Algorithm Reference - Cpp

Bipartite Match

bipartite_match.cpp

```cpp
/*
A bipartite matching algorithm finds the largest set of pairings between two disjoint vertex sets U and V
in a bipartite graph such that no vertex is in more than one pair.

Augmenting paths: repeatedly search for a path that alternates between unmatched and matched edges, starting and ending at free vertices. Flipping the edges along such a path increases the matching size by 1.

Time complexity: O(V · E), where V is the number of vertices and E the number of edges.
*/

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <cassert>

template<typename SourceT, typename SinkT>
class BipartiteMatch {
private:
    std::map<SourceT, std::vector<SinkT>> edges;
    std::map<SourceT, SinkT> used_sources;
    std::map<SinkT, SourceT> used_sinks;
    std::map<SourceT, int> coloring;

    void flip(std::vector<SourceT>& source_stack, std::vector<SinkT>& sink_stack) {
        while (!source_stack.empty()) {
            used_sources[source_stack.back()] = sink_stack.back();
            used_sinks[sink_stack.back()] = source_stack.back();
            source_stack.pop_back();
            sink_stack.pop_back();
        }
    }

    bool update(SourceT start_source, int cur_color) {
        if (used_sources.find(start_source) != used_sources.end()) {
            return false;
        }

        std::vector<SourceT> source_stack = {start_source};
        std::vector<SinkT> sink_stack;
        std::vector<size_t> index_stack = {0};

        while (true) {
            SourceT source = source_stack.back();
            size_t index = index_stack.back();
            index_stack.pop_back();

            if (index == edges[source].size()) {
                if (index_stack.empty()) {
                    return false;
                }
                source_stack.pop_back();
                sink_stack.pop_back();
                continue;
            }
            index_stack.push_back(index + 1);

            SinkT sink = edges[source][index];
            sink_stack.push_back(sink);

            if (used_sinks.find(sink) == used_sinks.end()) {
                flip(source_stack, sink_stack);
                return true;
            }
```

```
            source = used_sinks[sink];
            if (coloring[source] == cur_color) {
                sink_stack.pop_back();
            } else {
                coloring[source] = cur_color;
                source_stack.push_back(source);
                index_stack.push_back(0);
            }
        }
    }

public:
    std::map<SourceT, SinkT> match;

    BipartiteMatch(const std::vector<std::pair<SourceT, SinkT>>& edge_list) {
        for (const auto& [source, sink] : edge_list) {
            edges[source].push_back(sink);
        }

        // Get ordered sources for deterministic behavior
        std::vector<SourceT> ordered_sources;
        for (const auto& [source, _] : edges) {
            ordered_sources.push_back(source);
            coloring[source] = 0;
        }

        // Initial pass
        for (const auto& [source, sink] : edge_list) {
            if (used_sources.find(source) == used_sources.end() &&
                used_sinks.find(sink) == used_sinks.end()) {
                used_sources[source] = sink;
                used_sinks[sink] = source;
                break;
            }
        }

        bool progress = true;
        int cur_color = 1;
        while (progress) {
            progress = false;
            for (const auto& source : ordered_sources) {
                if (update(source, cur_color)) {
                    progress = true;
                }
            }
            cur_color++;
        }

        match = used_sources;
    }
};

void test_main() {
    BipartiteMatch<int, std::string> b({
        {1, "X"}, {2, "Y"}, {3, "X"}, {1, "Z"}, {2, "Z"}, {3, "Y"}
    });
    assert(b.match.size() == 3);
    assert(b.match[1] == "Z");
    assert(b.match[2] == "Y");
    assert(b.match[3] == "X");
}

// Don't write tests below during competition.

void test_a() {
    BipartiteMatch<int, double> bm({
        {1, 2.2}, {2, 3.3}, {1, 1.1}, {2, 2.2}, {3, 3.3}
    });
    assert(bm.match[1] == 1.1);
    assert(bm.match[2] == 2.2);
    assert(bm.match[3] == 3.3);
```

```
}

void test_b() {
    BipartiteMatch<std::string, std::string> bm({
        {"1", "3"}, {"2", "4"}, {"3", "2"}, {"4", "4"}, {"1", "1"}
    });
    assert(bm.match["3"] == "2");
    assert(bm.match["1"] == "3");
    assert(bm.match["2"] == "4");
}

void test_c() {
    BipartiteMatch<int, std::string> bm({
        {1, "B"}, {2, "A"}, {3, "A"}
    });
    assert(bm.match[1] == "B");
    assert(bm.match[2] == "A");
    assert(bm.match.size() == 2);
}

void test_empty_graph() {
    BipartiteMatch<int, int> bm({});
    assert(bm.match.empty());
}

void test_single_edge() {
    BipartiteMatch<int, int> bm({{1, 2}});
    assert(bm.match.size() == 1);
    assert(bm.match[1] == 2);
}

void test_no_matching() {
    // All sources want same sink
    BipartiteMatch<int, std::string> bm({
        {1, "A"}, {2, "A"}, {3, "A"}
    });
    // Only one can be matched
    assert(bm.match.size() == 1);
    assert(bm.match[bm.match.begin()->first] == "A");
}

void test_perfect_matching() {
    // Perfect matching possible
    BipartiteMatch<int, int> bm({
        {1, 10}, {2, 20}, {3, 30}
    });
    assert(bm.match.size() == 3);
}

void test_augmenting_path() {
    // Requires augmenting path to find maximum matching
    BipartiteMatch<int, std::string> bm({
        {1, "A"}, {1, "B"},
        {2, "B"}, {2, "C"},
        {3, "C"}
    });
    assert(bm.match.size() == 3);
}

void test_large_bipartite() {
    // Larger graph
    std::vector<std::pair<int, int>> edges;
    for (int i = 0; i < 10; i++) {
        for (int j = i; j < std::min(i + 3, 10); j++) {
            edges.push_back({i, j + 100});
        }
    }

    BipartiteMatch<int, int> bm(edges);
    // Should find a good matching
    assert(bm.match.size() >= 8);
}
```

```cpp
int main() {
    test_a();
    test_b();
    test_c();
    test_empty_graph();
    test_single_edge();
    test_no_matching();
    test_perfect_matching();
    test_augmenting_path();
    test_large_bipartite();
    test_main();
    std::cout << "All tests passed!" << std::endl;
    return 0;
}
```

Dijkstra

dijkstra.cpp

```cpp
/*
Dijkstra's algorithm for single-source shortest path in weighted graphs.

Finds shortest paths from a source vertex to all other vertices in a graph with
non-negative edge weights. Uses a priority queue (heap) for efficient vertex selection.

Time complexity: O((V + E) log V) with binary heap, where V is vertices and E is edges.
Space complexity: O(V + E) for the graph representation and auxiliary data structures.
*/

#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <set>
#include <optional>
#include <algorithm>
#include <cassert>
#include <limits>

template<typename NodeT, typename WeightT>
class Dijkstra {
private:
    WeightT infinity;
    WeightT zero;
    std::map<NodeT, std::vector<std::pair<NodeT, WeightT>>> graph;

public:
    Dijkstra(WeightT infinity, WeightT zero) : infinity(infinity), zero(zero) {}

    void add_edge(NodeT u, NodeT v, WeightT weight) {
        graph[u].push_back({v, weight});
    }

    std::pair<std::map<NodeT, WeightT>, std::map<NodeT, std::optional<NodeT>>>
    shortest_paths(NodeT source) {
        std::map<NodeT, WeightT> distances;
        std::map<NodeT, std::optional<NodeT>> predecessors;
        distances[source] = zero;
        predecessors[source] = std::nullopt;

        // Min heap: pair of (distance, node)
        std::priority_queue<
            std::pair<WeightT, NodeT>,
            std::vector<std::pair<WeightT, NodeT>>,
            std::greater<std::pair<WeightT, NodeT>>
        > pq;

        pq.push({zero, source});
        std::set<NodeT> visited;

        while (!pq.empty()) {
            auto [current_dist, u] = pq.top();
            pq.pop();

            if (visited.count(u)) {
                continue;
            }
            visited.insert(u);

            if (graph.find(u) == graph.end()) {
                continue;
            }

            for (const auto& [v, weight] : graph[u]) {
                WeightT new_dist = current_dist + weight;
```

```
                if (distances.find(v) == distances.end() || new_dist < distances[v]) {
                    distances[v] = new_dist;
                    predecessors[v] = u;
                    pq.push({new_dist, v});
                }
            }
        }

        return {distances, predecessors};
    }

    std::optional<std::vector<NodeT>> shortest_path(NodeT source, NodeT target) {
        auto [distances, predecessors] = shortest_paths(source);

        if (predecessors.find(target) == predecessors.end()) {
            return std::nullopt;
        }

        std::vector<NodeT> path;
        std::optional<NodeT> current = target;

        while (current.has_value()) {
            path.push_back(current.value());
            current = predecessors[current.value()];
        }

        std::reverse(path.begin(), path.end());
        return path;
    }
};

void test_main() {
    Dijkstra<std::string, double> d(std::numeric_limits<double>::infinity(), 0.0);
    d.add_edge("A", "B", 4.0);
    d.add_edge("A", "C", 2.0);
    d.add_edge("B", "C", 1.0);
    d.add_edge("B", "D", 5.0);
    d.add_edge("C", "D", 8.0);

    auto [distances, _] = d.shortest_paths("A");
    assert(distances["D"] == 9.0);

    auto path = d.shortest_path("A", "D");
    assert(path.has_value());
    assert(path.value() == std::vector<std::string>({"A", "B", "D"}));
}

// Don't write tests below during competition.

void test_single_node() {
    Dijkstra<std::string, double> d(std::numeric_limits<double>::infinity(), 0.0);

    auto [distances, predecessors] = d.shortest_paths("A");
    assert(distances.size() == 1);
    assert(distances["A"] == 0.0);
    assert(predecessors["A"] == std::nullopt);

    auto path = d.shortest_path("A", "A");
    assert(path.has_value());
    assert(path.value() == std::vector<std::string>({"A"}));
}

void test_unreachable_nodes() {
    Dijkstra<int, int> d(999999, 0);
    d.add_edge(1, 2, 5);
    d.add_edge(3, 4, 3);

    auto [distances, _] = d.shortest_paths(1);
    assert(distances[2] == 5);
    assert(distances.find(3) == distances.end());
    assert(distances.find(4) == distances.end());
```

```
}

void test_multiple_paths() {
    Dijkstra<std::string, int> d(999999, 0);
    d.add_edge("S", "A", 2);
    d.add_edge("S", "B", 2);
    d.add_edge("A", "T", 3);
    d.add_edge("B", "T", 3);

    auto [distances, _] = d.shortest_paths("S");
    assert(distances["T"] == 5);

    auto path = d.shortest_path("S", "T");
    assert(path.has_value());
    assert(path.value().size() == 3);
    assert(path.value()[0] == "S");
    assert(path.value()[2] == "T");
}

void test_self_loops() {
    Dijkstra<int, int> d(999999, 0);
    d.add_edge(1, 1, 5);
    d.add_edge(1, 2, 3);

    auto [distances, _] = d.shortest_paths(1);
    assert(distances[1] == 0);
    assert(distances[2] == 3);
}

void test_negative_zero_weights() {
    Dijkstra<std::string, double> d(std::numeric_limits<double>::infinity(), 0.0);
    d.add_edge("A", "B", 0.0);
    d.add_edge("B", "C", 0.0);
    d.add_edge("A", "C", 5.0);

    auto [distances, _] = d.shortest_paths("A");
    assert(distances["C"] == 0.0);  // Should take A->B->C path
}

void test_dense_graph() {
    // Complete graph with 5 nodes
    Dijkstra<int, int> d(999999, 0);

    // Add edges between all pairs
    std::map<std::pair<int, int>, int> weights = {
        {{0, 1}, 4}, {{0, 2}, 2}, {{0, 3}, 7}, {{0, 4}, 1},
        {{1, 0}, 4}, {{1, 2}, 3}, {{1, 3}, 2}, {{1, 4}, 5},
        {{2, 0}, 2}, {{2, 1}, 3}, {{2, 3}, 4}, {{2, 4}, 8},
        {{3, 0}, 7}, {{3, 1}, 2}, {{3, 2}, 4}, {{3, 4}, 6},
        {{4, 0}, 1}, {{4, 1}, 5}, {{4, 2}, 8}, {{4, 3}, 6},
    };

    for (const auto& [edge, weight] : weights) {
        d.add_edge(edge.first, edge.second, weight);
    }

    auto [distances, _] = d.shortest_paths(0);

    // Verify shortest distances from node 0
    assert(distances[1] == 4);
    assert(distances[2] == 2);
    assert(distances[3] == 6);  // 0->1->3 = 4+2 = 6
    assert(distances[4] == 1);
}

void test_large_graph() {
    // Linear chain: 0->1->2->...->99
    Dijkstra<int, int> d(999999, 0);

    for (int i = 0; i < 99; i++) {
        d.add_edge(i, i + 1, 1);
    }
```

```cpp
    auto [distances, _] = d.shortest_paths(0);

    // Distance to node i should be i
    for (int i = 0; i < 100; i++) {
        assert(distances[i] == i);
    }

    // Test path reconstruction
    auto path = d.shortest_path(0, 50);
    assert(path.has_value());
    assert(path.value().size() == 51);
    for (int i = 0; i <= 50; i++) {
        assert(path.value()[i] == i);
    }
}

void test_decimal_weights() {
    Dijkstra<std::string, double> d(999999.0, 0.0);
    d.add_edge("A", "B", 1.5);
    d.add_edge("B", "C", 2.7);
    d.add_edge("A", "C", 5.0);

    auto [distances, _] = d.shortest_paths("A");
    assert(std::abs(distances["C"] - 4.2) < 1e-9);  // 1.5 + 2.7
}

void test_stress_many_nodes() {
    // Star graph: center connected to many nodes
    Dijkstra<int, int> d(999999, 0);

    int center = 0;
    for (int i = 1; i <= 500; i++) {  // 500 nodes connected to center
        d.add_edge(center, i, i);
    }

    auto [distances, _] = d.shortest_paths(center);

    // Distance to node i should be i
    for (int i = 1; i <= 500; i++) {
        assert(distances[i] == i);
    }

    // Path from center to any node should be direct
    auto path = d.shortest_path(center, 100);
    assert(path.has_value());
    assert(path.value().size() == 2);
    assert(path.value()[0] == 0);
    assert(path.value()[1] == 100);
}

int main() {
    test_single_node();
    test_unreachable_nodes();
    test_negative_zero_weights();
    test_dense_graph();
    test_large_graph();
    test_multiple_paths();
    test_self_loops();
    test_decimal_weights();
    test_stress_many_nodes();
    test_main();
    std::cout << "All tests passed!" << std::endl;
    return 0;
}
```

