

Algorithm Reference - Python

Bellman Ford

Bipartite Match

Convex Hull

Dijkstra

Edmonds Karp

Fenwick Tree

Kmp

Kosaraju Scc

Lca

Polygon Area

Prefix Tree

Priority Queue

Segment Tree

Skiplist

Sprague Grundy

Suffix Array

Topological Sort

Two Sat

Union Find

Bellman Ford

bellman_ford.py

"""

Bellman-Ford algorithm for single-source shortest paths with negative edge weights.

Finds shortest paths from a source vertex to all other vertices, even with negative edge weights. Can detect negative cycles reachable from the source. Uses edge relaxation V-1 times, then checks for negative cycles.

Time complexity: $O(VE)$ where V is vertices and E is edges.

Space complexity: $O(V + E)$ for graph representation and distance array.

"""

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar
```

```
from typing_extensions import Self
```

```
class Comparable(Protocol):
```

```
    def __lt__(self, other: Self, /) -> bool: ...
```

```
    def __add__(self, other: Self, /) -> Self: ...
```

```
WeightT = TypeVar("WeightT", bound=Comparable)
```

```
NodeT = TypeVar("NodeT")
```

```
class BellmanFord(Generic[NodeT, WeightT]):
```

```
    def __init__(self, infinity: WeightT, zero: WeightT) -> None:
```

```
        self.infinity: Final[WeightT] = infinity
```

```
        self.zero: Final[WeightT] = zero
```

```
        self.edges: list[tuple[NodeT, NodeT, WeightT]] = []
```

```
        self.nodes: set[NodeT] = set()
```

```
    def add_edge(self, u: NodeT, v: NodeT, weight: WeightT) -> None:
```

```
        """Add directed edge from u to v with given weight."""
```

```
        self.edges.append((u, v, weight))
```

```
        self.nodes.add(u)
```

```
        self.nodes.add(v)
```

```
    def shortest_paths(
```

```
        self, source: NodeT
```

```
) -> tuple[dict[NodeT, WeightT], dict[NodeT, NodeT | None]] | None:
```

```
    """
```

```
        Find shortest paths from source to all reachable vertices.
```

```
        Returns (distances, predecessors) if no negative cycle reachable from source,  
        None if negative cycle detected.
```

```
        - distances[v] = shortest distance from source to v
```

```
        - predecessors[v] = previous vertex in shortest path to v (None for source)
```

```
    """
```

```
        distances: dict[NodeT, WeightT] = {}
```

```
        predecessors: dict[NodeT, NodeT | None] = {}
```

```
        # Initialize distances
```

```
        for node in self.nodes:
```

```
            distances[node] = self.infinity
```

```
        distances[source] = self.zero
```

```
        predecessors[source] = None
```

```
        # Relax edges V-1 times
```

```
        for _ in range(len(self.nodes) - 1):
```

```
            for u, v, weight in self.edges:
```

```
                if distances[u] != self.infinity and distances[u] + weight < distances[v]:
```

```
                    distances[v] = distances[u] + weight
```

```
                    predecessors[v] = u
```

```
        # Check for negative cycles
```

```

    for u, v, weight in self.edges:
        if distances[u] != self.infinity and distances[u] + weight < distances[v]:
            return None # Negative cycle detected

    return distances, predecessors

def shortest_path(self, source: NodeT, target: NodeT) -> list[NodeT] | None:
    """
    Get the shortest path from source to target.

    Returns path as list of nodes, or None if unreachable or negative cycle detected.
    """
    result = self.shortest_paths(source)
    if result is None:
        return None # Negative cycle

    _, predecessors = result
    if target not in predecessors:
        return None # Unreachable

    path = []
    current: NodeT | None = target
    while current is not None:
        path.append(current)
        current = predecessors.get(current)

    return path[::-1]

```

```

def test_main() -> None:
    bf: BellmanFord[str, float] = BellmanFord(float("inf"), 0.0)
    bf.add_edge("A", "B", 4.0)
    bf.add_edge("A", "C", 2.0)
    bf.add_edge("B", "C", -3.0) # Negative edge makes A->B->C better than A->C
    bf.add_edge("C", "D", 2.0)
    bf.add_edge("D", "B", 1.0)

    result = bf.shortest_paths("A")
    assert result is not None
    distances, _ = result
    # A->B: 4, A->C: min(2, 4+(-3)) = 1, A->D: 1+2 = 3
    assert distances["C"] == 1.0
    assert distances["D"] == 3.0

    path = bf.shortest_path("A", "D")
    assert path is not None
    assert path[0] == "A"
    assert path[-1] == "D"

```

Don't write tests below during competition.

```

def test_negative_cycle() -> None:
    bf: BellmanFord[int, int] = BellmanFord(999999, 0)
    bf.add_edge(0, 1, 1)
    bf.add_edge(1, 2, -3)
    bf.add_edge(2, 0, 1) # Cycle: 0->1->2->0 with total weight 1 + (-3) + 1 = -1

    result = bf.shortest_paths(0)
    assert result is None # Should detect negative cycle

```

```

def test_single_node() -> None:
    bf: BellmanFord[str, float] = BellmanFord(float("inf"), 0.0)
    bf.nodes.add("A")

    result = bf.shortest_paths("A")
    assert result is not None
    distances, predecessors = result
    assert distances["A"] == 0.0
    assert predecessors["A"] is None

```

```

def test_unreachable_nodes() -> None:

```

```
bf: BellmanFord[int, int] = BellmanFord(999999, 0)
bf.add_edge(1, 2, 5)
bf.add_edge(3, 4, 3)
```

```
result = bf.shortest_paths(1)
assert result is not None
distances, _ = result
assert distances[2] == 5
assert distances[3] == 999999 # Unreachable
assert distances[4] == 999999 # Unreachable
```

```
def test_all_negative_edges() -> None:
```

```
bf: BellmanFord[str, int] = BellmanFord(999999, 0)
bf.add_edge("A", "B", -1)
bf.add_edge("B", "C", -2)
bf.add_edge("C", "D", -3)
```

```
result = bf.shortest_paths("A")
assert result is not None
distances, _ = result
assert distances["D"] == -6 # -1 + (-2) + (-3)
```

```
def test_path_reconstruction() -> None:
```

```
bf: BellmanFord[int, int] = BellmanFord(999999, 0)
bf.add_edge(0, 1, 5)
bf.add_edge(1, 2, 3)
bf.add_edge(0, 2, 10)
```

