Algorithm Reference - Cpp

Bellman Ford

bellman_ford.cpp

```cpp
/*
Bellman-Ford algorithm for single-source shortest paths with negative edge weights.

Time complexity: O(VE) where V is vertices and E is edges.
Space complexity: O(V + E).
*/

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

template <typename NodeT, typename WeightT>
class BellmanFord {
  private:
    struct Edge {
        NodeT from, to;
        WeightT weight;
    };

    std::vector<Edge> edges;
    std::set<NodeT> nodes;
    WeightT infinity;

  public:
    BellmanFord(WeightT infinity) : infinity(infinity) {}

    void add_edge(NodeT u, NodeT v, WeightT weight) {
        edges.push_back({u, v, weight});
        nodes.insert(u);
        nodes.insert(v);
    }

    std::optional<std::map<NodeT, WeightT>> shortest_paths(NodeT source) {
        std::map<NodeT, WeightT> distances;
        for (const auto& node : nodes) { distances[node] = infinity; }
        distances[source] = 0;

        for (size_t i = 0; i < nodes.size() - 1; i++) {
            for (const auto& e : edges) {
                if (distances[e.from] != infinity &&
                    distances[e.from] + e.weight < distances[e.to]) {
                    distances[e.to] = distances[e.from] + e.weight;
                }
            }
        }

        for (const auto& e : edges) {
            if (distances[e.from] != infinity && distances[e.from] + e.weight < distances[e.to]) {
                return std::nullopt;
            }
        }

        return distances;
    }
};

void test_main() {
    BellmanFord<int, int> bf(999999);
    bf.add_edge(0, 1, 4);
    bf.add_edge(0, 2, 2);
    bf.add_edge(1, 2, -3);
    bf.add_edge(2, 3, 2);
    bf.add_edge(3, 1, 1);
```

```
    auto result = bf.shortest_paths(0);
    assert(result.has_value());
    assert(result.value()[2] == 1);
    assert(result.value()[3] == 3);
}
```

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

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");
}
```

Convex Hull

convex_hull.cpp

```cpp
/*
Andrew's monotone chain algorithm for computing the convex hull of 2D points.

Time complexity: O(n log n) dominated by sorting.
Space complexity: O(n).
*/

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

struct Point {
    double x, y;

    bool operator<(const Point& other) const {
        return x < other.x || (x == other.x && y < other.y);
    }

    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
};

double cross(const Point& o, const Point& a, const Point& b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

std::vector<Point> convex_hull(std::vector<Point> points) {
    if (points.size() <= 1) { return points; }

    std::sort(points.begin(), points.end());

    std::vector<Point> lower;
    for (const auto& p : points) {
        while (lower.size() >= 2 &&
               cross(lower[lower.size() - 2], lower[lower.size() - 1], p) <= 0) {
            lower.pop_back();
        }
        lower.push_back(p);
    }

    std::vector<Point> upper;
    for (int i = points.size() - 1; i >= 0; i--) {
        const auto& p = points[i];
        while (upper.size() >= 2 &&
               cross(upper[upper.size() - 2], upper[upper.size() - 1], p) <= 0) {
            upper.pop_back();
        }
        upper.push_back(p);
    }

    lower.pop_back();
    upper.pop_back();
    lower.insert(lower.end(), upper.begin(), upper.end());

    return lower;
}

void test_main() {
    std::vector<Point> pts = {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0.5, 0.5}};
    auto hull = convex_hull(pts);
    assert(hull.size() == 4);
    assert(std::find(hull.begin(), hull.end(), Point{0, 0}) != hull.end());
    assert(std::find(hull.begin(), hull.end(), Point{0.5, 0.5}) == hull.end());
}
```

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

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(const NodeT& source, const 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"}));
}
```

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

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);
}
```

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

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);
```

```cpp
        }
        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);
}
```

Kmp

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

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. */
```

```cpp
    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}));
}
```

Kosaraju Scc

kosaraju_scc.cpp

```cpp
/*
Kosaraju's algorithm for finding strongly connected components (SCCs) in directed graphs.