Edmonds Karp

edmonds_karp.cpp

```cpp
/*
Edmonds-Karp is a specialization of the Ford-Fulkerson method for computing the maximum flow in a
directed graph.

* It repeatedly searches for an augmenting path from source to sink.
* The search is done with BFS, guaranteeing the path found is the shortest (fewest edges).
* Each augmentation increases the total flow, and each edge's residual capacity is updated.
* The algorithm terminates when no augmenting path exists.

Time complexity: O(V · E²), where V is the number of vertices and E the number of edges.
*/

#include <vector>
#include <queue>
#include <algorithm>
#include <limits>
#include <cassert>
#include <iostream>

template<typename T>
class EdmondsKarp {
private:
    int n;
    std::vector<std::vector<T>> capacity;
    std::vector<std::vector<T>> flow;
    T total_flow;

public:
    EdmondsKarp(int vertices) : n(vertices), total_flow(0) {
        capacity.assign(n, std::vector<T>(n, 0));
        flow.assign(n, std::vector<T>(n, 0));
    }

    void add_edge(int from, int to, T cap) {
        capacity[from][to] += cap;
    }

    bool bfs(int source, int sink, std::vector<int>& parent) {
        std::vector<bool> visited(n, false);
        std::queue<int> q;
        q.push(source);
        visited[source] = true;
        parent[source] = -1;

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

            for (int v = 0; v < n; v++) {
                // Check residual capacity: forward capacity minus forward flow, plus backward flow
                T residual = capacity[u][v] - flow[u][v];
                if (!visited[v] && residual > 0) {
                    q.push(v);
                    parent[v] = u;
                    visited[v] = true;
                    if (v == sink) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    T max_flow(int source, int sink) {
        total_flow = 0;
```

```
        std::vector<int> parent(n);

        while (bfs(source, sink, parent)) {
            T path_flow = std::numeric_limits<T>::max();

            // Find minimum residual capacity along the path
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                path_flow = std::min(path_flow, capacity[u][v] - flow[u][v]);
            }

            // Add path flow to overall flow
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                flow[u][v] += path_flow;
                flow[v][u] -= path_flow;
            }

            total_flow += path_flow;
        }

        return total_flow;
    }

    T get_total_flow() const {
        return total_flow;
    }
};

void test_main() {
    EdmondsKarp<int> e(4);
    e.add_edge(0, 1, 10);
    e.add_edge(0, 2, 8);
    e.add_edge(1, 2, 2);
    e.add_edge(1, 3, 5);
    e.add_edge(2, 3, 7);
    assert(e.max_flow(0, 3) == 12);
}

// Don't write tests below during competition.

void test_basic() {
    // Simple flow network
    // Paths: 0->1->3 (10), 0->2->3 (10), 0->1->2->3 (10) = 30 total
    EdmondsKarp<int> ek(4);
    ek.add_edge(0, 1, 20);
    ek.add_edge(0, 2, 10);
    ek.add_edge(1, 2, 30);
    ek.add_edge(1, 3, 10);
    ek.add_edge(2, 3, 20);

    int max_flow = ek.max_flow(0, 3);
    assert(max_flow == 30);
}

void test_no_flow() {
    // No path from source to sink
    EdmondsKarp<int> ek(4);
    ek.add_edge(0, 1, 10);
    ek.add_edge(2, 3, 10);

    int max_flow = ek.max_flow(0, 3);
    assert(max_flow == 0);
}

void test_single_edge() {
    // Single edge network
    EdmondsKarp<int> ek(2);
    ek.add_edge(0, 1, 5);

    int max_flow = ek.max_flow(0, 1);
    assert(max_flow == 5);
```

```
}

void test_bottleneck() {
    // Path with bottleneck
    EdmondsKarp<int> ek(4);
    ek.add_edge(0, 1, 100);
    ek.add_edge(1, 2, 1);
    ek.add_edge(2, 3, 100);

    int max_flow = ek.max_flow(0, 3);
    assert(max_flow == 1);
}

void test_parallel_edges() {
    // Multiple parallel paths
    EdmondsKarp<int> ek(4);
    ek.add_edge(0, 1, 5);
    ek.add_edge(0, 2, 5);
    ek.add_edge(1, 3, 5);
    ek.add_edge(2, 3, 5);

    int max_flow = ek.max_flow(0, 3);
    assert(max_flow == 10);
}

void test_empty_graph() {
    // Empty graph (no edges)
    EdmondsKarp<int> ek(2);
    int max_flow = ek.max_flow(0, 1);
    assert(max_flow == 0);
}

void test_complex_network() {
    // More complex network with multiple paths
    EdmondsKarp<int> ek(6);
    ek.add_edge(0, 1, 10);
    ek.add_edge(0, 2, 10);
    ek.add_edge(1, 2, 2);
    ek.add_edge(1, 3, 4);
    ek.add_edge(1, 4, 8);
    ek.add_edge(2, 4, 9);
    ek.add_edge(3, 5, 10);
    ek.add_edge(4, 3, 6);
    ek.add_edge(4, 5, 10);

    int max_flow = ek.max_flow(0, 5);
    assert(max_flow == 19);
}

int main() {
    test_basic();
    test_no_flow();
    test_single_edge();
    test_bottleneck();
    test_parallel_edges();
    test_empty_graph();
    test_complex_network();
    test_main();
    std::cout << "All Edmonds-Karp tests passed!" << std::endl;
    return 0;
}
```