```
path = bf.shortest_path(0, 2)
assert path is not None
assert path == [0, 1, 2] # Should take 0->1->2 (cost 8) not 0->2 (cost 10)
```

```
def test_negative_edge_relaxation() -> None:
```

```
# Test that negative edges properly relax distances
```

```
bf: BellmanFord[int, int] = BellmanFord(999999, 0)
bf.add_edge(0, 1, 10)
bf.add_edge(0, 2, 5)
bf.add_edge(2, 1, -8) # This should make path 0->2->1 better than 0->1
```

```
result = bf.shortest_paths(0)
assert result is not None
distances, _ = result
assert distances[1] == -3 # 5 + (-8) = -3, better than direct path of 10
```

```
def test_disconnected_graph() -> None:
```

```
bf: BellmanFord[int, int] = BellmanFord(999999, 0)
bf.add_edge(0, 1, 1)
bf.add_edge(2, 3, 1)
```

```
result = bf.shortest_paths(0)
assert result is not None
distances, _ = result
assert distances[1] == 1
assert distances[2] == 999999
assert distances[3] == 999999
```

```
def test_self_loop_negative() -> None:
```

```
bf: BellmanFord[int, int] = BellmanFord(999999, 0)
bf.add_edge(0, 0, -1) # Negative self-loop
```

```
result = bf.shortest_paths(0)
assert result is None # Should detect negative cycle
```

```
def test_complex_graph() -> None:
```

```
bf: BellmanFord[str, int] = BellmanFord(999999, 0)
bf.add_edge("S", "A", 10)
bf.add_edge("S", "E", 8)
bf.add_edge("A", "C", 2)
bf.add_edge("C", "D", 5) # Changed to avoid negative cycle
```

```
bf.add_edge("D", "B", 3)
bf.add_edge("E", "D", 1)

result = bf.shortest_paths("S")
assert result is not None
distances, _ = result
# S->E->D: 8 + 1 = 9
# S->E->D->B: 9 + 3 = 12
assert distances["D"] == 9
assert distances["B"] == 12
```

```
def main() -> None:
    test_main()
    test_negative_cycle()
    test_single_node()
    test_unreachable_nodes()
    test_all_negative_edges()
    test_path_reconstruction()
    test_negative_edge_relaxation()
    test_disconnected_graph()
    test_self_loop_negative()
    test_complex_graph()
```

```
if __name__ == "__main__":
    main()
```

Bipartite Match

bipartite_match.py

"""

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: $O(V \cdot E)$, where V is the number of vertices and E the number of edges.

"""

```
from __future__ import annotations
```

```
from collections import defaultdict
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar
```

```
from typing_extensions import Self
```

```
class Comparable(Protocol):
```

```
    def __lt__(self, other: Self, /) -> bool: ...
```

```
SourceT = TypeVar("SourceT", bound=Comparable)
```

```
SinkT = TypeVar("SinkT")
```

```
class BipartiteMatch(Generic[SourceT, SinkT]):
```

```
    def __init__(self, edges: list[tuple[SourceT, SinkT]]) -> None:
```

```
        self.edges: defaultdict[SourceT, list[SinkT]] = defaultdict(list)
```

```
        for source, sink in edges:
```

```
            self.edges[source].append(sink)
```

```
        # For deterministic behaviour
```

```
        ordered_sources = sorted(self.edges)
```

```
        used_sources: dict[SourceT, SinkT] = {}
```

```
        used_sinks: dict[SinkT, SourceT] = {}
```

```
        # Initial pass
```

```
        for source, sink in edges:
```

```
            if used_sources.get(source) is None and used_sinks.get(sink) is None:
```

```
                progress = True
```

```
                used_sources[source] = sink
```

```
                used_sinks[sink] = source
```

```
                break
```

```
        coloring = dict.fromkeys(ordered_sources, 0)
```

```
    def update(source: SourceT, cur_color: int) -> bool:
```

```
        sink = used_sources.get(source)
```

```
        if sink is not None:
```

```
            return False
```

```
        source_stack: list[SourceT] = [source]
```

```
        sink_stack: list[SinkT] = []
```

```
        index_stack: list[int] = [0]
```

```
    def flip() -> None:
```

```
        while source_stack:
```

```
            used_sources[source_stack[-1]] = sink_stack[-1]
```

```
            used_sinks[sink_stack[-1]] = source_stack[-1]
```

```
            source_stack.pop()
```

```
            sink_stack.pop()
```

```
    while True:
```

```
        source = source_stack[-1]
```

```
        index = index_stack.pop()
```

```
        if index == len(self.edges[source]):
```

```

        if not index_stack:
            return False
        source_stack.pop()
        sink_stack.pop()
        continue
    index_stack.append(index + 1)

    sink = self.edges[source][index]
    sink_stack.append(sink)
    if sink not in used_sinks:
        flip()
        return True
    source = used_sinks[sink]
    if coloring[source] == cur_color:
        sink_stack.pop()
    else:
        coloring[source] = cur_color
        source_stack.append(source)
        index_stack.append(0)

    progress = True
    cur_color = 1
    while progress:
        progress = any(update(source, cur_color) for source in ordered_sources)
        cur_color += 1

    self.match: Final = used_sources

```

```

def test_main() -> None:
    b = BipartiteMatch([(1, "X"), (2, "Y"), (3, "X"), (1, "Z"), (2, "Z"), (3, "Y")])
    assert len(b.match) == 3
    assert b.match == {1: "Z", 2: "Y", 3: "X"}

```

Don't write tests below during competition.

```

def test_a() -> None:
    bm: BipartiteMatch[int, float] = BipartiteMatch(
        [
            (1, 2.2),
            (2, 3.3),
            (1, 1.1),
            (2, 2.2),
            (3, 3.3),
        ]
    )
    assert bm.match == {1: 1.1, 2: 2.2, 3: 3.3}

```

```

def test_b() -> None:
    bm: BipartiteMatch[str, str] = BipartiteMatch(
        [
            ("1", "3"),
            ("2", "4"),
            ("3", "2"),
            ("4", "4"),
            ("1", "1"),
        ]
    )
    assert bm.match == {"3": "2", "1": "3", "2": "4"}

```

```

def test_c() -> None:
    bm: BipartiteMatch[int, str] = BipartiteMatch(
        [
            (1, "B"),
            (2, "A"),
            (3, "A"),
        ]
    )
    assert bm.match == {1: "B", 2: "A"}

```

