0$, i.e., the distance from u to itself is negative, then the graph has a negative cycle.

We number the vertices as 1 to V and maintain a matrix, $d[][]$ for storing the value of d_{ij}.

The algorithm has V-phases. After the n-phase, $d[i][j]$ for any vertices i and j will have the length of the shortest path between i and j, considering only the set of vertices $\{1, 2, \ldots, n\}$.

For the 0-th phase, $d[i][j] = weight_{ij}$, if there exists an edge between i and j. Otherwise, $d[i][j] = \infty$, which is some large number.

In every phase, for every pair of vertices i and j, we minimise the value of $d[i][j]$ as a minimum over $(d[i][n] + d[n][j])$. There are V phases and since there are V^2 pairs of vertices, every phase requires $O(V^2)$ time.

Below is a implementation of the algorithm in `floydWarshall.cpp` which uses a adjacency matrix instead of a list. It takes the `weightedGraph` class object as a parameter defined in the `graph.hpp`.

```cpp
// floydWarshall.cpp

#include "graph.hpp"
using ll = long long int;
const ll INF = 0x3f3f3f3f3f3f3f3f;

/**
 * @brief Finds the lengths of shortest paths between
 * all pair of vertices using Floyd-Warshall's algorithm.
 * Prints "Graph has a negative cycle" if a negative cycle
 * exists.
 *
 * @param G The weighted Graph object G.
 * @return std::vector<std::vector<ll>> A vector of vectors
 * container storing the result. d[0][0] = 0 if graph has a
 * negative cycle, otherwise 1.
 */
std::vector<std::vector<ll>>
floydWarshall(const weightedGraph &G) {

    // Number of vertices
    int V = G.getV();

    // initialize d[i][j] to INF
    std::vector<std::vector<ll>>
        d(V + 1, std::vector<ll> (V + 1, INF));

    for (Vertex i = 1; i <= V; ++i)
```

```
        d[i][i] = 0LL;

    // If there is an edge between i and j,
    // set d[i][j] = weight
    for (Vertex u = 1; u <= V; ++u) {
        const std::vector<Edge>& adj = G.getAdj(u);
        for (Edge e : adj) {
            d[u][e.first] = std::min(d[u][e.first],
                                     (ll)e.second);
        }
    }

    for (Vertex n = 1; n <= V; ++n) {
        for (Vertex i = 1; i <= V; ++i) {
            for (Vertex j = 1; j <= V; ++j) {
                d[i][j] = std::min(d[i][j],
                                   (d[i][n] + d[n][j]));
            }
         }
    }

    for (Vertex u = 1; u <= V; ++u) {
        if (d[u][u] < 0) {
            std::cout << "Graph has a negative cycle\n";
            d[0][0] = 0;
            return d;
        }
    }
    d[0][0] = 1;
    return d;
}
```

Therefore, for this algorithm,

- Time complexity: $O(V^3)$.
- Space complexity: $O(V^2)$.

Minimum Spanning Tree (MST)

Given an undirected, weighted graph, find a subtree of this graph which connects all vertices together, without any cycles and has the least weight (i.e, total sum of weights of the all the edges is minimum) of all possible spanning trees. This spanning tree is known as minimum spanning tree.

An MST is unique if all the weights of the edges are distinct. Otherwise, there may be minimum spanning trees. The algorithms below give out one such particular MST of all possible MSTs for the given graph.

Prim's algorithm

Prim's algorithm builds the minimum spanning tree gradually by adding edges one at a time. At first, the MST consists only of a single vertex. This can be the root vertex or any arbitrary vertex. Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. Now, we have selected two vertices. Let this be a set of selected vertices. Then, we select an edge with the minimum weight that has one end as a selected vertex from the set of selected vertices. This process of adding the edge with minimum weight with one end in selected vertices set is done until all vertices are selected or the MST consists of $V-1$ edges.

The process of extracting minimum edge is $O(E)$ if done naively. This extraction is performed for $V-1$ times to get a complete MST. Therefore, total complexity is $O(EV)$. This is really slow.

But if we use a data structure like `std::priority_queue` which is based on a binary heap, the extraction is done in $O(\log V)$. This approach uses a slightly different method which is shared below.