Fenwick Tree

fenwick_tree.cpp

```cpp
/*
Fenwick tree (Binary Indexed Tree) for efficient range sum queries and point updates.

A Fenwick tree maintains cumulative frequency information and supports two main operations:
* update(i, delta): add delta to the element at index i
* query(i): return the sum of elements from index 0 to i (inclusive)
* range_query(left, right): return the sum of elements from left to right (inclusive)

The tree uses a clever indexing scheme based on the binary representation of indices
to achieve logarithmic time complexity for both operations.

Time complexity: O(log n) for update and query operations.
Space complexity: O(n) where n is the size of the array.
*/

#include <vector>
#include <stdexcept>
#include <cassert>
#include <iostream>

template<typename T>
class FenwickTree {
private:
    int size;
    T zero;
    std::vector<T> tree; // 1-indexed tree for easier bit manipulation

public:
    FenwickTree(int size, T zero) : size(size), zero(zero), tree(size + 1, zero) {}

    static FenwickTree from_array(const std::vector<T>& arr, T zero) {
        int n = arr.size();
        FenwickTree ft(n, zero);

        // Compute prefix sums
        std::vector<T> prefix(n + 1, zero);
        for (int i = 0; i < n; i++) {
            prefix[i + 1] = prefix[i] + arr[i];
        }

        // Build tree in O(n): each tree[i] contains sum of range [i - (i & -i) + 1, i]
        for (int i = 1; i <= n; i++) {
            int range_start = i - (i & (-i)) + 1;
            ft.tree[i] = prefix[i] - prefix[range_start - 1];
        }

        return ft;
    }

    void update(int index, T delta) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("Index out of bounds");
        }

        // Convert to 1-indexed
        index++;
        while (index <= size) {
            tree[index] = tree[index] + delta;
            // Move to next index by adding the lowest set bit
            index += index & (-index);
        }
    }

    T query(int index) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("Index out of bounds");
```

```
    }

    // Convert to 1-indexed
    index++;
    T result = zero;
    while (index > 0) {
        result = result + tree[index];
        // Move to parent by removing the lowest set bit
        index -= index & (-index);
    }
    return result;
}

T range_query(int left, int right) {
    if (left > right || left < 0 || right >= size) {
        return zero;
    }
    if (left == 0) {
        return query(right);
    }
    return query(right) - query(left - 1);
}

// Optional functionality (not always needed during competition)

T get_value(int index) {
    if (index < 0 || index >= size) {
        throw std::out_of_range("Index out of bounds");
    }
    if (index == 0) {
        return query(0);
    }
    return query(index) - query(index - 1);
}

// Find smallest index >= start_index with value > zero (REQUIRES: all updates are non-negative)
int first_nonzero_index(int start_index) {
    start_index = std::max(start_index, 0);
    if (start_index >= size) {
        return -1;  // Use -1 to indicate "not found" in C++
    }

    T prefix_before = (start_index > 0) ? query(start_index - 1) : zero;
    T total = query(size - 1);
    if (total == prefix_before) {
        return -1;
    }

    // Fenwick lower_bound: first idx with prefix_sum(idx) > prefix_before
    int idx = 0;  // 1-based cursor
    T cur = zero;  // running prefix at 'idx'
    int bit = 1;
    while (bit <= size) bit <<= 1;
    bit >>= 1;

    while (bit > 0) {
        int nxt = idx + bit;
        if (nxt <= size) {
            T cand = cur + tree[nxt];
            if (cand <= prefix_before) {  // move right while prefix <= target
                cur = cand;
                idx = nxt;
            }
        }
        bit >>= 1;
    }

    // idx is the largest position with prefix <= prefix_before (1-based).
    // The answer is idx (converted to 0-based).
    return idx;
}
```

```
    int length() const {
        return size;
    }
};

void test_main() {
    FenwickTree<int> f(5, 0);
    f.update(0, 7);
    f.update(2, 13);
    f.update(4, 19);
    assert(f.query(4) == 39);
    assert(f.range_query(1, 3) == 13);

    // Optional functionality (not always needed during competition)

    assert(f.get_value(2) == 13);
    auto g = FenwickTree<int>::from_array({1, 2, 3, 4, 5}, 0);
    assert(g.query(4) == 15);
}

// Don't write tests below during competition.

void test_basic() {
    // Test with integers
    FenwickTree<int> ft(5, 0);

    // Initial array: [0, 0, 0, 0, 0]
    assert(ft.query(0) == 0);
    assert(ft.query(4) == 0);
    assert(ft.range_query(1, 3) == 0);

    // Update operations
    ft.update(0, 5);  // [5, 0, 0, 0, 0]
    ft.update(2, 3);  // [5, 0, 3, 0, 0]
    ft.update(4, 7);  // [5, 0, 3, 0, 7]

    // Query operations
    assert(ft.query(0) == 5);
    assert(ft.query(2) == 8);  // 5 + 0 + 3
    assert(ft.query(4) == 15); // 5 + 0 + 3 + 0 + 7

    // Range queries
    assert(ft.range_query(0, 2) == 8);
    assert(ft.range_query(2, 4) == 10);
    assert(ft.range_query(1, 3) == 3);

    // Get individual values
    assert(ft.get_value(0) == 5);
    assert(ft.get_value(2) == 3);
    assert(ft.get_value(4) == 7);
}

void test_from_array() {
    std::vector<int> arr = {1, 3, 5, 7, 9, 11};
    auto ft = FenwickTree<int>::from_array(arr, 0);

    // Test that prefix sums match
    int expected_sum = 0;
    for (int i = 0; i < arr.size(); i++) {
        expected_sum += arr[i];
        assert(ft.query(i) == expected_sum);
    }

    // Test range queries
    assert(ft.range_query(1, 3) == 3 + 5 + 7); // 15
    assert(ft.range_query(2, 4) == 5 + 7 + 9); // 21

    // Test updates
    ft.update(2, 10); // arr[2] becomes 15
    assert(ft.get_value(2) == 15);
    assert(ft.range_query(1, 3) == 3 + 15 + 7); // 25
}
```

```cpp
void test_edge_cases() {
    FenwickTree<int> ft(1, 0);

    // Single element tree
    ft.update(0, 42);
    assert(ft.query(0) == 42);
    assert(ft.range_query(0, 0) == 42);
    assert(ft.get_value(0) == 42);

    // Empty range
    FenwickTree<int> ft_large(10, 0);
    assert(ft_large.range_query(5, 3) == 0); // left > right
}

void test_negative_values() {
    FenwickTree<int> ft(4, 0);

    // Mix of positive and negative updates
    ft.update(0, 10);
    ft.update(1, -5);
    ft.update(2, 8);
    ft.update(3, -3);

    assert(ft.query(3) == 10); // 10 + (-5) + 8 + (-3)
    assert(ft.range_query(1, 2) == 3); // (-5) + 8

    // Update with negative delta
    ft.update(0, -5); // Subtract 5 from position 0
    assert(ft.get_value(0) == 5);
    assert(ft.query(3) == 5); // 5 + (-5) + 8 + (-3)
}

void test_bounds_checking() {
    FenwickTree<int> ft(5, 0);

    // Test update bounds
    bool caught = false;
    try {
        ft.update(-1, 10);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        ft.update(5, 10);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    // Test query bounds
    caught = false;
    try {
        ft.query(-1);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        ft.query(5);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    // Test range_query bounds - should return zero for invalid ranges
    assert(ft.range_query(-1, 2) == 0);
```

```
    assert(ft.range_query(0, 5) == 0);

    // Test get_value bounds
    caught = false;
    try {
        ft.get_value(-1);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        ft.get_value(5);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);
}

void test_first_nonzero_bounds() {
    FenwickTree<int> ft(10, 0);
    ft.update(5, 1);

    // Negative start_index should be clamped to 0
    assert(ft.first_nonzero_index(-5) == 5);

    // Start from exactly where nonzero is
    assert(ft.first_nonzero_index(5) == 5);

    // Start past all nonzero elements
    assert(ft.first_nonzero_index(10) == -1);
    assert(ft.first_nonzero_index(100) == -1);

    // Empty tree
    FenwickTree<int> ft_empty(10, 0);
    assert(ft_empty.first_nonzero_index(0) == -1);
}

void test_linear_from_array() {
    // Test that the optimized from_array produces identical results
    std::vector<std::vector<int>> test_cases = {
        {1, 3, 5, 7, 9, 11},
        {10, -5, 8, -3, 15, 2, -7, 12},
    };

    for (const auto& arr : test_cases) {
        auto ft = FenwickTree<int>::from_array(arr, 0);

        // Verify all prefix sums match expected
        int expected_sum = 0;
        for (size_t i = 0; i < arr.size(); i++) {
            expected_sum += arr[i];
            assert(ft.query(i) == expected_sum);
        }

        // Verify individual values
        for (size_t i = 0; i < arr.size(); i++) {
            assert(ft.get_value(i) == arr[i]);
        }

        // Test range queries
        if (arr.size() >= 3) {
            int range_sum = arr[1] + arr[2];
            assert(ft.range_query(1, 2) == range_sum);
        }
    }

    // Test on larger array
    std::vector<int> large_arr(1000);
    for (int i = 0; i < 1000; i++) {
        large_arr[i] = i;
```

```
    }
    auto ft_optimized = FenwickTree<int>::from_array(large_arr, 0);

    // Verify correctness on large array
    std::vector<int> test_indices = {0, 100, 500, 999};
    for (int i : test_indices) {
        int expected = 0;
        for (int j = 0; j <= i; j++) {
            expected += large_arr[j];
        }
        assert(ft_optimized.query(i) == expected);
    }
}

void test_first_nonzero_index() {
    FenwickTree<int> ft(10, 0);
    ft.update(2, 1);
    ft.update(8, 1);
    assert(ft.first_nonzero_index(5) == 8);
    assert(ft.first_nonzero_index(8) == 8);
    assert(ft.first_nonzero_index(0) == 2);
    assert(ft.first_nonzero_index(9) == -1);
}

int main() {
    test_basic();
    test_from_array();
    test_edge_cases();
    test_bounds_checking();
    test_first_nonzero_bounds();
    test_negative_values();
    test_linear_from_array();
    test_first_nonzero_index();
    test_main();
    std::cout << "All Fenwick tree tests passed!" << std::endl;
    return 0;
}
```

Kmp

kmp.cpp

```cpp
/*
Knuth-Morris-Pratt (KMP) algorithm for efficient string pattern matching.

Finds all occurrences of a pattern string within a text string using a failure function
to avoid redundant comparisons. The preprocessing phase builds a table that allows
skipping characters during mismatches.

Time complexity: O(n + m) where n is text length and m is pattern length.
Space complexity: O(m) for the failure function table.
*/

#include <iostream>
#include <vector>
#include <string>
#include <cassert>

std::vector<int> compute_failure_function(const std::string& pattern) {
    /*
    Compute the failure function for KMP algorithm.

    failure[i] = length of longest proper prefix of pattern[0:i+1]
    that is also a suffix of pattern[0:i+1]
    */
    int m = pattern.length();
    std::vector<int> failure(m, 0);
    int j = 0;

    for (int i = 1; i < m; i++) {
        while (j > 0 && pattern[i] != pattern[j]) {
            j = failure[j - 1];
        }

        if (pattern[i] == pattern[j]) {
            j++;
        }

        failure[i] = j;
    }

    return failure;
}

std::vector<int> kmp_search(const std::string& text, const std::string& pattern) {
    /*
    Find all starting positions where pattern occurs in text.

    Returns a list of 0-indexed positions where pattern begins in text.
    */
    if (pattern.empty()) {
        return {};
    }

    int n = text.length(), m = pattern.length();
    if (m > n) {
        return {};
    }

    std::vector<int> failure = compute_failure_function(pattern);
    std::vector<int> matches;
    int j = 0;  // index for pattern

    for (int i = 0; i < n; i++) {  // index for text
        while (j > 0 && text[i] != pattern[j]) {
            j = failure[j - 1];
        }
```

```
        if (text[i] == pattern[j]) {
            j++;
        }

        if (j == m) {
            matches.push_back(i - m + 1);
            j = failure[j - 1];
        }
    }

    return matches;
}

int kmp_count(const std::string& text, const std::string& pattern) {
    /* Count number of occurrences of pattern in text. */
    return kmp_search(text, pattern).size();
}

void test_main() {
    std::string text = "ababcababa";
    std::string pattern = "aba";
    std::vector<int> matches = kmp_search(text, pattern);
    assert(matches == std::vector<int>({0, 5, 7}));
    assert(kmp_count(text, pattern) == 3);

    // Test failure function
    std::vector<int> failure = compute_failure_function("abcabcab");
    assert(failure == std::vector<int>({0, 0, 0, 1, 2, 3, 4, 5}));
}

// Don't write tests below during competition.

void test_empty_patterns() {
    // Empty pattern should return empty list
    assert(kmp_search("hello", "").empty());
    assert(kmp_count("hello", "") == 0);

    // Empty text with non-empty pattern
    assert(kmp_search("", "abc").empty());
    assert(kmp_count("", "abc") == 0);

    // Both empty
    assert(kmp_search("", "").empty());
    assert(kmp_count("", "") == 0);
}

void test_single_character() {
    // Single character pattern in single character text
    assert(kmp_search("a", "a") == std::vector<int>({0}));
    assert(kmp_search("a", "b").empty());

    // Single character pattern in longer text
    assert(kmp_search("aaaa", "a") == std::vector<int>({0, 1, 2, 3}));
    assert(kmp_search("abab", "a") == std::vector<int>({0, 2}));
    assert(kmp_search("abab", "b") == std::vector<int>({1, 3}));
}

void test_pattern_longer_than_text() {
    assert(kmp_search("abc", "abcdef").empty());
    assert(kmp_search("x", "xyz").empty());
    assert(kmp_count("short", "verylongpattern") == 0);
}

void test_overlapping_matches() {
    // Pattern that overlaps with itself
    std::string text = "aaaa";
    std::string pattern = "aa";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 1, 2}));
    assert(kmp_count(text, pattern) == 3);

    // More complex overlapping
    text = "abababab";
```

```
    pattern = "abab";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 2, 4}));
}

void test_no_matches() {
    assert(kmp_search("abcdef", "xyz").empty());
    assert(kmp_search("hello world", "goodbye").empty());
    assert(kmp_count("mississippi", "xyz") == 0);
}

void test_full_text_match() {
    std::string text = "hello";
    std::string pattern = "hello";
    assert(kmp_search(text, pattern) == std::vector<int>({0}));
    assert(kmp_count(text, pattern) == 1);
}

void test_repeated_patterns() {
    // All same character
    std::string text = "aaaaaaa";
    std::string pattern = "aaa";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 1, 2, 3, 4}));

    // Repeated subpattern
    text = "abcabcabcabc";
    pattern = "abcabc";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 3, 6}));
}

void test_failure_function_edge_cases() {
    // No repeating prefixes
    std::vector<int> failure = compute_failure_function("abcdef");
    assert(failure == std::vector<int>({0, 0, 0, 0, 0, 0}));

    // All same character
    failure = compute_failure_function("aaaa");
    assert(failure == std::vector<int>({0, 1, 2, 3}));

    // Complex pattern with multiple prefix-suffix matches
    failure = compute_failure_function("abcabcabcab");
    assert(failure == std::vector<int>({0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8}));

    // Pattern with internal repetition
    failure = compute_failure_function("ababcabab");
    assert(failure == std::vector<int>({0, 0, 1, 2, 0, 1, 2, 3, 4}));
}

void test_case_sensitive() {
    // Should be case sensitive
    assert(kmp_search("Hello", "hello").empty());
    assert(kmp_search("HELLO", "hello").empty());
    assert(kmp_search("Hello", "H") == std::vector<int>({0}));
    assert(kmp_search("Hello", "h").empty());
}

void test_special_characters() {
    std::string text = "a@b#c$d%e";
    std::string pattern = "@b#";
    assert(kmp_search(text, pattern) == std::vector<int>({1}));

    text = "...test...";
    pattern = "...";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 7}));
}

void test_large_text_small_pattern() {
    // Large text with small repeated pattern
    std::string text = std::string(1000, 'a') + "b" + std::string(1000, 'a');
    std::string pattern = "b";
    assert(kmp_search(text, pattern) == std::vector<int>({1000}));
    assert(kmp_count(text, pattern) == 1);
```

```
    // Pattern at end
    text = std::string(999, 'x') + "target";
    pattern = "target";
    assert(kmp_search(text, pattern) == std::vector<int>({999}));
}

void test_stress_many_matches() {
    // Many overlapping matches
    std::string text(100, 'a');
    std::string pattern(10, 'a');
    std::vector<int> expected;
    for (int i = 0; i <= 90; i++) {
        expected.push_back(i);
    }
    assert(kmp_search(text, pattern) == expected);
    assert(kmp_count(text, pattern) == 91);
}

void test_binary_strings() {
    // Binary pattern matching
    std::string text = "1010101010";
    std::string pattern = "101";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 2, 4, 6}));

    // No matches in binary
    text = "0000000000";
    pattern = "101";
    assert(kmp_search(text, pattern).empty());
}

void test_periodic_patterns() {
    // Highly periodic pattern
    std::string text = "abababababab";
    std::string pattern = "ababab";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 2, 4, 6}));

    // Pattern is prefix of text
    text = "abcdefghijk";
    pattern = "abcde";
    assert(kmp_search(text, pattern) == std::vector<int>({0}));
}

void test_failure_function_comprehensive() {
    // Test various complex failure function cases

    // Palindromic pattern
    std::vector<int> failure = compute_failure_function("abacaba");
    assert(failure == std::vector<int>({0, 0, 1, 0, 1, 2, 3}));

    // Pattern with nested repetitions
    failure = compute_failure_function("aabaaaba");
    assert(failure == std::vector<int>({0, 1, 0, 1, 2, 2, 3, 4}));

    // Long repetitive pattern
    failure = compute_failure_function("abababababab");
    assert(failure == std::vector<int>({0, 0, 1, 2, 3, 4, 5, 6, 7, 8}));
}

void test_unicode_strings() {
    // Unicode text and pattern
    std::string text = "αβγδεζηθ";
    std::string pattern = "γδε";
    assert(kmp_search(text, pattern) == std::vector<int>({4}));  // Note: byte offset, not char
offset

    text = "🙂🙃🙂🙃🙂";
    pattern = "🙂🙃";
    assert(kmp_search(text, pattern) == std::vector<int>({0, 8}));  // Note: byte offsets
}

int main() {
    test_empty_patterns();
```

```
    test_single_character();
    test_pattern_longer_than_text();
    test_overlapping_matches();
    test_no_matches();
    test_full_text_match();
    test_repeated_patterns();
    test_failure_function_edge_cases();
    test_case_sensitive();
    test_special_characters();
    test_large_text_small_pattern();
    test_stress_many_matches();
    test_binary_strings();
    test_periodic_patterns();
    test_failure_function_comprehensive();
    test_unicode_strings();
    test_main();
    std::cout << "All tests passed!" << std::endl;
    return 0;
}
```