```

def test_empty_graph() -> None:
    bm: BipartiteMatch[int, int] = BipartiteMatch([])
    assert bm.match == {}

def test_single_edge() -> None:
    bm: BipartiteMatch[int, int] = BipartiteMatch([(1, 2)])
    assert bm.match == {1: 2}

def test_no_matching() -> None:
    # All sources want same sink
    bm: BipartiteMatch[int, str] = BipartiteMatch(
        [(1, "A"), (2, "A"), (3, "A")]
    )
    # Only one can be matched
    assert len(bm.match) == 1
    assert bm.match[next(iter(bm.match.keys()))] == "A"

def test_perfect_matching() -> None:
    # Perfect matching possible
    bm: BipartiteMatch[int, int] = BipartiteMatch(
        [(1, 10), (2, 20), (3, 30)]
    )
    assert len(bm.match) == 3

def test_augmenting_path() -> None:
    # Requires augmenting path to find maximum matching
    bm: BipartiteMatch[int, str] = BipartiteMatch(
        [
            (1, "A"), (1, "B"),
            (2, "B"), (2, "C"),
            (3, "C"),
        ]
    )
    assert len(bm.match) == 3

def test_large_bipartite() -> None:
    # Larger graph
    edges: list[tuple[int, int]] = []
    for i in range(10):
        edges.extend((i, j + 100) for j in range(i, min(i + 3, 10)))

    bm: BipartiteMatch[int, int] = BipartiteMatch(edges)
    # Should find a good matching
    assert len(bm.match) >= 8

def main() -> None:
    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()

if __name__ == "__main__":
    main()

```


Convex Hull

convex_hull.py

"""

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

Computes the convex hull (smallest convex polygon containing all points) using a simple and robust algorithm. Works in sorted order to build lower and upper hulls.

Time complexity: $O(n \log n)$ dominated by sorting, where n is number of points.

Space complexity: $O(n)$ for the hull and auxiliary structures.

"""

from __future__ import annotations

Don't use annotations during contest

from dataclasses import dataclass

@dataclass(frozen=True, order=True)

class Point:

 x: float

 y: float

def cross(o: Point, a: Point, b: Point) -> float:

 """

 Cross product of vectors OA and OB.

 Positive if OAB makes a counter-clockwise turn, negative if clockwise, zero if collinear.

 """

 return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x)

def convex_hull(points: list[Point]) -> list[Point]:

 """

 Compute convex hull using Andrew's monotone chain algorithm.

 Returns points on the convex hull in counter-clockwise order starting from the leftmost point. Includes collinear points only if they are extreme points.

 """

 if len(points) <= 1:

 return points[:]

 # Sort points lexicographically (first by x, then by y)

 sorted_points = sorted(points)

 # Build lower hull

 lower: list[Point] = []

 for p in sorted_points:

 while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:

 lower.pop()

 lower.append(p)

 # Build upper hull

 upper: list[Point] = []

 for p in reversed(sorted_points):

 while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:

 upper.pop()

 upper.append(p)

 # Remove last point of each half because it's repeated

 return lower[:-1] + upper[:-1]

def test_main() -> None:

 # Test square

 pts = [Point(0, 0), Point(1, 0), Point(1, 1), Point(0, 1), Point(0.5, 0.5)]

 hull = convex_hull(pts)

 assert len(hull) == 4

 assert Point(0, 0) in hull

 assert Point(1, 0) in hull

 assert Point(1, 1) in hull

```
assert Point(0, 1) in hull
assert Point(0.5, 0.5) not in hull
```

Don't write tests below during competition.

```
def test_empty() -> None:
    assert convex_hull([]) == []
```

```
def test_single_point() -> None:
    pts = [Point(1, 2)]
    hull = convex_hull(pts)
    assert hull == pts
```

```
def test_two_points() -> None:
    pts = [Point(0, 0), Point(1, 1)]
    hull = convex_hull(pts)
    assert len(hull) == 2
    assert Point(0, 0) in hull
    assert Point(1, 1) in hull
```

```
def test_collinear_points() -> None:
    pts = [Point(0, 0), Point(1, 1), Point(2, 2), Point(3, 3)]
    hull = convex_hull(pts)
    # Only endpoints should be in hull
    assert len(hull) == 2
    assert Point(0, 0) in hull
    assert Point(3, 3) in hull
```

```
def test_triangle() -> None:
    pts = [Point(0, 0), Point(2, 0), Point(1, 2)]
    hull = convex_hull(pts)
    assert len(hull) == 3
    for p in pts:
        assert p in hull
```

```
def test_with_interior_points() -> None:
    # Pentagon with interior points
    pts = [
        Point(0, 0),
        Point(4, 0),
        Point(4, 3),
        Point(2, 4),
        Point(0, 3),
        Point(2, 2), # Interior
        Point(2, 1), # Interior
    ]
    hull = convex_hull(pts)
    assert len(hull) == 5
    assert Point(2, 2) not in hull
    assert Point(2, 1) not in hull
```

```
def test_all_same_point() -> None:
    pts = [Point(1, 1), Point(1, 1), Point(1, 1)]
    hull = convex_hull(pts)
    # Algorithm returns one point per position (duplicates removed by sorting)
    assert all(p == Point(1, 1) for p in hull)
```

```
def test_negative_coordinates() -> None:
    pts = [Point(-1, -1), Point(1, -1), Point(1, 1), Point(-1, 1)]
    hull = convex_hull(pts)
    assert len(hull) == 4
```

```
def test_large_square_with_interior() -> None:
    # 10x10 grid, hull should be the 4 corners
    pts = [Point(float(i), float(j)) for i in range(11) for j in range(11)]
```

```
hull = convex_hull(pts)
assert len(hull) == 4
assert Point(0, 0) in hull
assert Point(10, 0) in hull
assert Point(10, 10) in hull
assert Point(0, 10) in hull
```

```
def test_circle_approximation() -> None:
    import math
```

```
    # Points on a circle
```

```
    pts = [Point(math.cos(i * 0.5), math.sin(i * 0.5)) for i in range(13)]
```

```
    hull = convex_hull(pts)
```

```
    # All points should be on the hull (circle is convex)
```

```
    assert len(hull) == len(pts)
```

```
def main() -> None:
```

```
    test_main()
```

```
    test_empty()
```

```
    test_single_point()
```

```
    test_two_points()
```

```
    test_collinear_points()
```

```
    test_triangle()
```

```
    test_with_interior_points()
```

```
    test_all_same_point()
```

```
    test_negative_coordinates()
```

```
    test_large_square_with_interior()
```

```
    test_circle_approximation()
```

```
if __name__ == "__main__":
    main()
```

Dijkstra

dijkstra.py

```
"""
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.
"""
```

```
from __future__ import annotations
```

```
import heapq
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar
```

```
from typing_extensions import Self
```

```
class Comparable(Protocol):
    def __lt__(self, other: Self, /) -> bool: ...
    def __add__(self, other: Self, /) -> Self: ...
```