Below is an implementation of the algorithm in `prim.hpp`, which contains a `struct Prim` derived publicly from `class  weightedGraph` from `graph.hpp` mentioned before.

```
// prim.hpp

#ifndef PRIM_HPP
#define PRIM_HPP

#include "graph.hpp"

struct Prim : public weightedGraph {
    const long long int INF = 0x3f3f3f3f3f3f3f3f;

    /**
     * @brief Constructs a new Prim object.
     *
     * @param _V Number of vertices.
     * @param _directed Default = false. specify true if graph is directed.
     */
    Prim(Vertex _V, bool _directed = false) :
        weightedGraph(_V, _directed) {}

    /**
     * @brief Constructs a new Prim object from a weighted Graph object.
     *
     * @param G The weighted Graph Object to be copied.
     */
    Prim(weightedGraph G) : weightedGraph(G) {}

    /**
     * @brief Constructs a new Prim object from an old one.
     * It calls the weightedGraph copy constructor.
     *
     * @param copy The old Prim object to be copied.
     */
    Prim(const Prim &copy) : weightedGraph(copy) {}

    /**
     * @brief Solves the problem of minimum spanning tree using Prim's
     * algorithm.
     *
     * @return ll The sum of all the weights in the minimum spanning tree.
     */
    long long int PrimMST() {
        long long int sumMST = 0LL;
        int weightCount = 0;
        std::vector<bool> visited(V + 1);
        std::priority_queue<Edge, std::vector<Edge>,
                                    std::greater<Edge>> pQ;

        // first - Edge weight.
        // second - Vertex.
        pQ.push(Edge(0, 1));

        while (not pQ.empty()) {
            Edge E = pQ.top();
            pQ.pop();
            Vertex u = E.second;
            if (visited[u])
                continue;
            visited[u] = true;
            sumMST += 1LL * E.first;
            weightCount++;
            if (weightCount == V)
                break;
            for (Edge e : adj[u]) {
                int v = e.first;
                if (visited[v])
                    continue;
                pQ.push(Edge(e.second, v));
            }
        }

        return sumMST;
    }
};

#endif
```

For this approach,

- Time complexity: $O(E \log V)$.
- Space complexity: $O(V + E)$.

Kruskal's algorithm

Kruskal's algorithm, initially, considers every node of the graph, V different isolated forests. Then, the edges are sorted based on their weights. At every iteration, the two trees on the ends of an edge are merged into one. Gradually, after E iterations, several trees are merged into one. The idea is to color all trees with different color in the beginning and as they get merged, color them the same color.

We are going to use a data structure known as Disjoint Set Union or Union Find due to its operations of combining any two sets (union) and finding the set a particular element belongs to (find). With a technique called path compression, these operations are reduced to $O(\log n)$ to $O(1)$ on average. Since the focus is on the algorithm, we will move on to the implementation.

This is a implementation very different from the other implementations in this book. It does not make use of the header `graph.hpp`. A different `strcut Edge` is defined for the purpose of this algorithm as there is a need to sort the edges in the ascending order of their weights.

```cpp
// kruskal.cpp

#include<vector>
#include<algorithm>

struct Edge {
    int u, v, weight;
    Edge(int a, int b, int w) :
        u(a), v(b), weight(w) {}
    bool operator<(const Edge& other) const {
        return (weight < other.weight);
    }
};

struct DisjointSetUnion {
    int n;
    int *parent, *size;
    DisjointSetUnion(int _n) : n(_n) {
        parent = new int[n + 1]();
        size = new int[n + 1]();
        for (int i = 1; i <= n; ++i) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /**
     * @brief Finds the parent of the set containing vertex v
     * with path compression.
     *
     * @param v The vertex whose parent set is to be found.
     * @return int The parent of the set containing v.
     */
    int findSet(int v) {
        if (v == parent[v])
            return v;
        parent[v] = findSet(parent[v]);
        return parent[v];
    }

    /**
     * @brief Unions two sets containing a and b.
     * Calls the findSet function.
     *
     * @param a The vertex a
     * @param b The vertex b
     * @return true Returns true if a union was done.
     * @return false Returns false if no union was done,
     * i.e., params a and b already belonged to the same
     * set.
     */
    bool unionSets(int a, int b) {
        a = findSet(a);
        b = findSet(b);
        if (a != b) {
            if (size[a] < size[b])
                std::swap(a, b);
            parent[b] = a;
            size[a] += size[b];
            size[b] = 0;
            return true;
        }
        return false;
    }
};

/**
 * @brief Solves the problem of minimum spanning tree
 * using Kruskal's algorithm.
 *
 * @param edges Container of edges.
 * @param V Number of vertices.
 * @return long long int The sum of all the weights in the minimum spanning tree.
 */
long long int
kruskalMST(std::vector<Edge> edges, int V) {
    std::sort(edges.begin(), edges.end());
    long long int sumMST = 0LL;
    DisjointSetUnion dsu(V);
    for (Edge e : edges) {
        if (dsu.unionSets(e.u, e.v)) {
            sumMST += 1LL * e.weight;
        }
    }
    return sumMST;
```

```
}
```

The average time for both `unionSets(a, b)` and `findSet(v)` is $O(1)$. These function is called for $O(E)$ times . The construction of the structure takes $O(V)$ time. The `std::sort` sorts the container in $O(E \log E) \approx O(E \log V)$ time.