Lca

lca.cpp

```cpp
/*
Lowest Common Ancestor (LCA) using binary lifting preprocessing.

Finds the lowest common ancestor of two nodes in a tree efficiently after O(n log n)
preprocessing. Binary lifting allows answering LCA queries in O(log n) time by
maintaining ancestors at powers-of-2 distances.

Time complexity: O(n log n) preprocessing, O(log n) per LCA query.
Space complexity: O(n log n) for the binary lifting table.
*/

#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <algorithm>
#include <stdexcept>
#include <cassert>

template<typename NodeT>
class LCA {
private:
    NodeT root;
    std::map<NodeT, std::vector<NodeT>> graph;
    std::map<NodeT, int> depth;
    std::map<NodeT, std::map<int, NodeT>> up;  // up[node][i] = 2^i-th ancestor
    std::set<NodeT> has_parent;  // nodes that have a parent
    int max_log;

    void dfs_depth(NodeT node, bool has_par, NodeT par, int d) {
        depth[node] = d;
        for (const auto& neighbor : graph[node]) {
            if (!has_par || neighbor != par) {
                dfs_depth(neighbor, true, node, d + 1);
            }
        }
    }

    void dfs_parents(NodeT node, bool has_par, NodeT par) {
        if (has_par) {
            up[node][0] = par;
            has_parent.insert(node);
        }
        for (const auto& neighbor : graph[node]) {
            if (!has_par || neighbor != par) {
                dfs_parents(neighbor, true, node);
            }
        }
    }

public:
    LCA(NodeT root) : root(root), max_log(0) {}

    void add_edge(NodeT u, NodeT v) {
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    void preprocess() {
        // Find max depth to determine log table size
        dfs_depth(root, false, root, 0);

        int n = depth.size();
        max_log = 0;
        while ((1 << max_log) <= n) {
            max_log++;
```

```
        }

        // Fill first column (direct parents)
        dfs_parents(root, false, root);

        // Fill binary lifting table
        for (int j = 1; j < max_log; j++) {
            for (const auto& [node, _] : depth) {
                if (up[node].count(j - 1)) {
                    NodeT parent_j_minus_1 = up[node][j - 1];
                    if (up[parent_j_minus_1].count(j - 1)) {
                        up[node][j] = up[parent_j_minus_1][j - 1];
                    }
                }
            }
        }
    }

    NodeT lca(NodeT u, NodeT v) {
        if (depth[u] < depth[v]) {
            std::swap(u, v);
        }

        // Bring u to same level as v
        int diff = depth[u] - depth[v];
        for (int i = 0; i < max_log; i++) {
            if ((diff >> i) & 1) {
                if (up[u].count(i)) {
                    u = up[u][i];
                }
            }
        }

        if (u == v) {
            return u;
        }

        // Binary search for LCA
        for (int i = max_log - 1; i >= 0; i--) {
            bool u_has = up[u].count(i);
            bool v_has = up[v].count(i);
            if (u_has && v_has && up[u][i] != up[v][i]) {
                u = up[u][i];
                v = up[v][i];
            }
        }

        if (!up[u].count(0)) {
            throw std::runtime_error("LCA computation failed - invalid tree structure");
        }
        return up[u][0];
    }

    int distance(NodeT u, NodeT v) {
        NodeT lca_node = lca(u, v);
        return depth[u] + depth[v] - 2 * depth[lca_node];
    }
};

void test_main() {
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}, {3, 6}};
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    assert(lca.lca(4, 5) == 2);
    assert(lca.lca(4, 6) == 1);
    assert(lca.distance(4, 6) == 4);
}
```

```cpp
// Don't write tests below during competition.

void test_linear_chain() {
    // Test on a simple linear chain: 1-2-3-4-5
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges = {{1, 2}, {2, 3}, {3, 4}, {4, 5}};
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    // LCA of nodes at different depths
    assert(lca.lca(1, 5) == 1);
    assert(lca.lca(2, 5) == 2);
    assert(lca.lca(3, 5) == 3);
    assert(lca.lca(4, 5) == 4);
    assert(lca.lca(5, 5) == 5);

    // Distance tests
    assert(lca.distance(1, 5) == 4);
    assert(lca.distance(2, 4) == 2);
    assert(lca.distance(3, 3) == 0);
}

void test_single_node() {
    // Test with single node tree
    LCA<int> lca(1);
    lca.preprocess();

    assert(lca.lca(1, 1) == 1);
    assert(lca.distance(1, 1) == 0);
}

void test_binary_tree() {
    // Perfect binary tree
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges = {{1, 2}, {1, 3}, {2, 4}, {2, 5}, {3, 6}, {3, 7}};
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    // Same parent
    assert(lca.lca(4, 5) == 2);
    assert(lca.lca(6, 7) == 3);

    // Different parents
    assert(lca.lca(4, 6) == 1);
    assert(lca.lca(5, 7) == 1);

    // One is ancestor of other
    assert(lca.lca(1, 4) == 1);
    assert(lca.lca(2, 5) == 2);

    // Distance tests
    assert(lca.distance(4, 5) == 2);
    assert(lca.distance(4, 7) == 4);
    assert(lca.distance(2, 3) == 2);
}

void test_star_tree() {
    // Star tree with center at 1
    LCA<int> lca(1);
    for (int i = 2; i <= 10; i++) {
        lca.add_edge(1, i);
    }

    lca.preprocess();
```

```cpp
    // All pairs should have LCA = 1 (center)
    for (int i = 2; i <= 10; i++) {
        for (int j = i + 1; j <= 10; j++) {
            assert(lca.lca(i, j) == 1);
            assert(lca.distance(i, j) == 2);
        }
    }

    // Center to leaf
    for (int i = 2; i <= 10; i++) {
        assert(lca.lca(1, i) == 1);
        assert(lca.distance(1, i) == 1);
    }
}

void test_deep_tree() {
    // Deep tree (linear chain of depth 100)
    LCA<int> lca(1);
    for (int i = 1; i < 100; i++) {
        lca.add_edge(i, i + 1);
    }

    lca.preprocess();

    // Test some LCA queries
    assert(lca.lca(1, 100) == 1);
    assert(lca.lca(50, 100) == 50);
    assert(lca.lca(25, 75) == 25);

    // Distance tests
    assert(lca.distance(1, 100) == 99);
    assert(lca.distance(50, 60) == 10);
    assert(lca.distance(25, 75) == 50);
}

void test_string_nodes() {
    // Test with string node identifiers
    LCA<std::string> lca("root");
    lca.add_edge("root", "left");
    lca.add_edge("root", "right");
    lca.add_edge("left", "left_child");
    lca.add_edge("right", "right_child");

    lca.preprocess();

    assert(lca.lca("left_child", "right_child") == "root");
    assert(lca.lca("left", "left_child") == "left");
    assert(lca.distance("left_child", "right_child") == 4);
}

void test_unbalanced_tree() {
    // Highly unbalanced tree (path with branches)
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges = {
        {1, 2}, {2, 3}, {3, 4}, {4, 5},
        {2, 10}, {3, 11}, {4, 12}
    };
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    assert(lca.lca(5, 12) == 4);
    assert(lca.lca(10, 11) == 2);
    assert(lca.lca(10, 5) == 2);
    assert(lca.lca(11, 12) == 3);

    // Distance in unbalanced tree
    assert(lca.distance(10, 5) == 4);  // 10->2->3->4->5
    assert(lca.distance(11, 12) == 3);  // 11->3->4->12
}
```

```cpp
void test_large_balanced_tree() {
    // Complete binary tree with 15 nodes (4 levels)
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges;
    for (int i = 1; i <= 7; i++) {  // Internal nodes
        int left_child = 2 * i;
        int right_child = 2 * i + 1;
        if (left_child <= 15) {
            edges.push_back({i, left_child});
        }
        if (right_child <= 15) {
            edges.push_back({i, right_child});
        }
    }

    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    // Test leaf nodes
    assert(lca.lca(8, 9) == 4);
    assert(lca.lca(10, 11) == 5);
    assert(lca.lca(8, 10) == 2);
    assert(lca.lca(12, 13) == 6);
    assert(lca.lca(8, 15) == 1);

    // Distance between leaves
    assert(lca.distance(8, 9) == 2);
    assert(lca.distance(8, 15) == 6);
}

void test_complex_tree() {
    // More complex tree structure
    LCA<int> lca(0);
    std::vector<std::pair<int, int>> edges = {
        {0, 1}, {0, 2}, {0, 3},
        {1, 4}, {1, 5},
        {2, 6}, {2, 7}, {2, 8},
        {3, 9},
        {4, 10}, {4, 11},
        {6, 12}, {6, 13},
        {9, 14}, {9, 15}
    };
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    // Test various combinations
    assert(lca.lca(10, 11) == 4);
    assert(lca.lca(4, 5) == 1);
    assert(lca.lca(10, 5) == 1);
    assert(lca.lca(12, 8) == 2);
    assert(lca.lca(14, 15) == 9);
    assert(lca.lca(10, 14) == 0);

    // Complex distance calculations
    assert(lca.distance(10, 11) == 2);  // 10->4->11
    assert(lca.distance(10, 14) == 6);  // 10->4->1->0->3->9->14
    assert(lca.distance(12, 8) == 3);   // 12->6->2->8
}

void test_edge_cases() {
    // Tree with only two nodes
    LCA<int> lca(1);
    lca.add_edge(1, 2);
    lca.preprocess();
```

```
    assert(lca.lca(1, 2) == 1);
    assert(lca.lca(2, 1) == 1);
    assert(lca.distance(1, 2) == 1);

    // Same node queries
    assert(lca.lca(1, 1) == 1);
    assert(lca.lca(2, 2) == 2);
}

void test_large_star() {
    // Large star graph to test scalability
    LCA<int> lca(0);
    int n = 100;
    for (int i = 1; i <= n; i++) {
        lca.add_edge(0, i);
    }

    lca.preprocess();

    // All leaves should have distance 2 from each other
    assert(lca.lca(1, 50) == 0);
    assert(lca.lca(25, 75) == 0);
    assert(lca.distance(1, 50) == 2);
    assert(lca.distance(25, 100) == 2);
}

void test_long_path() {
    // Very long path to test binary lifting efficiency
    LCA<int> lca(0);
    int n = 64;  // Power of 2 for clean binary lifting
    for (int i = 0; i < n; i++) {
        lca.add_edge(i, i + 1);
    }

    lca.preprocess();

    // Test LCA at various distances
    assert(lca.lca(0, 64) == 0);
    assert(lca.lca(32, 64) == 32);
    assert(lca.lca(16, 48) == 16);

    // Distance should be difference in path positions
    assert(lca.distance(0, 64) == 64);
    assert(lca.distance(16, 48) == 32);
    assert(lca.distance(30, 35) == 5);
}

void test_fibonacci_tree() {
    // Tree based on Fibonacci structure
    LCA<int> lca(1);
    std::vector<std::pair<int, int>> edges = {
        {1, 2}, {1, 3}, {2, 4}, {2, 5}, {3, 6}, {4, 7}, {5, 8}, {5, 9}
    };
    for (const auto& [u, v] : edges) {
        lca.add_edge(u, v);
    }

    lca.preprocess();

    assert(lca.lca(7, 8) == 2);
    assert(lca.lca(7, 6) == 1);
    assert(lca.lca(8, 9) == 5);
    assert(lca.distance(7, 9) == 4);  // 7->4->2->5->9
}

int main() {
    test_linear_chain();
    test_single_node();
    test_binary_tree();
    test_star_tree();
    test_deep_tree();
    test_string_nodes();
```

```
    test_unbalanced_tree();
    test_large_balanced_tree();
    test_complex_tree();
    test_edge_cases();
    test_large_star();
    test_long_path();
    test_fibonacci_tree();
    test_main();
    std::cout << "All LCA tests passed!" << std::endl;
    return 0;
}
```