A strongly connected component is a maximal set of vertices where every vertex is
reachable from every other vertex in the set. Uses two DFS passes.

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

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

template <typename NodeT>
class KosarajuSCC {
  private:
    std::map<NodeT, std::vector<NodeT>> graph;
    std::map<NodeT, std::vector<NodeT>> transpose;

    void dfs1(NodeT node, std::set<NodeT>& visited, std::vector<NodeT>& finish_order) {
        visited.insert(node);
        if (graph.find(node) != graph.end()) {
            for (const auto& neighbor : graph[node]) {
                if (visited.find(neighbor) == visited.end()) {
                    dfs1(neighbor, visited, finish_order);
                }
            }
        }
        finish_order.push_back(node);
    }

    void dfs2(NodeT node, std::set<NodeT>& visited, std::vector<NodeT>& scc) {
        visited.insert(node);
        scc.push_back(node);
        if (transpose.find(node) != transpose.end()) {
            for (const auto& neighbor : transpose[node]) {
                if (visited.find(neighbor) == visited.end()) { dfs2(neighbor, visited, scc); }
            }
        }
    }

  public:
    void add_edge(NodeT u, NodeT v) {
        graph[u].push_back(v);
        transpose[v].push_back(u);
        graph.try_emplace(v, std::vector<NodeT>{});
        transpose.try_emplace(u, std::vector<NodeT>{});
    }

    std::vector<std::vector<NodeT>> find_sccs() {
        std::set<NodeT> visited;
        std::vector<NodeT> finish_order;

        for (const auto& [node, _] : graph) {
            if (visited.find(node) == visited.end()) { dfs1(node, visited, finish_order); }
        }

        visited.clear();
        std::vector<std::vector<NodeT>> sccs;

        for (auto it = finish_order.rbegin(); it != finish_order.rend(); ++it) {
            if (visited.find(*it) == visited.end()) {
                std::vector<NodeT> scc;
                dfs2(*it, visited, scc);
                sccs.push_back(scc);
```

```
            }
        }

        return sccs;
    }
};

void test_main() {
    KosarajuSCC<int> g;
    g.add_edge(0, 1);
    g.add_edge(1, 2);
    g.add_edge(2, 0);
    g.add_edge(1, 3);
    g.add_edge(3, 4);
    g.add_edge(4, 5);
    g.add_edge(5, 3);

    auto sccs = g.find_sccs();
    assert(sccs.size() == 2);

    std::vector<std::vector<int>> sorted;
    for (auto& scc : sccs) {
        std::sort(scc.begin(), scc.end());
        sorted.push_back(scc);
    }
    std::sort(sorted.begin(), sorted.end());

    assert(sorted[0] == std::vector<int>({0, 1, 2}));
    assert(sorted[1] == std::vector<int>({3, 4, 5}));
}
```

Lca

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

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, const 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, const 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(const 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)) {
```

```cpp
                    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);
}
```

Polygon Area

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

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

double polygon_area(const std::vector<PolygonPoint>& 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<PolygonPoint>& 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();
    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;
    }
```

```cpp
    return area / 2.0;
}

bool is_clockwise(const std::vector<PolygonPoint>& 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<PolygonPoint> 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<PolygonPoint> triangle = {{0.0, 0.0}, {3.0, 0.0}, {1.5, 4.0}};
    assert(polygon_area(triangle) == 6.0);

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

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

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
```

```cpp
        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.starts_with(keys[pos]) && 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].starts_with(s) && 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);
}
```

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.

Standard library alternatives:
- C++: std::priority_queue (basic operations only, no key-based updates/removal)
- Python: heapq module (min-heap only, no key-based updates/removal)
- Java: PriorityQueue class (basic operations only, no key-based updates/removal)

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

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

        Entry(PriorityT p, const 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--;
```

```cpp
    }

    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);
}
```

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

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, const 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, const 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);
}
```

Skiplist

skiplist.cpp

```cpp
/*
Skip list is a probabilistic data structure that maintains a sorted collection of elements.

It uses multiple levels of linked lists to achieve O(log n) average time complexity for
search, insertion, and deletion operations. Elements are inserted with randomly determined
heights, creating express lanes for faster traversal.