Time $= O(E + V + E \log V) \ = \ O(E \log V)$.

Therefore, for this algorithm,

- Time complexity: $O(E \log V)$.
- Space complexity: $O(V + E)$.

Lowest Common Ancestor

(Using RMQ)

Given a tree G and queries of the from (u_1, u_2), for each query, the task is to find the lowest common ancestor, i.e. a vertex u that lies on the path from root to u_1 and the path from root to u_2, and the the vertex u should be the lowest (lowest depth possible).

The idea is to preprocess the tree using DFS at root. We do an *Euler tour* of the tree and store the order of visiting vertices in a container $euler$ and store the depth of all nodes in $depth$ as we perform the DFS. We also maintain an array $firstFoundAt[]$ which stores the position of the vertex i in the container $euler$ when it was first encountered during the Euler tour.

Using this information, we can find the LCA of any two vertices in the tree. It is obvious to see that from $euler[firstFoundAt[u_1]]$ to $euler[firstFoundAt[u_2]]$, this is a path between u_1 and u_2. The $LCA(u_1, u_2)$ lies on this path. It is the vertex with lowest depth in all the vertices on the path said above.

We can maintain a segment tree to return the vertex in the Euler tour with minimum depth. This is how the problem of lowest common ancestor is associated with Range Minimum Query (RMQ). The segment tree performs update and minimum extraction operations in $(O \log V)$ time with $O(V)$ preprocessing time needed for building the tree.

Below is the implementation of the approach discussed above, in a `struct LCA`.

```cpp
// lcaSegmentTree.cpp

#include <vector>
#include <algorithm>

struct LCA {
    const int INF = 0x3f3f3f3f;
    std::vector<int> depth, euler, firstFoundAt, segTree;
    std::vector<bool> visited;
    int V, segTreeSize;

    LCA(std::vector<std::vector<int>> &adj, int _V, int root = 1) :
        V(_V),
        depth(std::vector<int>(V + 1)),
        firstFoundAt(std::vector<int>(V + 1)),
        visited(std::vector<bool>(V + 1)) {

        euler.reserve(2 * V);
        depth[root] = -1;
        eulerTourDFS(adj, root, root);
        depth[root] = INF;
        int m = euler.size(), sz;
        for (sz = 1; sz < m; sz <<= 1);
        segTree.resize(sz << 1);
        for (int i = 0; i < m; ++i)
            segTree[sz + i] = euler[i];

        for (int i = sz - 1; i >= 1; --i) {
            int l = segTree[i << 1], r = segTree[i << 1 | 1];
            segTree[i] = (depth[l] > depth[r] ? r : l);
        }
        segTreeSize = sz << 1;
    }

    void eulerTourDFS(std::vector<std::vector<int>> &adj, int u, int parent) {
        visited[u] = true;
        depth[u] = depth[parent] + 1;
        firstFoundAt[u] = int(euler.size());
        euler.push_back(u);
        for (int v : adj[u]) {
            if (not visited[v]) {
                eulerTourDFS(adj, v, u);
                euler.push_back(u);
            }
        }
    }

    int query(int L, int R, int l = 0, int r = - 1, int v = 1)
    {
        if (r == -1)
            r += segTreeSize;
        if (R < l or L > r)
            return 0;
        if (L <= l and R >= r)
            return segTree[v];
        int mid = l + ((r - l) >> 1);
        int left = query(L, R, l, mid, v << 1);
        int right = query(L, R, mid + 1, r, v << 1 | 1);
        return (depth[left] > depth[right] ? right : left);
    }

    int lca(int u, int v) {
        int l = firstFoundAt[u];
        int r = firstFoundAt[v];
        if (l > r)
            std::swap(l, r);
        return query(l, r);
    }
};

int main() {}
```

The segment query requires $O(\log V)$ time. The preprocessing time for the segment tree is $O(V)$ and for the Euler Tour DFS is $O(V + E)$.