Polygon Area

polygon_area.cpp

```cpp
/*
Shoelace formula (Gauss's area formula) for computing the area of a polygon.

Computes the area of a simple polygon given its vertices in order (clockwise or
counter-clockwise). Works for both convex and concave polygons.

The formula: Area = ½ |∑(xᵢ × yᵢ₊₁ - xᵢ₊₁ × yᵢ)|

Time complexity: O(n) where n is the number of vertices.
Space complexity: O(1) additional space.
*/

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

struct Point {
    double x, y;
    Point(double x = 0, double y = 0) : x(x), y(y) {}
};

double polygon_area(const std::vector<Point>& vertices) {
    /*
    Calculate the area of a polygon using the Shoelace formula.

    Args:
        vertices: Vector of points in order (clockwise or counter-clockwise)

    Returns:
        The area of the polygon (always positive)
    */
    if (vertices.size() < 3) {
        return 0.0;
    }

    int n = vertices.size();
    double area = 0.0;

    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        area += vertices[i].x * vertices[j].y;
        area -= vertices[j].x * vertices[i].y;
    }

    return std::abs(area) / 2.0;
}

double polygon_signed_area(const std::vector<Point>& vertices) {
    /*
    Calculate the signed area of a polygon.

    Returns positive area for counter-clockwise vertices, negative for clockwise.
    Useful for determining polygon orientation.

    Args:
        vertices: Vector of points in order

    Returns:
        The signed area (positive for CCW, negative for CW)
    */
    if (vertices.size() < 3) {
        return 0.0;
    }

    int n = vertices.size();
```

```cpp
    double area = 0.0;

    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        area += vertices[i].x * vertices[j].y;
        area -= vertices[j].x * vertices[i].y;
    }

    return area / 2.0;
}

bool is_clockwise(const std::vector<Point>& vertices) {
    /* Check if polygon vertices are in clockwise order. */
    return polygon_signed_area(vertices) < 0;
}

void test_main() {
    // Simple square with side length 2
    std::vector<Point> square = {{0.0, 0.0}, {2.0, 0.0}, {2.0, 2.0}, {0.0, 2.0}};
    assert(polygon_area(square) == 4.0);

    // Triangle with base 3 and height 4
    std::vector<Point> triangle = {{0.0, 0.0}, {3.0, 0.0}, {1.5, 4.0}};
    assert(polygon_area(triangle) == 6.0);

    // Test orientation
    std::vector<Point> ccw_square = {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}};
    assert(!is_clockwise(ccw_square));
}

// Don't write tests below during competition.

void test_rectangle() {
    // Rectangle 5 x 3
    std::vector<Point> rect = {{0.0, 0.0}, {5.0, 0.0}, {5.0, 3.0}, {0.0, 3.0}};
    assert(polygon_area(rect) == 15.0);

    // Same rectangle, clockwise order
    std::vector<Point> rect_cw = {{0.0, 0.0}, {0.0, 3.0}, {5.0, 3.0}, {5.0, 0.0}};
    assert(polygon_area(rect_cw) == 15.0);
}

void test_triangle_variations() {
    // Right triangle
    std::vector<Point> tri1 = {{0.0, 0.0}, {4.0, 0.0}, {0.0, 3.0}};
    assert(polygon_area(tri1) == 6.0);

    // Same triangle, different order
    std::vector<Point> tri2 = {{0.0, 3.0}, {0.0, 0.0}, {4.0, 0.0}};
    assert(polygon_area(tri2) == 6.0);

    // Equilateral-ish triangle
    std::vector<Point> tri3 = {{0.0, 0.0}, {2.0, 0.0}, {1.0, 1.732}};
    double area = polygon_area(tri3);
    assert(std::abs(area - 1.732) < 0.01);
}

void test_pentagon() {
    // Regular pentagon (approximate)
    int n = 5;
    double radius = 1.0;
    std::vector<Point> vertices;

    for (int i = 0; i < n; i++) {
        double angle = 2 * M_PI * i / n;
        double x = radius * std::cos(angle);
        double y = radius * std::sin(angle);
        vertices.push_back({x, y});
    }

    double area = polygon_area(vertices);
    // Area of regular pentagon with radius 1
```

```
    double expected = 2.377;  // approximately
    assert(std::abs(area - expected) < 0.01);
}

void test_concave_polygon() {
    // L-shaped polygon (concave)
    std::vector<Point> l_shape = {
        {0.0, 0.0}, {2.0, 0.0}, {2.0, 1.0},
        {1.0, 1.0}, {1.0, 2.0}, {0.0, 2.0}
    };
    // Area = 2x1 rectangle + 1x1 square = 3
    assert(polygon_area(l_shape) == 3.0);
}

void test_degenerate_cases() {
    // Empty polygon
    assert(polygon_area({}) == 0.0);

    // Single point
    assert(polygon_area({{1.0, 1.0}}) == 0.0);

    // Two points (line segment)
    assert(polygon_area({{0.0, 0.0}, {1.0, 1.0}}) == 0.0);
}

void test_floating_point() {
    // Polygon with floating point coordinates
    std::vector<Point> poly = {{0.5, 0.5}, {3.7, 0.5}, {3.7, 2.8}, {0.5, 2.8}};
    double area = polygon_area(poly);
    double expected = (3.7 - 0.5) * (2.8 - 0.5);
    assert(std::abs(area - expected) < 1e-10);
}

void test_signed_area() {
    // Counter-clockwise square (positive area)
    std::vector<Point> ccw = {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}};
    assert(polygon_signed_area(ccw) == 1.0);
    assert(!is_clockwise(ccw));

    // Clockwise square (negative area)
    std::vector<Point> cw = {{0.0, 0.0}, {0.0, 1.0}, {1.0, 1.0}, {1.0, 0.0}};
    assert(polygon_signed_area(cw) == -1.0);
    assert(is_clockwise(cw));
}

void test_large_polygon() {
    // Polygon with many vertices (octagon)
    int n = 8;
    double radius = 5.0;
    std::vector<Point> vertices;

    for (int i = 0; i < n; i++) {
        double angle = 2 * M_PI * i / n;
        double x = radius * std::cos(angle);
        double y = radius * std::sin(angle);
        vertices.push_back({x, y});
    }

    double area = polygon_area(vertices);
    // Area of regular polygon: (n * r^2 * sin(2π/n)) / 2
    double expected = (n * radius * radius * std::sin(2 * M_PI / n)) / 2;
    assert(std::abs(area - expected) < 0.01);
}

void test_negative_coordinates() {
    // Polygon with negative coordinates
    std::vector<Point> poly = {{-2.0, -1.0}, {1.0, -1.0}, {1.0, 2.0}, {-2.0, 2.0}};
    double area = polygon_area(poly);
    double expected = 3.0 * 3.0;
    assert(area == expected);
}
```

```cpp
void test_diamond() {
    // Diamond shape (rhombus)
    std::vector<Point> diamond = {{0.0, 2.0}, {3.0, 0.0}, {0.0, -2.0}, {-3.0, 0.0}};
    double area = polygon_area(diamond);
    // Area = (d1 * d2) / 2 where d1=6, d2=4
    double expected = 12.0;
    assert(area == expected);
}

void test_integer_coordinates() {
    // Ensure integer coordinates work correctly
    std::vector<Point> poly = {{0, 0}, {10, 0}, {10, 5}, {0, 5}};
    double area = polygon_area(poly);
    assert(area == 50.0);
}

int main() {
    test_rectangle();
    test_triangle_variations();
    test_pentagon();
    test_concave_polygon();
    test_degenerate_cases();
    test_floating_point();
    test_signed_area();
    test_large_polygon();
    test_negative_coordinates();
    test_diamond();
    test_integer_coordinates();
    test_main();
    std::cout << "All tests passed!" << std::endl;
    return 0;
}
```