Standard library alternatives:
- C++: std::set / std::map (red-black tree, O(log n) guaranteed)
- Python: No built-in sorted set (use bisect module for sorted lists)
- Java: TreeSet / TreeMap (red-black tree, O(log n) guaranteed)

Time complexity: O(log n) average for search, insert, and delete operations.
Space complexity: O(n) on average, where n is the number of elements.
*/

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

template <typename T>
class SkipListNode {
  public:
    T value;
    std::vector<SkipListNode*> forward;

    SkipListNode(const T& val, int level) : value(val), forward(level + 1, nullptr) {}
};

template <typename T>
class SkipList {
  private:
    int max_level;
    float p;
    int level;
    SkipListNode<T>* header;

    int random_level() {
        int lvl = 0;
        while ((float)rand() / RAND_MAX < p && lvl < max_level) { lvl++; }
        return lvl;
    }

  public:
    SkipList(int max_lvl = 16, float prob = 0.5) : max_level(max_lvl), p(prob), level(0) {
        header = new SkipListNode<T>(T(), max_level);
    }

    ~SkipList() {
        SkipListNode<T>* current = header;
        while (current != nullptr) {
            SkipListNode<T>* next = current->forward[0];
            delete current;
            current = next;
        }
    }

    // Delete copy and move operations (not needed for competition)
    SkipList(const SkipList&) = delete;
    SkipList& operator=(const SkipList&) = delete;
    SkipList(SkipList&&) = delete;
    SkipList& operator=(SkipList&&) = delete;

    SkipList& insert(const T& value) {
        std::vector<SkipListNode<T>*> update(max_level + 1);
        SkipListNode<T>* current = header;
```

```
    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i]->value < value) {
            current = current->forward[i];
        }
        update[i] = current;
    }

    int lvl = random_level();
    if (lvl > level) {
        for (int i = level + 1; i <= lvl; i++) { update[i] = header; }
        level = lvl;
    }

    SkipListNode<T>* new_node = new SkipListNode<T>(value, lvl);
    for (int i = 0; i <= lvl; i++) {
        new_node->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = new_node;
    }

    return *this;
}

bool search(const T& value) {
    SkipListNode<T>* current = header;
    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i]->value < value) {
            current = current->forward[i];
        }
    }
    current = current->forward[0];
    return current != nullptr && current->value == value;
}

bool remove(const T& value) {
    std::vector<SkipListNode<T>*> update(max_level + 1);
    SkipListNode<T>* current = header;

    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i]->value < value) {
            current = current->forward[i];
        }
        update[i] = current;
    }

    current = current->forward[0];
    if (current == nullptr || current->value != value) { return false; }

    for (int i = 0; i <= level; i++) {
        if (update[i]->forward[i] != current) { break; }
        update[i]->forward[i] = current->forward[i];
    }

    delete current;

    while (level > 0 && header->forward[level] == nullptr) { level--; }

    return true;
}

// Optional functionality (not always needed during competition)

int size() const {
    int count = 0;
    SkipListNode<T>* current = header->forward[0];
    while (current != nullptr) {
        count++;
        current = current->forward[0];
    }
    return count;
}

std::vector<T> to_vector() const {
    std::vector<T> result;
    SkipListNode<T>* current = header->forward[0];
```

```cpp
        while (current != nullptr) {
            result.push_back(current->value);
            current = current->forward[0];
        }
        return result;
    }

    bool contains(const T& value) {
        return search(value);
    }
};

void test_main() {
    srand(42);
    SkipList<int> sl;
    sl.insert(10).insert(20).insert(5).insert(15);
    assert(sl.search(10));
    assert(sl.search(20));
    assert(!sl.search(25));
    assert(sl.remove(10));
    assert(!sl.search(10));
    assert(!sl.remove(30));

    // Optional functionality (not always needed during competition)
    srand(42);
    SkipList<int> sl2;
    sl2.insert(3).insert(1).insert(4).insert(1).insert(5);
    assert(sl2.size() == 5);
    std::vector<int> expected = {1, 1, 3, 4, 5};
    assert(sl2.to_vector() == expected);
    assert(sl2.contains(3));
    assert(!sl2.contains(7));
}
```

Sprague Grundy

sprague_grundy.cpp

```cpp
/*
Sprague-Grundy theorem implementation for impartial games (finite, acyclic, normal-play).