```
WeightT = TypeVar("WeightT", bound=Comparable)
NodeT = TypeVar("NodeT")
```

```
class Dijkstra(Generic[NodeT, WeightT]):
    def __init__(self, infinity: WeightT, zero: WeightT) -> None:
        self.infinity: Final[WeightT] = infinity
        self.zero: Final[WeightT] = zero
        self.graph: dict[NodeT, list[tuple[NodeT, WeightT]]] = {}

    def add_edge(self, u: NodeT, v: NodeT, weight: WeightT) -> None:
        """Add directed edge from u to v with given weight."""
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append((v, weight))

    def shortest_paths(
        self, source: NodeT
    ) -> tuple[dict[NodeT, WeightT], dict[NodeT, NodeT | None]]:
        """
        Find shortest paths from source to all reachable vertices.

        Returns (distances, predecessors) where:
        - distances[v] = shortest distance from source to v
        - predecessors[v] = previous vertex in shortest path to v (None for source)
        """
        distances: dict[NodeT, WeightT] = {source: self.zero}
        predecessors: dict[NodeT, NodeT | None] = {source: None}
        pq: list[tuple[WeightT, NodeT]] = [(self.zero, source)]
        visited: set[NodeT] = set()

        while pq:
            current_dist, u = heapq.heappop(pq)

            if u in visited:
                continue
            visited.add(u)

            if u not in self.graph:
                continue

            for v, weight in self.graph[u]:
                new_dist = current_dist + weight
```

```

        if v not in distances or new_dist < distances[v]:
            distances[v] = new_dist
            predecessors[v] = u
            heapq.heappush(pq, (new_dist, v))

```

```

    return distances, predecessors

```

```

def shortest_path(self, source: NodeT, target: NodeT) -> list[NodeT] | None:
    """Get the shortest path from source to target, or None if unreachable."""
    _, predecessors = self.shortest_paths(source)

```

```

    if target not in predecessors:
        return None

```

```

    path = []
    current: NodeT | None = target
    while current is not None:
        path.append(current)
        current = predecessors.get(current)

```

```

    return path[::-1]

```

```

def test_main() -> None:
    d: Dijkstra[str, float] = Dijkstra(float("inf"), 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)

```

```

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

```

```

    path = d.shortest_path("A", "D")
    assert path == ["A", "B", "D"]

```

Don't write tests below during competition.

```

def test_single_node() -> None:
    d: Dijkstra[str, float] = Dijkstra(float("inf"), 0.0)

    distances, predecessors = d.shortest_paths("A")
    assert distances == {"A": 0.0}
    assert predecessors == {"A": None}

    path = d.shortest_path("A", "A")
    assert path == ["A"]

```

```

def test_unreachable_nodes() -> None:
    d: Dijkstra[int, int] = Dijkstra(999999, 0)
    d.add_edge(1, 2, 5)
    d.add_edge(3, 4, 3)

    distances, _ = d.shortest_paths(1)
    assert distances[2] == 5
    assert 3 not in distances
    assert 4 not in distances

    path = d.shortest_path(1, 4)
    assert path is None

```

```

def test_negative_zero_weights() -> None:
    d: Dijkstra[str, float] = Dijkstra(float("inf"), 0.0)
    d.add_edge("A", "B", 0.0)
    d.add_edge("B", "C", 0.0)
    d.add_edge("A", "C", 5.0)

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

```

```

def test_dense_graph() -> None:
    # Complete graph with 5 nodes
    d: Dijkstra[int, int] = Dijkstra(999999, 0)

    # Add edges between all pairs
    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 (u, v), weight in weights.items():
        d.add_edge(u, v, weight)

    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

def test_large_graph() -> None:
    # Linear chain: 0->1->2->...->99
    d: Dijkstra[int, int] = Dijkstra(9999999, 0)

    for i in range(99):
        d.add_edge(i, i + 1, 1)

    distances, _ = d.shortest_paths(0)

    # Distance to node i should be i
    for i in range(100):
        assert distances[i] == i

    # Test path reconstruction
    path = d.shortest_path(0, 50)
    assert path == list(range(51))

def test_multiple_equal_paths() -> None:
    # Diamond-shaped graph with equal path lengths
    d: Dijkstra[str, int] = Dijkstra(9999999, 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)

    distances, _ = d.shortest_paths("S")
    assert distances["T"] == 5 # Both paths S->A->T and S->B->T have length 5

    path = d.shortest_path("S", "T")
    assert path is not None
    assert len(path) == 3
    assert path[0] == "S"
    assert path[-1] == "T"

def test_self_loops() -> None:
    d: Dijkstra[int, int] = Dijkstra(9999999, 0)
    d.add_edge(1, 1, 5) # Self-loop
    d.add_edge(1, 2, 3)

    distances, _ = d.shortest_paths(1)
    assert distances[1] == 0 # Distance to self is always 0
    assert distances[2] == 3

def test_decimal_weights() -> None:
    from decimal import Decimal

```

```
d: Dijkstra[str, Decimal] = Dijkstra(Decimal(999999), Decimal(0))
d.add_edge("A", "B", Decimal("1.5"))
d.add_edge("B", "C", Decimal("2.7"))
d.add_edge("A", "C", Decimal("5.0"))
```

```
distances, _ = d.shortest_paths("A")
assert distances["C"] == Decimal("4.2") # 1.5 + 2.7
```

```
def test_stress_many_nodes() -> None:
    # Star graph: center connected to many nodes
    d: Dijkstra[int, int] = Dijkstra(999999, 0)

    center = 0
    for i in range(1, 501): # 500 nodes connected to center
        d.add_edge(center, i, i)

    distances, _ = d.shortest_paths(center)

    # Distance to node i should be i
    for i in range(1, 501):
        assert distances[i] == i

    # Path from center to any node should be direct
    path = d.shortest_path(center, 100)
    assert path == [0, 100]
```

```
def main() -> None:
    test_main()
    test_single_node()
    test_unreachable_nodes()
    test_negative_zero_weights()
    test_dense_graph()
    test_large_graph()
    test_multiple_equal_paths()
    test_self_loops()
    test_decimal_weights()
    test_stress_many_nodes()
```