Therefore,

- Time complexity for preprocessing: $O(V + E)$.

- Time complexity for LCA query: $O(\log V)$.
- Space complexity: $O(V)$.

The main idea was to understand how the *Euler Tour* of the tree with RMQ can be used to find the LCA of any two vertices in the tree. Instead of a segment tree, a sparse table can also be used.

The RMQ operation in a sparse table requires $O(1)$ time but $O(V \log V)$ preprocessing time to build the table.

Similarly, a square-root decomposition tree can be used whose query requires $O(\sqrt{V})$ time and $O(V)$ preprocessing time.

Lowest Common Ancestor

(Binary Lifting)

Given a tree G and queries of the from (u_1, u_2), for each query, the task is to find the lowest common ancestor, $LCA(u_1, u_2)$ described in the last topic.

The algorithm discussed before used RMQ on *Euler Tour* of the tree. It found the vertex with the least depth on the path between u_1 and u_2. However, that algorithm can only do limited tasks.

This algorithm can be used to tell the weight of the heaviest edge or the lightest edge in the path between any two vertices. This application is interesting and has many uses in other problems. One such problem is finding the second best minimum spanning tree. For that purpose, we will learn this algorithm.

In Binary Lifting, we will maintain an array $ancestor[i][j]$ which stores the $2^{j\text{-th}}$ ancestor of the node i. This allows us to jump to any ancestor of any vertex in $O(\log V)$ time. This array can be filled with a simple DFS traversal. We will also maintain arrays $tin[]$ and $tout[]$ which record the in-time and out-time of each vertex during the DFS.

Consider a query $LCA(u, v)$. First, we check if u or v are ancestors of each other using the $tin[]$ and $tout[]$ arrays in $O(1)$ time. If yes, we return the result. Otherwise, we jump to an ancestor of u which is also the highest node that is not an ancestor of v. That is, we find a node q such that q is not an ancestor of v, but $ancestor[q][0]$ is. We can find q in $O(\log V)$ time, again using $ancestor[][]$.

Let $L = \lceil \log V \rceil$. Assign $i = L$. If $ancestor[u][i]$ is not an ancestor of v, then we assign $u = ancestor[u][i]$ and decrement i. If $ancestor[u][i]$ is an ancestor, then we just decrement i. Clearly, after doing this for all non-negative i the node u will be the desired node - i.e. u is still not an ancestor of v, but $ancestor[u][0]$ is.

Now, obviously, the answer to LCA will be $ancestor[u][0]$ - i.e., the node with the highest depth among the ancestors of the node u, which is also an ancestor of v.

Below is the implementation of the approach discussed above, in a `struct LCA`.

```
// lcaBinaryLift.cpp

#include <vector>
#include <algorithm>

struct LCA {
    int timer, L, V;
    std::vector<int> tin, tout;
    std::vector<std::vector<int>> ancestor;

    LCA(std::vector<std::vector<int>> &adj, int _V, int root = 1) :
        timer(0),
        V(_V),
        tin(std::vector<int>(V + 1)),
        tout(std::vector<int>(V + 1)) {

        for (L = 1; L < V; L <<= 1); // L = ceil(log V)
        ancestor.assign(V + 1, std::vector<int>(L + 1));
        dfs(adj, root, root);
    }

    void dfs(std::vector<std::vector<int>> &adj, int u, int parent) {
        tin[u] = ++timer;
        ancestor[u][0] = parent;
        for (int i = 1; i <= L; ++i)
            ancestor[u][i] = ancestor[ancestor[u][i - 1]][i - 1];
        for (int v : adj[u]) {
            if (v != parent) {
                dfs(adj, v, u);
            }
        }
        tout[u] = ++timer;
    }

    bool isAncestor(int u, int v) {
        return (tin[u] <= tin[v] and tout[u] >= tout[v]);
    }

    int lca(int u, int v)
    {
        if (isAncestor(u, v))
            return u;
        if (isAncestor(v, u))
            return v;
        for (int i = L; i >= 0; --i) {
            if (not isAncestor(ancestor[u][i], v))
                u = ancestor[u][i];
        }
        return ancestor[u][0];
    }
};
```

So answering a LCA query will iterate i from L to 0 and checks in each iteration if one node is the ancestor of the other. So, each query can be answered in $O(\log V)$.

Therefore,

- Time complexity for preprocessing: $O(V \log V)$.
- Time complexity for LCA query: $O(\log V)$.
- Space complexity: $O(V \log V)$.