The Sprague-Grundy theorem states that every impartial game is equivalent to a Nim heap
of size equal to its Grundy number (nimber). For multiple independent games,
XOR the Grundy numbers to determine the combined game value.

API:
- GrundyEngine(moveFunction): makes it easy to plug in any game.
- grundy(state): compute nimber for a state (must be hashable).
- grundy_multi(states): XOR of nimbers for independent subgames.
- is_winning_position(states): true iff XOR != 0.

Includes implementations for:
- Nim (single heap).
- Subtraction game (allowed moves = {1,3,4}) with period detection.
- Kayles (bowling pins) with splits into subgames via vector representation.

Requirements:
- State must be hashable and canonically represented (e.g., sorted vectors).
- moveFunction must not create cycles.
*/

#include <algorithm>
#include <cassert>
#include <functional>
#include <iostream>
#include <map>
#include <optional>
#include <set>
#include <vector>

// Minimum EXcludant: smallest non-negative integer not occurring in 'values'
int mex(const std::vector<int>& values) {
    std::set<int> s(values.begin(), values.end());
    int g = 0;
    while (s.count(g)) { g++; }
    return g;
}

template <typename T>
class GrundyEngine {
  private:
    std::function<std::vector<T>(const T&)> moves;
    mutable std::map<T, int> cache;

  public:
    GrundyEngine(std::function<std::vector<T>(const T&)> move_function) : moves(move_function) {}

    int grundy(const T& state) const {
        if (cache.count(state)) { return cache[state]; }

        std::vector<T> next_states = moves(state);
        if (next_states.empty()) {
            cache[state] = 0;
            return 0;
        }

        std::vector<int> nimbers;
        for (const auto& next_state : next_states) { nimbers.push_back(grundy(next_state)); }

        int result = mex(nimbers);
        cache[state] = result;
        return result;
    }

    int grundy_multi(const std::vector<T>& states) const {
        int result = 0;
        for (const auto& state : states) { result ^= grundy(state); }
```

```
        return result;
    }

    bool is_winning_position(const std::vector<T>& states) const {
        return grundy_multi(states) != 0;
    }
};

// Optional functionality (not always needed during competition)

std::optional<int> detect_period(const std::vector<int>& seq, int min_period = 1,
                                 std::optional<int> max_period = std::nullopt) {
    int n = seq.size();
    if (!max_period) { max_period = n / 2; }

    for (int p = min_period; p <= *max_period; p++) {
        bool ok = true;
        for (int i = 0; i < n; i++) {
            if (seq[i] != seq[i % p]) {
                ok = false;
                break;
            }
        }
        if (ok) { return p; }
    }
    return std::nullopt;
}

std::vector<int> nim_moves_single_heap(int n) {
    std::vector<int> moves;
    for (int k = 0; k < n; k++) {
        moves.push_back(k);  // leave 0..n-1
    }
    return moves;
}

std::function<std::vector<int>(int)> subtraction_game_moves_factory(const std::set<int>& allowed) {
    std::vector<int> allowed_sorted(allowed.begin(), allowed.end());

    return [allowed_sorted](int n) -> std::vector<int> {
        std::vector<int> moves;
        for (int d : allowed_sorted) {
            if (d <= n) { moves.push_back(n - d); }
        }
        return moves;
    };
}

std::vector<std::vector<int>> kayles_moves(const std::vector<int>& segments) {
    std::set<std::vector<int>> result_set;

    for (size_t idx = 0; idx < segments.size(); idx++) {
        int n = segments[idx];
        if (n <= 0) continue;