Prefix Tree

prefix_tree.cpp

```cpp
/*
Write-only prefix tree (trie) for efficient string storage and retrieval.

Supports adding strings and finding all strings that are prefixes of a given string.
The tree structure allows for efficient storage of strings with common prefixes.

Time complexity: O(m) for add and find operations, where m is the length of the string.
Space complexity: O(ALPHABET_SIZE * N * M) in the worst case, where N is the number
of strings and M is the average length of strings.
*/

#include <string>
#include <vector>
#include <algorithm>
#include <cassert>
#include <iostream>

class PrefixTree {
private:
    std::vector<std::string> keys;
    std::vector<PrefixTree*> values;

public:
    PrefixTree() {}

    ~PrefixTree() {
        for (auto* child : values) {
            delete child;
        }
    }

    void pp(int indent = 0) {
        // Pretty-print tree structure for debugging
        for (size_t i = 0; i < keys.size(); i++) {
            for (int j = 0; j < indent; j++) std::cout << " ";
            std::cout << keys[i] << ": " << (values[i] == nullptr ? "-" : "") << std::endl;
            if (values[i] != nullptr) {
                values[i]->pp(indent + 2);
            }
        }
    }

    void find_all(const std::string& s, int offset, std::vector<int>& append_to) {
        // Find all strings in tree that are prefixes of s[offset:]. Appends end positions.
        if (!keys.empty() && keys[0] == "") {
            append_to.push_back(offset);
        }
        if (offset >= s.length()) {
            return;
        }
        std::string target_char = s.substr(offset, 1);
        auto it = std::lower_bound(keys.begin(), keys.end(), target_char);
        int index = it - keys.begin();
        if (index == keys.size()) {
            return;
        }
        std::string key_substr = s.substr(offset, keys[index].length());
        if (key_substr == keys[index]) {
            PrefixTree* pt = values[index];
            if (pt == nullptr) {
                append_to.push_back(offset + keys[index].length());
            } else {
                pt->find_all(s, offset + keys[index].length(), append_to);
            }
        }
    }
```

```
    int max_len() {
        // Return length of longest string in tree
        int result = 0;
        for (size_t i = 0; i < keys.size(); i++) {
            result = std::max(result, (int)keys[i].length() + (values[i] == nullptr ? 0 : values[i]-
>max_len()));
        }
        return result;
    }

    void add(const std::string& s) {
        // Add string to tree
        if (s.empty() || keys.empty()) {
            keys.insert(keys.begin(), s);
            values.insert(values.begin(), nullptr);
            return;
        }

        auto it = std::lower_bound(keys.begin(), keys.end(), s);
        int pos = it - keys.begin();
        if (pos > 0 && !keys[pos - 1].empty() && keys[pos - 1][0] == s[0]) {
            pos--;
        }
        if (pos < keys.size() && !keys[pos].empty() && keys[pos][0] == s[0]) {
            // Merge
            if (s.find(keys[pos]) == 0 && s.length() >= keys[pos].length()) {
                // s starts with keys[pos]
                PrefixTree* pt = values[pos];
                if (pt == nullptr) {
                    PrefixTree* child = new PrefixTree();
                    child->keys.push_back("");
                    child->values.push_back(nullptr);
                    values[pos] = pt = child;
                }
                pt->add(s.substr(keys[pos].length()));
            } else if (keys[pos].find(s) == 0 && keys[pos].length() >= s.length()) {
                // keys[pos] starts with s
                PrefixTree* child = new PrefixTree();
                child->keys.push_back("");
                child->values.push_back(nullptr);
                child->keys.push_back(keys[pos].substr(s.length()));
                child->values.push_back(values[pos]);
                keys[pos] = s;
                values[pos] = child;
            } else {
                // Find common prefix
                int prefix = 1;
                while (prefix < s.length() && prefix < keys[pos].length() && s[prefix] == keys[pos]
[prefix]) {
                    prefix++;
                }
                PrefixTree* child = new PrefixTree();
                if (s < keys[pos]) {
                    child->keys.push_back(s.substr(prefix));
                    child->values.push_back(nullptr);
                }
                child->keys.push_back(keys[pos].substr(prefix));
                child->values.push_back(values[pos]);
                if (s >= keys[pos]) {
                    child->keys.push_back(s.substr(prefix));
                    child->values.push_back(nullptr);
                }
                keys[pos] = s.substr(0, prefix);
                values[pos] = child;
            }
        } else {
            keys.insert(keys.begin() + pos, s);
            values.insert(values.begin() + pos, nullptr);
        }
    }
};
```

```
void test_main() {
    PrefixTree p;
    p.add("cat");
    p.add("car");
    p.add("card");
    std::vector<int> l;
    p.find_all("card", 0, l);
    assert(l.size() == 2 && l[0] == 3 && l[1] == 4);
    assert(p.max_len() == 4);
}

// Don't write tests below during competition.

void test_empty_tree() {
    PrefixTree p;
    std::vector<int> l;
    p.find_all("test", 0, l);
    assert(l.empty());
    assert(p.max_len() == 0);
}

void test_single_string() {
    PrefixTree p;
    p.add("hello");
    std::vector<int> l;
    p.find_all("hello world", 0, l);
    assert(l.size() == 1);
    assert(l[0] == 5);
    assert(p.max_len() == 5);
}

void test_empty_string() {
    PrefixTree p;
    p.add("");
    std::vector<int> l;
    p.find_all("anything", 0, l);
    assert(l.size() == 1);
    assert(l[0] == 0);  // Empty string matches at position 0
}

void test_no_match() {
    PrefixTree p;
    p.add("cat");
    p.add("car");
    std::vector<int> l;
    p.find_all("dog", 0, l);
    assert(l.empty());
}

void test_partial_match() {
    PrefixTree p;
    p.add("catalog");
    std::vector<int> l;
    p.find_all("cat", 0, l);
    assert(l.empty());  // "catalog" is not a prefix of "cat"
}

void test_overlapping_strings() {
    PrefixTree p;
    p.add("a");
    p.add("ab");
    p.add("abc");
    std::vector<int> l;
    p.find_all("abcdef", 0, l);
    assert(l.size() == 3);
    assert(l[0] == 1);
    assert(l[1] == 2);
    assert(l[2] == 3);
}

void test_different_offsets() {
```

```
    PrefixTree p;
    p.add("test");
    std::vector<int> l;
    p.find_all("xxtest", 2, l);
    assert(l.size() == 1);
    assert(l[0] == 6);  // "test" found starting at offset 2, ends at 6
}

void test_multiple_words() {
    PrefixTree p;
    std::vector<std::string> words = {"the", "then", "there", "answer", "any", "by", "bye", "their"};
    for (const auto& word : words) {
        p.add(word);
    }

    std::vector<int> l;
    p.find_all("their", 0, l);
    // "the", "their" are prefixes of "their"
    bool found_3 = false, found_5 = false;
    for (int pos : l) {
        if (pos == 3) found_3 = true;
        if (pos == 5) found_5 = true;
    }
    assert(found_3 && found_5);
}

void test_common_prefix() {
    PrefixTree p;
    p.add("pre");
    p.add("prefix");
    p.add("prepare");

    std::vector<int> l;
    p.find_all("prefix", 0, l);
    assert(l.size() == 2);
    assert(l[0] == 3);   // "pre"
    assert(l[1] == 6);   // "prefix"
}

void test_max_len() {
    PrefixTree p;
    assert(p.max_len() == 0);

    p.add("a");
    assert(p.max_len() == 1);

    p.add("abc");
    assert(p.max_len() == 3);

    p.add("ab");
    assert(p.max_len() == 3);
}

void test_duplicate_add() {
    PrefixTree p;
    p.add("test");
    p.add("test");  // Add same string again

    std::vector<int> l;
    p.find_all("test", 0, l);
    // Should still work correctly
    bool found_4 = false;
    for (int pos : l) {
        if (pos == 4) found_4 = true;
    }
    assert(found_4);
}

int main() {
    test_empty_tree();
    test_single_string();
    test_empty_string();
```

```
    test_no_match();
    test_partial_match();
    test_overlapping_strings();
    test_different_offsets();
    test_multiple_words();
    test_common_prefix();
    test_max_len();
    test_duplicate_add();
    test_main();
    std::cout << "All Prefix Tree tests passed!" << std::endl;
    return 0;
}
```

Priority Queue

priority_queue.cpp

```cpp
/*
Priority queue implementation using a binary heap.

This module provides a generic priority queue that supports adding items with priorities,
updating priorities, removing items, and popping the item with the lowest priority.
The implementation uses C++ std::priority_queue for efficient heap operations.

Time complexity: O(log n) for add/update and pop operations, O(log n) for remove.
Space complexity: O(n) where n is the number of items in the queue.
*/

#include <queue>
#include <unordered_map>
#include <functional>
#include <cassert>
#include <iostream>
#include <stdexcept>
#include <utility>

template<typename KeyT, typename PriorityT>
class PriorityQueue {
private:
    struct Entry {
        PriorityT priority;
        KeyT key;
        size_t version;

        Entry(PriorityT p, KeyT k, size_t v) : priority(p), key(k), version(v) {}

        // For min-heap behavior (lowest priority first)
        bool operator>(const Entry& other) const {
            return priority > other.priority;
        }
    };

    std::priority_queue<Entry, std::vector<Entry>, std::greater<Entry>> pq;
    std::unordered_map<KeyT, size_t> key_versions;  // Maps keys to their current version
    size_t next_version;
    int size_count;

public:
    PriorityQueue() : next_version(0), size_count(0) {}

    void set(const KeyT& key, const PriorityT& priority) {
        // Add a new task or update the priority of an existing task
        if (key_versions.find(key) != key_versions.end()) {
            // Key exists, this will invalidate the old entry
            size_count--; // We'll increment it back below
        }

        size_count++;
        size_t version = next_version++;
        key_versions[key] = version;
        pq.push(Entry(priority, key, version));
    }

    void remove(const KeyT& key) {
        // Mark an existing task as removed by updating its version
        auto it = key_versions.find(key);
        if (it == key_versions.end()) {
            throw std::runtime_error("Key not found in priority queue");
        }
        key_versions.erase(it);
        size_count--;
    }
```

```
    std::pair<KeyT, PriorityT> pop() {
        // Remove and return the lowest priority task. Throw exception if empty.
        while (!pq.empty()) {
            Entry top = pq.top();
            pq.pop();

            // Check if this entry is still valid (not removed/updated)
            auto it = key_versions.find(top.key);
            if (it != key_versions.end() && it->second == top.version) {
                key_versions.erase(it);
                size_count--;
                return std::make_pair(top.key, top.priority);
            }
        }
        throw std::runtime_error("pop from an empty priority queue");
    }

    std::pair<KeyT, PriorityT> peek() {
        // Return the lowest priority task without removing. Returns empty result throws if empty.
        while (!pq.empty()) {
            Entry top = pq.top();

            // Check if this entry is still valid (not removed/updated)
            auto it = key_versions.find(top.key);
            if (it != key_versions.end() && it->second == top.version) {
                return std::make_pair(top.key, top.priority);
            }
            // Remove the invalid entry from the top
            pq.pop();
        }
        throw std::runtime_error("peek from an empty priority queue");
    }

    bool contains(const KeyT& key) const {
        return key_versions.find(key) != key_versions.end();
    }

    int size() const {
        return size_count;
    }

    bool empty() const {
        return size_count == 0;
    }
};

void test_main() {
    PriorityQueue<std::string, int> p;
    p.set("x", 15);
    p.set("y", 23);
    p.set("z", 8);
    auto peek_result = p.peek();
    assert(peek_result.first == "z" && peek_result.second == 8);
    auto result1 = p.pop();
    assert(result1.first == "z" && result1.second == 8);
    auto result2 = p.pop();
    assert(result2.first == "x" && result2.second == 15);
}

// Don't write tests below during competition.

void test_basic_operations() {
    PriorityQueue<std::string, int> pq;

    // Test empty queue
    assert(pq.empty());
    assert(pq.size() == 0);

    // Add items
    pq.set("task1", 10);
    pq.set("task2", 5);
    pq.set("task3", 15);
```

```
    assert(pq.size() == 3);
    auto peek_result = pq.peek();
    assert(peek_result.first == "task2" && peek_result.second == 5);

    // Pop in priority order
    auto item1 = pq.pop();
    assert(item1.first == "task2" && item1.second == 5);
    assert(pq.size() == 2);

    auto item2 = pq.pop();
    assert(item2.first == "task1" && item2.second == 10);

    auto item3 = pq.pop();
    assert(item3.first == "task3" && item3.second == 15);

    assert(pq.size() == 0);
}

void test_update_priority() {
    PriorityQueue<std::string, int> pq;

    pq.set("task1", 10);
    pq.set("task2", 5);

    // Update task1 to have higher priority
    pq.set("task1", 3);
    assert(pq.peek().first == "task1");
    assert(pq.peek().second == 3);
    assert(pq.size() == 2);

    // Pop should now give task1 first
    auto item1 = pq.pop();
    assert(item1.first == "task1" && item1.second == 3);

    auto item2 = pq.pop();
    assert(item2.first == "task2" && item2.second == 5);
}

void test_remove() {
    PriorityQueue<std::string, int> pq;

    pq.set("task1", 10);
    pq.set("task2", 5);
    pq.set("task3", 15);

    // Remove middle priority task
    pq.remove("task1");
    assert(pq.size() == 2);
    assert(!pq.contains("task1"));

    // Verify correct items remain
    auto item1 = pq.pop();
    assert(item1.first == "task2" && item1.second == 5);

    auto item2 = pq.pop();
    assert(item2.first == "task3" && item2.second == 15);
}

void test_contains() {
    PriorityQueue<std::string, int> pq;

    pq.set("task1", 10);
    pq.set("task2", 5);

    assert(pq.contains("task1"));
    assert(pq.contains("task2"));
    assert(!pq.contains("task3"));

    pq.remove("task1");
    assert(!pq.contains("task1"));
}
```

```cpp
void test_empty_operations() {
    PriorityQueue<std::string, int> pq;

    // Test pop on empty queue
    bool caught = false;
    try {
        pq.pop();
    } catch (const std::runtime_error&) {
        caught = true;
    }
    assert(caught);

    // Test peek on empty queue
    caught = false;
    try {
        pq.peek();
    } catch (const std::runtime_error&) {
        caught = true;
    }
    assert(caught);
}

void test_remove_nonexistent() {
    PriorityQueue<std::string, int> pq;

    pq.set("task1", 10);

    bool caught = false;
    try {
        pq.remove("nonexistent");
    } catch (const std::runtime_error&) {
        caught = true;
    }
    assert(caught);
}

void test_single_element() {
    PriorityQueue<std::string, int> pq;

    pq.set("only", 42);
    assert(pq.size() == 1);
    assert(pq.peek().first == "only" && pq.peek().second == 42);
    assert(pq.pop().first == "only");
    assert(pq.size() == 0);
}

void test_duplicate_priorities() {
    PriorityQueue<std::string, int> pq;

    pq.set("task1", 10);
    pq.set("task2", 10);
    pq.set("task3", 10);

    assert(pq.size() == 3);

    // All should pop eventually
    std::vector<std::pair<std::string, int>> results;
    results.push_back(pq.pop());
    results.push_back(pq.pop());
    results.push_back(pq.pop());

    assert(results.size() == 3);
    for (const auto& [key, priority] : results) {
        assert(priority == 10);
    }
}

void test_with_floats() {
    PriorityQueue<std::string, double> pq;

    pq.set("a", 1.5);
```

```
    pq.set("b", 0.5);
    pq.set("c", 2.3);

    auto item1 = pq.pop();
    assert(item1.first == "b" && item1.second == 0.5);

    auto item2 = pq.pop();
    assert(item2.first == "a" && item2.second == 1.5);

    auto item3 = pq.pop();
    assert(item3.first == "c" && item3.second == 2.3);
}

int main() {
    test_basic_operations();
    test_update_priority();
    test_remove();
    test_contains();
    test_empty_operations();
    test_remove_nonexistent();
    test_single_element();
    test_duplicate_priorities();
    test_with_floats();
    test_main();
    std::cout << "All Priority Queue tests passed!" << std::endl;
    return 0;
}
```