```
if __name__ == "__main__":
    main()
```

Edmonds Karp

edmonds_karp.py

```
"""
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:  $O(V \cdot E^2)$ , where  $V$  is the number of vertices and  $E$  the number of edges.
"""
```

```
from __future__ import annotations
```

```
from collections import deque
from decimal import Decimal
```

```
# Don't use annotations during contest
from typing import Final, Generic, TypeVar
```

```
DEBUG: Final = True
```

```
CapacityT = TypeVar("CapacityT", int, float, Decimal)
NodeT = TypeVar("NodeT")
```

```
class Edge(Generic[NodeT, CapacityT]):
    def __init__(
        self,
        source: Node[NodeT, CapacityT],
        sink: Node[NodeT, CapacityT],
        capacity: CapacityT,
    ) -> None:
        self.source: Final[Node[NodeT, CapacityT]] = source
        self.sink: Final[Node[NodeT, CapacityT]] = sink
        self.original = True # Part of the input graph or added for the algorithm?
        # Modified by EdmondsKarp.run
        self.initial_capacity: CapacityT = capacity
        self.capacity: CapacityT = capacity

    @property
    def rev(self) -> Edge[NodeT, CapacityT]:
        return self.sink.edges[self.source.node]

    @property
    def flow(self) -> CapacityT:
        return self.initial_capacity - self.capacity

    def __str__(self) -> str:
        return f"Edge({self.source.node}, {self.sink.node}, {self.capacity})"
```

```
class Node(Generic[NodeT, CapacityT]):
    def __init__(self, node: NodeT) -> None:
        self.node: Final = node
        self.edges: dict[NodeT, Edge[NodeT, CapacityT]] = {}
        # Modified by EdmondsKarp.run
        self.color = 0
        self.used_edge: Edge[NodeT, CapacityT] | None = None

    def __str__(self) -> str:
        return "Node({}, out={}, color={}, used_edge={})".format(
            self.node,
            ", ".join(str(edge) for edge in self.edges.values()),
            self.color,
            self.used_edge,
        )
```



```

class EdmondsKarp(Generic[NodeT, CapacityT]):
    def __init__(
        self,
        edges: list[tuple[NodeT, NodeT, CapacityT]],
        main_source: NodeT,
        main_sink: NodeT,
        zero: CapacityT,
    ) -> None:
        self.main_source: Final = main_source
        self.main_sink: Final = main_sink
        self.zero: Final[CapacityT] = zero
        self.color = 1
        self.total_flow: CapacityT = self.zero

    def init_nodes() -> dict[NodeT, Node[NodeT, CapacityT]]:
        nodes: dict[NodeT, Node[NodeT, CapacityT]] = {}
        for source_t, sink_t, capacity in edges:
            source = nodes.setdefault(source_t, Node(source_t))
            assert sink_t not in source.edges, (
                f"The edge ({source_t}, {sink_t}) is specified more than once"
            )
            source.edges[sink_t] = Edge(
                source, nodes.setdefault(sink_t, Node(sink_t)), capacity
            )
        for source_t, sink_t, _ in edges:
            sink = nodes[sink_t]
            if source_t not in sink.edges:
                edge = Edge(sink, nodes[source_t], zero)
                edge.original = False
                sink.edges[source_t] = edge
            nodes.setdefault(main_source, Node(main_source))
            nodes.setdefault(main_sink, Node(main_sink))
        return nodes

    self.nodes: dict[NodeT, Node[NodeT, CapacityT]] = init_nodes()

    def change_initial_capacities(self, edges: list[tuple[NodeT, NodeT, CapacityT]]) -> None:
        """Update edge capacities. REQUIRES: new capacity >= current flow."""
        for source, sink, capacity in edges:
            edge = self.nodes[source].edges[sink]
            assert capacity >= edge.flow
            increase = capacity - edge.initial_capacity
            edge.initial_capacity += increase
            edge.capacity += increase

    def reset_flows(self) -> None:
        """Reset all flows to zero, keeping capacities."""
        self.total_flow = self.zero
        for node in self.nodes.values():
            for edge in node.edges.values():
                edge.capacity = edge.initial_capacity

    def run(self) -> None:
        """Run max-flow algorithm from source to sink."""
        self.color += 1
        progress = True
        while progress:
            progress = False

            border: deque[Node[NodeT, CapacityT]] = deque()
            self.nodes[self.main_source].color = self.color
            border.append(self.nodes[self.main_source])
            while border:
                source = border.popleft()
                for edge in source.edges.values():
                    sink = edge.sink
                    if sink.color == self.color or edge.capacity == self.zero:
                        continue

                    sink.used_edge = edge
                    sink.color = self.color
                    border.append(sink)

                if sink.node == self.main_sink:
                    used_edge = edge

```

```

        flow = used_edge.capacity
        while used_edge.source.node != self.main_source:
            assert used_edge.source.used_edge is not None
            used_edge = used_edge.source.used_edge
            flow = min(flow, used_edge.capacity)

        self.total_flow += flow

        used_edge = sink.used_edge
        used_edge.capacity -= flow
        used_edge.rev.capacity += flow
        while used_edge.source.node != self.main_source:
            assert used_edge.source.used_edge is not None
            used_edge = used_edge.source.used_edge
            used_edge.capacity -= flow
            used_edge.rev.capacity += flow

        progress = True
        self.color += 1
        border.clear()
        break

```

```

def print(self) -> None:
    """Print all edges with non-zero flow for debugging."""
    if DEBUG:
        for node in self.nodes.values():
            for edge in node.edges.values():
                if edge.capacity < edge.initial_capacity:
                    print(
                        f"Flow {edge.source.node} ---{edge.flow}/{edge.initial_capacity}---> "
                        f"{edge.sink.node}"
                    )

```

```

def test_main() -> None:
    e = EdmondsKarp([(0, 1, 10), (0, 2, 8), (1, 2, 2), (1, 3, 5), (2, 3, 7)], 0, 3, 0)
    e.run()
    assert e.total_flow == 12

```

Don't write tests below during competition.