        // Remove one pin at position i (0..n-1)
        for (int i = 0; i < n; i++) {
            int left = i;
            int right = n - i - 1;
            std::vector<int> new_seg;

            for (size_t j = 0; j < idx; j++) { new_seg.push_back(segments[j]); }
            if (left > 0) new_seg.push_back(left);
            if (right > 0) new_seg.push_back(right);
            for (size_t j = idx + 1; j < segments.size(); j++) { new_seg.push_back(segments[j]); }

            std::sort(new_seg.begin(), new_seg.end());
            result_set.insert(new_seg);
        }

        // Remove two adjacent pins at position i,i+1 (0..n-2)
        for (int i = 0; i < n - 1; i++) {
            int left = i;
            int right = n - i - 2;
            std::vector<int> new_seg;
```

```cpp
            for (size_t j = 0; j < idx; j++) { new_seg.push_back(segments[j]); }
            if (left > 0) new_seg.push_back(left);
            if (right > 0) new_seg.push_back(right);
            for (size_t j = idx + 1; j < segments.size(); j++) { new_seg.push_back(segments[j]); }

            std::sort(new_seg.begin(), new_seg.end());
            result_set.insert(new_seg);
        }
    }

    return std::vector<std::vector<int>>(result_set.begin(), result_set.end());
}

void test_main() {
    // Test Nim with larger values
    GrundyEngine<int> eng(nim_moves_single_heap);
    assert(eng.grundy(42) == 42);
    assert(eng.grundy_multi({17, 23, 31}) == 25);           // 17^23^31 = 25
    assert(eng.is_winning_position({15, 27, 36}) == true);  // 15^27^36 = 48 != 0

    // Test subtraction game {1,3,4} with period 7
    auto moves2 = subtraction_game_moves_factory({1, 3, 4});
    GrundyEngine<int> eng2(moves2);
    assert(eng2.grundy(14) == 0);  // 14 % 7 = 0 → grundy = 0
    assert(eng2.grundy(15) == 1);  // 15 % 7 = 1 → grundy = 1
    assert(eng2.grundy(18) == 2);  // 18 % 7 = 4 → grundy = 2

    // Test Kayles
    GrundyEngine<std::vector<int>> eng3(kayles_moves);
    assert(eng3.grundy({7}) == 2);     // K(7) = 2
    assert(eng3.grundy({3, 5}) == 7);  // K(3)^K(5) = 3^4 = 7
}
```

Suffix Array

suffix_array.cpp

```cpp
/*
Suffix Array construction with Longest Common Prefix (LCP) array using Kasai's algorithm.

Time complexity: O(n log n) for suffix array, O(n) for LCP array.
Space complexity: O(n).
*/

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

class SuffixArray {
  private:
    std::string text;
    int n;
    std::vector<int> sa;
    std::vector<int> lcp;

    std::vector<int> build_suffix_array() {
        std::vector<int> suffixes(n);
        for (int i = 0; i < n; i++) { suffixes[i] = i; }
        std::sort(suffixes.begin(), suffixes.end(),
                  [this](int a, int b) { return text.substr(a) < text.substr(b); });
        return suffixes;
    }

    std::vector<int> build_lcp_array() {
        if (n == 0) { return {}; }

        std::vector<int> rank(n);
        for (int i = 0; i < n; i++) { rank[sa[i]] = i; }

        std::vector<int> lcp(n, 0);
        int h = 0;

        for (int i = 0; i < n; i++) {
            if (rank[i] > 0) {
                int j = sa[rank[i] - 1];
                while (i + h < n && j + h < n && text[i + h] == text[j + h]) { h++; }
                lcp[rank[i]] = h;
                if (h > 0) { h--; }
            }
        }

        return lcp;
    }

  public:
    SuffixArray(const std::string& text) : text(text), n(text.length()) {
        sa = build_suffix_array();
        lcp = build_lcp_array();
    }

    std::vector<int> find_pattern(const std::string& pattern) {
        if (pattern.empty()) { return {}; }

        int m = pattern.length();
        int left = 0, right = n;

        while (left < right) {
            int mid = (left + right) / 2;
            if (text.substr(sa[mid]) < pattern) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
```