Segment Tree

segment_tree.cpp

```cpp
/*
Segment tree for efficient range queries and updates.

Supports range sum queries, point updates, and can be easily modified for other operations
like range minimum, maximum, or more complex functions. The tree uses 1-indexed array
representation with lazy propagation for range updates.

Time complexity: O(log n) for query and update operations, O(n) for construction.
Space complexity: O(n) for the tree structure.
*/

#include <iostream>
#include <vector>
#include <stdexcept>
#include <cassert>
#include <numeric>
#include <string>

template<typename T>
class SegmentTree {
private:
    int n;
    T zero;
    std::vector<T> tree;

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

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

    T query_helper(int node, int start, int end, int left, int right) {
        if (right < start || left > end) {
            return zero;
        }
        if (left <= start && end <= right) {
            return tree[node];
        }
        int mid = (start + end) / 2;
        T left_sum = query_helper(2 * node, start, mid, left, right);
        T right_sum = query_helper(2 * node + 1, mid + 1, end, left, right);
        return left_sum + right_sum;
    }

public:
    SegmentTree(const std::vector<T>& arr, T zero) : n(arr.size()), zero(zero) {
        tree.resize(4 * n, zero);
```

```
        if (!arr.empty()) {
            build(arr, 1, 0, n - 1);
        }
    }

    void update(int idx, T val) {
        if (idx < 0 || idx >= n) {
            throw std::out_of_range("Index " + std::to_string(idx) + " out of bounds for size " +
std::to_string(n));
        }
        update_helper(1, 0, n - 1, idx, val);
    }

    T query(int left, int right) {
        if (left < 0 || right >= n || left > right) {
            throw std::out_of_range("Invalid range [" + std::to_string(left) + ", " +
std::to_string(right) +
                                    "] for size " + std::to_string(n));
        }
        return query_helper(1, 0, n - 1, left, right);
    }
};

void test_main() {
    SegmentTree<int> st({1, 3, 5, 7, 9}, 0);
    assert(st.query(1, 3) == 15);
    st.update(2, 10);
    assert(st.query(1, 3) == 20);
    assert(st.query(0, 4) == 30);
}

// Don't write tests below during competition.

void test_large_array() {
    // Test with large array
    std::vector<int> arr(1000);
    std::iota(arr.begin(), arr.end(), 0);
    SegmentTree<int> st(arr, 0);

    // Test various range queries
    int sum_0_99 = 0;
    for (int i = 0; i < 100; i++) sum_0_99 += i;
    assert(st.query(0, 99) == sum_0_99);

    int sum_500_599 = 0;
    for (int i = 500; i < 600; i++) sum_500_599 += i;
    assert(st.query(500, 599) == sum_500_599);

    assert(st.query(999, 999) == 999);

    // Test updates on large array
    st.update(500, 9999);
    assert(st.query(500, 500) == 9999);
    assert(st.query(499, 501) == 499 + 9999 + 501);
}

void test_edge_cases() {
    // Single element
    SegmentTree<int> st({42}, 0);
    assert(st.query(0, 0) == 42);
    st.update(0, 100);
    assert(st.query(0, 0) == 100);

    // Empty array - skip test as it would throw on query

    // All zeros
    SegmentTree<int> st_zeros({0, 0, 0, 0}, 0);
    assert(st_zeros.query(0, 3) == 0);
    st_zeros.update(2, 5);
    assert(st_zeros.query(0, 3) == 5);
}
```

```
void test_single_point_queries() {
    SegmentTree<int> st({10, 20, 30, 40, 50}, 0);

    // Query single points
    assert(st.query(0, 0) == 10);
    assert(st.query(1, 1) == 20);
    assert(st.query(2, 2) == 30);
    assert(st.query(3, 3) == 40);
    assert(st.query(4, 4) == 50);

    // Update and requery
    st.update(2, 100);
    assert(st.query(2, 2) == 100);
    assert(st.query(0, 4) == 220);
}

void test_full_range() {
    SegmentTree<int> st({1, 2, 3, 4, 5}, 0);

    // Query entire range
    assert(st.query(0, 4) == 15);

    // Update and query entire range again
    st.update(0, 10);
    assert(st.query(0, 4) == 24);
}

void test_overlapping_ranges() {
    SegmentTree<int> st({1, 2, 3, 4, 5, 6, 7, 8}, 0);

    // Various overlapping ranges
    assert(st.query(0, 3) == 10);
    assert(st.query(2, 5) == 18);
    assert(st.query(4, 7) == 26);

    st.update(3, 100);
    assert(st.query(0, 3) == 106);
    assert(st.query(2, 5) == 114);
}

void test_negative_numbers() {
    SegmentTree<int> st({-5, -3, -1, 1, 3, 5}, 0);

    assert(st.query(0, 5) == 0);
    assert(st.query(0, 2) == -9);
    assert(st.query(3, 5) == 9);

    st.update(2, 10);
    assert(st.query(0, 5) == 11);
}

void test_string_concatenation() {
    // Test with strings (using + for concatenation)
    SegmentTree<std::string> st({"a", "b", "c", "d"}, "");

    assert(st.query(0, 3) == "abcd");
    assert(st.query(1, 2) == "bc");

    st.update(1, "X");
    assert(st.query(0, 3) == "aXcd");
    assert(st.query(0, 1) == "aX");
}

void test_multiple_updates() {
    SegmentTree<int> st({1, 1, 1, 1, 1}, 0);

    assert(st.query(0, 4) == 5);

    // Multiple updates
    st.update(0, 2);
    st.update(1, 2);
    st.update(2, 2);
```

```
    st.update(3, 2);
    st.update(4, 2);

    assert(st.query(0, 4) == 10);
    assert(st.query(1, 3) == 6);
}

void test_invalid_indices() {
    SegmentTree<int> st({1, 2, 3}, 0);

    // Test invalid indices
    bool caught = false;
    try {
        st.update(-1, 5);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        st.update(3, 5);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        st.query(-1, 2);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        st.query(0, 3);
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);

    caught = false;
    try {
        st.query(2, 1);  // left > right
    } catch (const std::out_of_range&) {
        caught = true;
    }
    assert(caught);
}

int main() {
    test_large_array();
    test_edge_cases();
    test_single_point_queries();
    test_full_range();
    test_overlapping_ranges();
    test_negative_numbers();
    test_string_concatenation();
    test_multiple_updates();
    test_invalid_indices();
    test_main();
    std::cout << "All Segment Tree tests passed!" << std::endl;
    return 0;
}
```