```

def test_a() -> None:
    bm = EdmondsKarp(
        [
            (0, 1, 1),
            (0, 2, 1),
            (0, 3, 1),
            (1, 12, 1),
            (2, 13, 1),
            (1, 11, 1),
            (2, 12, 1),
            (3, 13, 1),
            (11, 42, 1),
            (12, 42, 1),
            (13, 42, 1),
        ],
        main_source=0,
        main_sink=42,
        zero=0,
    )
    bm.run()
    bm.print()
    assert bm.total_flow == 3, bm.total_flow

```

```

def test_b() -> None:
    bm = EdmondsKarp(
        [
            # The +1 is to truncate to current decimal precision
            (0, 1, Decimal(1 / 3) + 0),
            (1, 2, Decimal(1 / 7) + 0),
            (2, 0, Decimal(1 / 9) + 0),
        ],
    )

```

```

        main_source=1,
        main_sink=0,
        zero=Decimal(0),
    )
    bm.run()
    bm.print()
    assert bm.total_flow == Decimal(1 / 9) + 0, bm.total_flow

```

```
def test_c() -> None:
```

```

    bm = EdmondsKarp(
        [
            ("source", "a", 1),
            ("source", "b", 2),
            ("b", "a", 1),
            ("a", "sink", 2),
            ("b", "sink", 1),
        ],
        main_source="source",
        main_sink="sink",
        zero=0,
    )
    bm.run()
    bm.print()
    assert bm.total_flow == 3, bm.total_flow

```

```
def test_d() -> None:
```

```

    bm = EdmondsKarp([], main_source="source", main_sink="sink", zero=0)
    bm.run()
    bm.print()
    assert bm.total_flow == 0, bm.total_flow

```

```
def test_single_edge() -> None:
```

```

    bm = EdmondsKarp([(0, 1, 5)], main_source=0, main_sink=1, zero=0)
    bm.run()
    assert bm.total_flow == 5

```

```
def test_no_path() -> None:
```

```

    # No path from source to sink
    bm = EdmondsKarp([(0, 1, 5), (2, 3, 5)], main_source=0, main_sink=3, zero=0)
    bm.run()
    assert bm.total_flow == 0

```

```
def test_bottleneck() -> None:
```

```

    # Path with bottleneck
    bm = EdmondsKarp([(0, 1, 100), (1, 2, 1), (2, 3, 100)], main_source=0, main_sink=3, zero=0)
    bm.run()
    assert bm.total_flow == 1

```

```
def test_parallel_edges() -> None:
```

```

    # Multiple parallel paths
    bm = EdmondsKarp(
        [(0, 1, 5), (0, 2, 5), (1, 3, 5), (2, 3, 5)],
        main_source=0,
        main_sink=3,
        zero=0,
    )
    bm.run()
    assert bm.total_flow == 10

```

```
def test_reset_flows() -> None:
```

```

    bm = EdmondsKarp([(0, 1, 10), (1, 2, 10)], main_source=0, main_sink=2, zero=0)
    bm.run()
    assert bm.total_flow == 10

    bm.reset_flows()
    assert bm.total_flow == 0

    bm.run()

```

```
assert bm.total_flow == 10
```

```
def test_change_capacity() -> None:
```

```
    bm = EdmondsKarp([(0, 1, 5), (1, 2, 10)], main_source=0, main_sink=2, zero=0)
```

```
    bm.run()
```

```
    assert bm.total_flow == 5
```

```
    # Increase capacity of bottleneck edge
```

```
    bm.change_initial_capacities([(0, 1, 8)])
```

```
    bm.run()
```

```
    assert bm.total_flow == 8 # Can now push 3 more
```

```
def main() -> None:
```

```
    test_a()
```

```
    print()
```

```
    test_b()
```

```
    print()
```

```
    test_c()
```

```
    print()
```

```
    test_d()
```

```
    print()
```

```
    test_single_edge()
```

```
    test_no_path()
```

```
    test_bottleneck()
```

```
    test_parallel_edges()
```

```
    test_reset_flows()
```

```
    test_change_capacity()
```

```
    test_main()
```

```
if __name__ == "__main__":
```

```
    main()
```

Fenwick Tree

fenwick_tree.py

```
"""
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.
"""
```

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar
```

```
from typing_extensions import Self
```

```
class Summable(Protocol):
    def __add__(self, other: Self, /) -> Self: ...
    def __sub__(self, other: Self, /) -> Self: ...
    def __le__(self, other: Self, /) -> bool: ...
```

```
ValueT = TypeVar("ValueT", bound=Summable)
```

```
class FenwickTree(Generic[ValueT]):
    def __init__(self, size: int, zero: ValueT) -> None:
        self.size: Final = size
        self.zero: Final = zero
        # 1-indexed tree for easier bit manipulation
        self.tree: list[ValueT] = [zero] * (size + 1)

    @classmethod
    def from_array(cls, arr: list[ValueT], zero: ValueT) -> Self:
        """Create a Fenwick tree from an existing array in  $O(n)$  time."""
        n = len(arr)
        tree = cls(n, zero)

        # Compute prefix sums
        prefix = [zero] * (n + 1)
        for i in range(n):
            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 i in range(1, n + 1):
            range_start = i - (i & (-i)) + 1
            tree.tree[i] = prefix[i] - prefix[range_start - 1]

        return tree

    def update(self, index: int, delta: ValueT) -> None:
        """Add delta to the element at the given index."""
        if not (0 <= index < self.size):
            msg = f"Index {index} out of bounds for size {self.size}"
            raise IndexError(msg)

        # Convert to 1-indexed
        index += 1
        while index <= self.size:
            self.tree[index] = self.tree[index] + delta
            # Move to next index by adding the lowest set bit
            index += index & (-index)
```

```

def query(self, index: int) -> ValueT:
    """Return the sum of elements from 0 to index (inclusive)."""
    if not (0 <= index < self.size):
        msg = f"Index {index} out of bounds for size {self.size}"
        raise IndexError(msg)

    # Convert to 1-indexed
    index += 1
    result = self.zero
    while index > 0:
        result = result + self.tree[index]
        # Move to parent by removing the lowest set bit
        index -= index & (-index)
    return result

def range_query(self, left: int, right: int) -> ValueT:
    """Sum of elements from left to right (inclusive). Returns zero for invalid ranges."""
    if left > right or left < 0 or right >= self.size:
        return self.zero
    if left == 0:
        return self.query(right)
    return self.query(right) - self.query(left - 1)

# Optional functionality (not always needed during competition)

def get_value(self, index: int) -> ValueT:
    """Get the current value at a specific index."""
    if not (0 <= index < self.size):
        msg = f"Index {index} out of bounds for size {self.size}"
        raise IndexError(msg)
    if index == 0:
        return self.query(0)
    return self.query(index) - self.query(index - 1)

def first_nonzero_index(self, start_index: int) -> int | None:
    """Find smallest index >= start_index with value > zero.

    REQUIRES: all updates are non-negative, ValueT is totally ordered (e.g., int, float).
    """
    start_index = max(start_index, 0)
    if start_index >= self.size:
        return None

    prefix_before = self.query(start_index - 1) if start_index > 0 else self.zero
    total = self.query(self.size - 1)
    if total == prefix_before:
        return None

    # Fenwick lower_bound: first idx with prefix_sum(idx) > prefix_before
    idx = 0 # 1-based cursor
    cur = self.zero # running prefix at 'idx'
    bit = 1 << (self.size.bit_length() - 1)
    while bit:
        nxt = idx + bit
        if nxt <= self.size:
            cand = cur + self.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

def __len__(self) -> int:
    return self.size

def test_main() -> None:
    f = FenwickTree(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
g = FenwickTree.from_array([1, 2, 3, 4, 5], 0)
assert g.query(4) == 15
```