```
        int start = left;
        left = start;
        right = n;

        while (left < right) {
            int mid = (left + right) / 2;
            std::string suffix = text.substr(sa[mid], std::min(m, n - sa[mid]));
            if (suffix <= pattern) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        int end = left;

        std::vector<int> result;
        for (int i = start; i < end; i++) {
            if (sa[i] + m <= n && text.substr(sa[i], m) == pattern) { result.push_back(sa[i]); }
        }

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

    const std::vector<int>& get_sa() const {
        return sa;
    }

    const std::vector<int>& get_lcp() const {
        return lcp;
    }
};

void test_main() {
    SuffixArray sa("banana");
    assert(sa.get_sa() == std::vector<int>({5, 3, 1, 0, 4, 2}));
    assert(sa.get_lcp() == std::vector<int>({0, 1, 3, 0, 0, 2}));

    auto positions = sa.find_pattern("ana");
    assert(positions == std::vector<int>({1, 3}));
}
```

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

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) {
        graph.try_emplace(u, std::vector<NodeT>{});
        in_degree.try_emplace(u, 0);
        in_degree.try_emplace(v, 0);
        graph.try_emplace(v, std::vector<NodeT>{});

        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); }
        }
```

```cpp
        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());
}
```

Two Sat

two_sat.cpp

```cpp
/*
2-SAT solver using Kosaraju's SCC algorithm on implication graph.

2-SAT determines if a Boolean formula in CNF with at most 2 literals per clause is satisfiable.

Time complexity: O(n + m) where n is variables and m is clauses.
Space complexity: O(n + m) for the implication graph.
*/

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

class TwoSAT {
  private:
    int n;
    std::vector<std::vector<int>> graph;
    std::vector<std::vector<int>> transpose;

    void dfs1(int node, std::vector<bool>& visited, std::vector<int>& finish_order) {
        visited[node] = true;
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) { dfs1(neighbor, visited, finish_order); }
        }
        finish_order.push_back(node);
    }

    void dfs2(int node, std::vector<bool>& visited, std::vector<int>& scc_id, int scc) {
        visited[node] = true;
        scc_id[node] = scc;
        for (int neighbor : transpose[node]) {
            if (!visited[neighbor]) { dfs2(neighbor, visited, scc_id, scc); }
        }
    }

  public:
    TwoSAT(int n) : n(n), graph(2 * n), transpose(2 * n) {}

    void add_clause(int a, bool a_neg, int b, bool b_neg) {
        int a_node = 2 * a + (a_neg ? 1 : 0);
        int b_node = 2 * b + (b_neg ? 1 : 0);
        int na_node = 2 * a + (a_neg ? 0 : 1);
        int nb_node = 2 * b + (b_neg ? 0 : 1);

        graph[na_node].push_back(b_node);
        graph[nb_node].push_back(a_node);
        transpose[b_node].push_back(na_node);
        transpose[a_node].push_back(nb_node);
    }

    std::vector<bool> solve() {
        std::vector<bool> visited(2 * n, false);
        std::vector<int> finish_order;

        for (int node = 0; node < 2 * n; node++) {
            if (!visited[node]) { dfs1(node, visited, finish_order); }
        }

        std::fill(visited.begin(), visited.end(), false);
        std::vector<int> scc_id(2 * n);
        int current_scc = 0;

        for (int i = finish_order.size() - 1; i >= 0; i--) {
            int node = finish_order[i];
            if (!visited[node]) {
                dfs2(node, visited, scc_id, current_scc);
                current_scc++;
            }
```

```cpp
        }

        for (int i = 0; i < n; i++) {
            if (scc_id[2 * i] == scc_id[2 * i + 1]) { return {}; }
        }

        std::vector<bool> assignment(n);
        for (int i = 0; i < n; i++) { assignment[i] = scc_id[2 * i] > scc_id[2 * i + 1]; }

        return assignment;
    }
};

void test_main() {
    TwoSAT sat(2);
    sat.add_clause(0, false, 1, false);
    sat.add_clause(0, true, 1, false);
    sat.add_clause(0, false, 1, true);

    auto result = sat.solve();
    assert(!result.empty());
    assert(result[0] || result[1]);
    assert(!result[0] || result[1]);
    assert(result[0] || !result[1]);
}
```

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;
}
```