Topological Sort

topological_sort.cpp

```cpp
/*
Topological sorting for Directed Acyclic Graphs (DAGs).

Produces a linear ordering of vertices such that for every directed edge (u, v),
vertex u comes before v in the ordering. Uses both DFS-based and Kahn's algorithm
(BFS-based) approaches for different use cases.

Time complexity: O(V + E) for both algorithms, where V is vertices and E is edges.
Space complexity: O(V + E) for the graph representation and auxiliary data structures.
*/

#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <algorithm>
#include <optional>
#include <stdexcept>
#include <cassert>

template<typename NodeT>
class TopologicalSort {
private:
    std::map<NodeT, std::vector<NodeT>> graph;
    std::map<NodeT, int> in_degree;

    enum Color { WHITE, GRAY, BLACK };

    bool dfs_helper(NodeT node, std::map<NodeT, Color>& color, std::vector<NodeT>& result) {
        if (color[node] == GRAY) {  // Back edge (cycle)
            return false;
        }
        if (color[node] == BLACK) {  // Already processed
            return true;
        }

        color[node] = GRAY;
        for (const auto& neighbor : graph[node]) {
            if (!dfs_helper(neighbor, color, result)) {
                return false;
            }
        }

        color[node] = BLACK;
        result.push_back(node);
        return true;
    }

public:
    void add_edge(NodeT u, NodeT v) {
        if (graph.find(u) == graph.end()) {
            graph[u] = {};
            in_degree[u] = 0;
        }
        if (in_degree.find(v) == in_degree.end()) {
            in_degree[v] = 0;
            graph[v] = {};
        }

        graph[u].push_back(v);
        in_degree[v]++;
    }

    std::optional<std::vector<NodeT>> kahn_sort() {
        /*
        Topological sort using Kahn's algorithm (BFS-based).
```

```
    Returns the topological ordering, or nullopt if the graph has a cycle.
    */
    std::map<NodeT, int> in_deg = in_degree;
    std::queue<NodeT> q;

    for (const auto& [node, deg] : in_deg) {
        if (deg == 0) {
            q.push(node);
        }
    }

    std::vector<NodeT> result;

    while (!q.empty()) {
        NodeT node = q.front();
        q.pop();
        result.push_back(node);

        for (const auto& neighbor : graph[node]) {
            in_deg[neighbor]--;
            if (in_deg[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    // Check if all nodes are processed (no cycle)
    if (result.size() != in_degree.size()) {
        return std::nullopt;
    }

    return result;
}

std::optional<std::vector<NodeT>> dfs_sort() {
    /*
    Topological sort using DFS.

    Returns the topological ordering, or nullopt if the graph has a cycle.
    */
    std::map<NodeT, Color> color;
    for (const auto& [node, _] : in_degree) {
        color[node] = WHITE;
    }

    std::vector<NodeT> result;

    for (const auto& [node, _] : in_degree) {
        if (color[node] == WHITE && !dfs_helper(node, color, result)) {
            return std::nullopt;
        }
    }

    std::reverse(result.begin(), result.end());
    return result;
}

bool has_cycle() {
    return !kahn_sort().has_value();
}

std::map<NodeT, int> longest_path() {
    /*
    Find longest path from each node in the DAG.

    Returns a map from each node to its longest path length.
    */
    auto topo_order = kahn_sort();
    if (!topo_order.has_value()) {
        throw std::runtime_error("Graph contains a cycle");
    }
```

```
        std::map<NodeT, int> dist;
        for (const auto& [node, _] : in_degree) {
            dist[node] = 0;
        }

        for (const auto& node : topo_order.value()) {
            for (const auto& neighbor : graph[node]) {
                dist[neighbor] = std::max(dist[neighbor], dist[node] + 1);
            }
        }

        return dist;
    }
};

void test_main() {
    TopologicalSort<int> ts;
    std::vector<std::pair<int, int>> edges = {{5, 2}, {5, 0}, {4, 0}, {4, 1}, {2, 3}, {3, 1}};
    for (const auto& [u, v] : edges) {
        ts.add_edge(u, v);
    }

    auto kahn_result = ts.kahn_sort();
    auto dfs_result = ts.dfs_sort();

    assert(kahn_result.has_value());
    assert(dfs_result.has_value());
    assert(!ts.has_cycle());

    // Test with cycle
    TopologicalSort<int> ts_cycle;
    ts_cycle.add_edge(1, 2);
    ts_cycle.add_edge(2, 3);
    ts_cycle.add_edge(3, 1);
    assert(ts_cycle.has_cycle());
}

// Don't write tests below during competition.

void test_empty_graph() {
    TopologicalSort<int> ts;

    // Empty graph should return empty list
    assert(ts.kahn_sort().value().empty());
    assert(ts.dfs_sort().value().empty());
    assert(!ts.has_cycle());
}

void test_single_node_self_loop() {
    TopologicalSort<std::string> ts;
    ts.add_edge("A", "A");  // This creates a self-loop (cycle)

    assert(ts.has_cycle());
    assert(!ts.kahn_sort().has_value());
    assert(!ts.dfs_sort().has_value());
}

void test_single_node_no_edges() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 2);  // Add a simple edge to create node

    TopologicalSort<int> ts2;
    // Can't easily test single isolated node, skip this case
}

void test_linear_chain() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 2);
    ts.add_edge(2, 3);
    ts.add_edge(3, 4);
    ts.add_edge(4, 5);
```

```
    auto kahn_result = ts.kahn_sort();
    auto dfs_result = ts.dfs_sort();

    assert(kahn_result.has_value());
    assert(dfs_result.has_value());
    assert(!ts.has_cycle());

    // Check ordering
    auto kahn = kahn_result.value();
    assert(kahn.size() == 5);
    // Verify 1 comes before 2, 2 before 3, etc.
    for (size_t i = 0; i < kahn.size() - 1; i++) {
        assert(kahn[i] < kahn[i + 1]);
    }
}

void test_multiple_sources() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 3);
    ts.add_edge(2, 3);
    ts.add_edge(3, 4);

    auto result = ts.kahn_sort();
    assert(result.has_value());
    assert(!ts.has_cycle());

    // Both 1 and 2 should come before 3
    auto vec = result.value();
    auto pos_1 = std::find(vec.begin(), vec.end(), 1) - vec.begin();
    auto pos_2 = std::find(vec.begin(), vec.end(), 2) - vec.begin();
    auto pos_3 = std::find(vec.begin(), vec.end(), 3) - vec.begin();
    auto pos_4 = std::find(vec.begin(), vec.end(), 4) - vec.begin();

    assert(pos_1 < pos_3);
    assert(pos_2 < pos_3);
    assert(pos_3 < pos_4);
}

void test_diamond_shape() {
    TopologicalSort<std::string> ts;
    ts.add_edge("A", "B");
    ts.add_edge("A", "C");
    ts.add_edge("B", "D");
    ts.add_edge("C", "D");

    auto kahn = ts.kahn_sort();
    auto dfs = ts.dfs_sort();

    assert(kahn.has_value());
    assert(dfs.has_value());
    assert(!ts.has_cycle());

    // A should come first, D should come last
    auto kahn_vec = kahn.value();
    assert(kahn_vec.front() == "A");
    assert(kahn_vec.back() == "D");
}

void test_complex_cycle() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 2);
    ts.add_edge(2, 3);
    ts.add_edge(3, 4);
    ts.add_edge(4, 2);  // Creates cycle 2->3->4->2

    assert(ts.has_cycle());
    assert(!ts.kahn_sort().has_value());
    assert(!ts.dfs_sort().has_value());
}

void test_disconnected_components() {
```

```
    TopologicalSort<int> ts;
    // Component 1
    ts.add_edge(1, 2);
    ts.add_edge(2, 3);
    // Component 2
    ts.add_edge(4, 5);
    ts.add_edge(5, 6);

    auto result = ts.kahn_sort();
    assert(result.has_value());
    assert(!ts.has_cycle());
    assert(result.value().size() == 6);
}

void test_longest_path() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 2);
    ts.add_edge(1, 3);
    ts.add_edge(2, 4);
    ts.add_edge(3, 4);
    ts.add_edge(4, 5);

    auto dist = ts.longest_path();

    assert(dist[1] == 0);
    assert(dist[5] == 3);  // Longest path: 1->2->4->5 or 1->3->4->5
}

void test_longest_path_with_cycle() {
    TopologicalSort<int> ts;
    ts.add_edge(1, 2);
    ts.add_edge(2, 3);
    ts.add_edge(3, 1);

    bool caught = false;
    try {
        ts.longest_path();
    } catch (const std::runtime_error&) {
        caught = true;
    }
    assert(caught);
}

void test_comparison_kahn_vs_dfs() {
    TopologicalSort<int> ts;
    ts.add_edge(5, 2);
    ts.add_edge(5, 0);
    ts.add_edge(4, 0);
    ts.add_edge(4, 1);
    ts.add_edge(2, 3);
    ts.add_edge(3, 1);

    auto kahn = ts.kahn_sort();
    auto dfs = ts.dfs_sort();

    assert(kahn.has_value());
    assert(dfs.has_value());

    // Both should produce valid topological orderings
    // (may differ, but both should be valid)
}

void test_large_graph() {
    TopologicalSort<int> ts;

    // Create a chain of 1000 nodes
    for (int i = 0; i < 999; i++) {
        ts.add_edge(i, i + 1);
    }

    auto result = ts.kahn_sort();
    assert(result.has_value());
```

```
    assert(result.value().size() == 1000);
    assert(!ts.has_cycle());
}

void test_string_nodes() {
    TopologicalSort<std::string> ts;
    ts.add_edge("undershirt", "shirt");
    ts.add_edge("pants", "belt");
    ts.add_edge("shirt", "belt");
    ts.add_edge("belt", "jacket");
    ts.add_edge("socks", "shoes");
    ts.add_edge("pants", "shoes");

    auto result = ts.kahn_sort();
    assert(result.has_value());
    assert(!ts.has_cycle());
}

int main() {
    test_empty_graph();
    test_single_node_self_loop();
    test_linear_chain();
    test_multiple_sources();
    test_diamond_shape();
    test_complex_cycle();
    test_disconnected_components();
    test_longest_path();
    test_longest_path_with_cycle();
    test_comparison_kahn_vs_dfs();
    test_large_graph();
    test_string_nodes();
    test_main();
    std::cout << "All Topological Sort tests passed!" << std::endl;
    return 0;
}
```

Union Find

union_find.cpp

```cpp
/*
Union-find (disjoint-set union, DSU) maintains a collection of disjoint sets under two operations:

* find(x): return the representative (root) of the set containing x.
* union(x, y): merge the sets containing x and y.

Time complexity: O(alpha(n)) per operation with path compression and union by rank,
where alpha is the inverse Ackermann function (effectively constant for practical purposes).
*/

#include <cassert>
#include <iostream>
#include <set>

class UnionFind {
public:
    UnionFind* parent;
    int rank;

    UnionFind() : parent(this), rank(0) {}

    virtual void merge(UnionFind* other) {
        // Override with desired functionality
    }

    UnionFind* find() {
        if (parent == this) {
            return this;
        }
        parent = parent->find();
        return parent;
    }

    UnionFind* union_with(UnionFind* other) {
        UnionFind* x = this->find();
        UnionFind* y = other->find();
        if (x == y) {
            return x;
        }
        if (x->rank < y->rank) {
            x->parent = y;
            y->merge(x);
            return y;
        }
        if (x->rank > y->rank) {
            y->parent = x;
            x->merge(y);
            return x;
        }
        x->parent = y;
        y->merge(x);
        y->rank++;
        return y;
    }
};

class Test : public UnionFind {
public:
    int size;

    Test() : UnionFind(), size(1) {}

    void merge(UnionFind* other) override {
        Test* other_test = static_cast<Test*>(other);
        this->size += other_test->size;
    }
```

```
};

void test_main() {
    Test* a = new Test();
    Test* b = new Test();
    Test* c = new Test();
    Test* d = static_cast<Test*>(a->union_with(b));
    Test* e = static_cast<Test*>(d->union_with(c));
    assert(static_cast<Test*>(e->find())->size == 3);
    assert(static_cast<Test*>(a->find())->size == 3);

    delete a;
    delete b;
    delete c;
}

// Don't write tests below during competition.

void test_single_element() {
    Test* a = new Test();
    assert(a->find() == a);
    assert(a->size == 1);
    delete a;
}

void test_union_same_set() {
    Test* a = new Test();
    Test* b = new Test();
    a->union_with(b);
    // Unioning again should be safe
    Test* root = static_cast<Test*>(a->union_with(b));
    assert(a->find() == b->find());
    assert(root->size == 2);
    delete a;
    delete b;
}

void test_multiple_unions() {
    Test* nodes[10];
    for (int i = 0; i < 10; i++) {
        nodes[i] = new Test();
    }

    // Chain union: 0-1-2-3-4-5-6-7-8-9
    for (int i = 0; i < 9; i++) {
        nodes[i]->union_with(nodes[i + 1]);
    }

    // All should have same root
    UnionFind* root = nodes[0]->find();
    for (int i = 0; i < 10; i++) {
        assert(nodes[i]->find() == root);
    }

    assert(static_cast<Test*>(root)->size == 10);

    for (int i = 0; i < 10; i++) {
        delete nodes[i];
    }
}

void test_union_order_independence() {
    // Test that union order doesn't affect final result
    Test* a1 = new Test();
    Test* b1 = new Test();
    Test* c1 = new Test();
    a1->union_with(b1)->union_with(c1);
    Test* root1 = static_cast<Test*>(a1->find());

    Test* a2 = new Test();
    Test* b2 = new Test();
    Test* c2 = new Test();
```

```
    c2->union_with(b2)->union_with(a2);
    Test* root2 = static_cast<Test*>(a2->find());

    assert(root1->size == 3);
    assert(root2->size == 3);

    delete a1; delete b1; delete c1;
    delete a2; delete b2; delete c2;
}

void test_disconnected_sets() {
    // Create two separate sets
    Test* a = new Test();
    Test* b = new Test();
    Test* c = new Test();
    Test* d = new Test();

    a->union_with(b);
    c->union_with(d);

    assert(a->find() == b->find());
    assert(c->find() == d->find());
    assert(a->find() != c->find());

    assert(static_cast<Test*>(a->find())->size == 2);
    assert(static_cast<Test*>(c->find())->size == 2);

    delete a; delete b; delete c; delete d;
}

void test_large_set() {
    // Create a large union-find structure
    Test* nodes[100];
    for (int i = 0; i < 100; i++) {
        nodes[i] = new Test();
    }

    // Union in pairs
    for (int i = 0; i < 100; i += 2) {
        nodes[i]->union_with(nodes[i + 1]);
    }

    // Now we have 50 sets of size 2
    std::set<UnionFind*> roots;
    for (int i = 0; i < 100; i++) {
        roots.insert(nodes[i]->find());
    }
    assert(roots.size() == 50);

    // Union all pairs together
    for (int i = 0; i < 100; i += 4) {
        if (i + 2 < 100) {
            nodes[i]->union_with(nodes[i + 2]);
        }
    }

    // Now we have 25 sets of size 4
    roots.clear();
    for (int i = 0; i < 100; i++) {
        roots.insert(nodes[i]->find());
    }
    assert(roots.size() == 25);

    for (int i = 0; i < 100; i++) {
        delete nodes[i];
    }
}

int main() {
    test_single_element();
    test_union_same_set();
    test_multiple_unions();
```

```
    test_union_order_independence();
    test_disconnected_sets();
    test_large_set();
    test_main();
    std::cout << "All tests passed!" << std::endl;
    return 0;
}
```