Don't write tests below during competition.

```
def test_basic() -> None:
```

```
    # Test with integers
    ft = FenwickTree(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
```

```
def test_from_array() -> None:
```

```
    arr = [1, 3, 5, 7, 9, 11]
    ft = FenwickTree.from_array(arr, 0)
```

```
    # Test that prefix sums match
    expected_sum = 0
    for i in range(len(arr)):
        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
```

```
def test_edge_cases() -> None:
```

```
    ft = FenwickTree(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
    ft_large = FenwickTree(10, 0)
    assert ft_large.range_query(5, 3) == 0 # left > right
```

```

def test_bounds_checking() -> None:
    """Test that out-of-bounds access raises appropriate errors."""
    ft = FenwickTree(5, 0)

    # Test update bounds
    try:
        ft.update(-1, 10)
        assert False, "Should raise IndexError for negative index"
    except IndexError:
        pass

    try:
        ft.update(5, 10)
        assert False, "Should raise IndexError for index >= size"
    except IndexError:
        pass

    # Test query bounds
    try:
        ft.query(-1)
        assert False, "Should raise IndexError for negative index"
    except IndexError:
        pass

    try:
        ft.query(5)
        assert False, "Should raise IndexError for index >= size"
    except IndexError:
        pass

    # 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
    try:
        ft.get_value(-1)
        assert False, "Should raise IndexError for negative index"
    except IndexError:
        pass

    try:
        ft.get_value(5)
        assert False, "Should raise IndexError for index >= size"
    except IndexError:
        pass

def test_first_nonzero_bounds() -> None:
    """Test first_nonzero_index with boundary conditions."""
    ft = FenwickTree(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) is None
    assert ft.first_nonzero_index(100) is None

    # Empty tree
    ft_empty = FenwickTree(10, 0)
    assert ft_empty.first_nonzero_index(0) is None

def test_negative_values() -> None:
    ft = FenwickTree(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)

```

```

def test_linear_from_array() -> None:
    """Test that the optimized from_array produces identical results."""
    import time

    # Test arrays of different sizes
    test_cases = [
        [1, 3, 5, 7, 9, 11],
        [10, -5, 8, -3, 15, 2, -7, 12],
        list(range(100)),
    ]

    for arr in test_cases:
        ft = FenwickTree.from_array(arr, 0)

        # Verify all prefix sums match expected
        expected_sum = 0
        for i in range(len(arr)):
            expected_sum += arr[i]
            assert ft.query(i) == expected_sum, f"Mismatch at index {i}"

        # Verify individual values
        for i in range(len(arr)):
            assert ft.get_value(i) == arr[i], f"Value mismatch at index {i}"

        # Test range queries
        if len(arr) >= 3:
            assert ft.range_query(1, 2) == sum(arr[1:3])

    # Simple performance comparison for large array
    large_arr = list(range(1000))

    # Time the optimized version (should be faster)
    start = time.perf_counter()
    ft_optimized = FenwickTree.from_array(large_arr, 0)
    optimized_time = time.perf_counter() - start

    # Verify correctness on large array
    for i in [0, 100, 500, 999]:
        expected = sum(large_arr[:i + 1])
        assert ft_optimized.query(i) == expected

    print(f"Linear from_array time for 1000 elements: {optimized_time:.6f}s")

```

```

def test_first_nonzero_index() -> None:
    ft = FenwickTree(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) is None

```

```

def main() -> None:
    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()  
print("All Fenwick tree tests passed!")
```

```
if __name__ == "__main__":  
    main()
```

Kmp

kmp.py

```
"""
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.
"""
```

```
from __future__ import annotations
```

```
def compute_failure_function(pattern: str) -> list[int]:
```

```
    """
    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]
    """
    m = len(pattern)
    failure = [0] * m
    j = 0

    for i in range(1, m):
        while j > 0 and pattern[i] != pattern[j]:
            j = failure[j - 1]

        if pattern[i] == pattern[j]:
            j += 1

        failure[i] = j

    return failure
```

```
def kmp_search(text: str, pattern: str) -> list[int]:
```

```
    """
    Find all starting positions where pattern occurs in text.

    Returns a list of 0-indexed positions where pattern begins in text.
    """
    if not pattern:
        return []

    n, m = len(text), len(pattern)
    if m > n:
        return []

    failure = compute_failure_function(pattern)
    matches = []
    j = 0 # index for pattern

    for i in range(n): # index for text
        while j > 0 and text[i] != pattern[j]:
            j = failure[j - 1]

        if text[i] == pattern[j]:
            j += 1

        if j == m:
            matches.append(i - m + 1)
            j = failure[j - 1]

    return matches
```

```
def kmp_count(text: str, pattern: str) -> int:
```

```
"""Count number of occurrences of pattern in text."""
return len(kmp_search(text, pattern))
```

```
def test_main() -> None:
    text = "ababcababa"
    pattern = "aba"
    matches = kmp_search(text, pattern)
    assert matches == [0, 5, 7]
    assert kmp_count(text, pattern) == 3

    # Test failure function
    failure = compute_failure_function("abcbcab")
    assert failure == [0, 0, 0, 1, 2, 3, 4, 5]
```

```
# Don't write tests below during competition.
```

```
def test_empty_patterns() -> None:
    # Empty pattern should return empty list
    assert kmp_search("hello", "") == []
    assert kmp_count("hello", "") == 0

    # Empty text with non-empty pattern
    assert kmp_search("", "abc") == []
    assert kmp_count("", "abc") == 0

    # Both empty
    assert kmp_search("", "") == []
    assert kmp_count("", "") == 0
```

```
def test_single_character() -> None:
    # Single character pattern in single character text
    assert kmp_search("a", "a") == [0]
    assert kmp_search("a", "b") == []

    # Single character pattern in longer text
    assert kmp_search("aaaa", "a") == [0, 1, 2, 3]
    assert kmp_search("abab", "a") == [0, 2]
    assert kmp_search("abab", "b") == [1, 3]
```

```
def test_pattern_longer_than_text() -> None:
    assert kmp_search("abc", "abcdef") == []
    assert kmp_search("x", "xyz") == []
    assert kmp_count("short", "verylongpattern") == 0
```

```
def test_overlapping_matches() -> None:
    # Pattern that overlaps with itself
    text = "aaaa"
    pattern = "aa"
    assert kmp_search(text, pattern) == [0, 1, 2]
    assert kmp_count(text, pattern) == 3

    # More complex overlapping
    text = "abababab"
    pattern = "abab"
    assert kmp_search(text, pattern) == [0, 2, 4]
```

```
def test_no_matches() -> None:
    assert kmp_search("abcdef", "xyz") == []
    assert kmp_search("hello world", "goodbye") == []
    assert kmp_count("mississippi", "xyz") == 0
```

```
def test_full_text_match() -> None:
    text = "hello"
    pattern = "hello"
    assert kmp_search(text, pattern) == [0]
    assert kmp_count(text, pattern) == 1
```

```

def test_repeated_patterns() -> None:
    # All same character
    text = "aaaaaaa"
    pattern = "aaa"
    assert kmp_search(text, pattern) == [0, 1, 2, 3, 4]

    # Repeated subpattern
    text = "abcabcabcabc"
    pattern = "abcabc"
    assert kmp_search(text, pattern) == [0, 3, 6]

def test_failure_function_edge_cases() -> None:
    # No repeating prefixes
    failure = compute_failure_function("abcdef")
    assert failure == [0, 0, 0, 0, 0, 0]

    # All same character
    failure = compute_failure_function("aaaa")
    assert failure == [0, 1, 2, 3]

    # Complex pattern with multiple prefix-suffix matches
    failure = compute_failure_function("abcabcabcab")
    assert failure == [0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8]

    # Pattern with internal repetition
    failure = compute_failure_function("abababab")
    assert failure == [0, 0, 1, 2, 0, 1, 2, 3, 4]

def test_case_sensitive() -> None:
    # Should be case sensitive
    assert kmp_search("Hello", "hello") == []
    assert kmp_search("HELLO", "hello") == []
    assert kmp_search("Hello", "H") == [0]
    assert kmp_search("Hello", "h") == []

def test_special_characters() -> None:
    text = "a@b#c$d%e"
    pattern = "@b#"
    assert kmp_search(text, pattern) == [1]

    text = "...test..."
    pattern = "..."
    assert kmp_search(text, pattern) == [0, 7]

def test_large_text_small_pattern() -> None:
    # Large text with small repeated pattern
    text = "a" * 1000 + "b" + "a" * 1000
    pattern = "b"
    assert kmp_search(text, pattern) == [1000]
    assert kmp_count(text, pattern) == 1

    # Pattern at end
    text = "x" * 999 + "target"
    pattern = "target"
    assert kmp_search(text, pattern) == [999]

def test_stress_many_matches() -> None:
    # Many overlapping matches
    text = "a" * 100
    pattern = "a" * 10
    expected = list(range(91)) # Positions 0 through 90
    assert kmp_search(text, pattern) == expected
    assert kmp_count(text, pattern) == 91

def test_binary_strings() -> None:
    # Binary pattern matching
    text = "1010101010"
    pattern = "101"

```

```
assert kmp_search(text, pattern) == [0, 2, 4, 6]
```

```
# No matches in binary
```

```
text = "0000000000"
```

```
pattern = "101"
```

```
assert kmp_search(text, pattern) == []
```

```
def test_periodic_patterns() -> None:
```

```
# Highly periodic pattern
```

```
text = "abababababab"
```

```
pattern = "ababab"
```

```
assert kmp_search(text, pattern) == [0, 2, 4, 6]
```

```
# Pattern is prefix of text
```

```
text = "abcdefghijk"
```

```
pattern = "abcde"
```

```
assert kmp_search(text, pattern) == [0]
```

```
def test_failure_function_comprehensive() -> None:
```

```
# Test various complex failure function cases
```

```
# Palindromic pattern
```

```
failure = compute_failure_function("abacaba")
```

```
assert failure == [0, 0, 1, 0, 1, 2, 3]
```

```
# Pattern with nested repetitions
```

```
failure = compute_failure_function("aabaaaba")
```

```
assert failure == [0, 1, 0, 1, 2, 2, 3, 4]
```

```
# Long repetitive pattern
```

```
failure = compute_failure_function("ababababab")
```

```
assert failure == [0, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
def test_unicode_strings() -> None:
```

```
# Unicode text and pattern
```

```
text = "αβγδεζηθ"
```

```
pattern = "γδε"
```

```
assert kmp_search(text, pattern) == [2]
```

```
text = "😄😄😄😄😄"
```

```
pattern = "😄😄"
```

```
assert kmp_search(text, pattern) == [0, 2]
```

```
def main() -> None:
```

```
test_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()
```

```
if __name__ == "__main__":
```

```
main()
```

Kosaraju Scc

kosaraju_scc.py

"""

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: one on original graph to compute finish times, another on transpose graph to extract SCCs.

Time complexity: $O(V + E)$ where V is vertices and E is edges.

Space complexity: $O(V + E)$ for graph representation and auxiliary structures.

"""

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Generic, TypeVar
```

```
NodeT = TypeVar("NodeT")
```

```
class KosarajuSCC(Generic[NodeT]):
```

```
    def __init__(self) -> None:
```

```
        self.graph: dict[NodeT, list[NodeT]] = {}
```

```
        self.transpose: dict[NodeT, list[NodeT]] = {}
```

```
    def add_edge(self, u: NodeT, v: NodeT) -> None:
```

```
        """Add directed edge from u to v."""
```

```
        if u not in self.graph:
```

```
            self.graph[u] = []
```

```
        if v not in self.graph:
```

```
            self.graph[v] = []
```

```
        if u not in self.transpose:
```

```
            self.transpose[u] = []
```

```
        if v not in self.transpose:
```

```
            self.transpose[v] = []
```

```
        self.graph[u].append(v)
```

```
        self.transpose[v].append(u)
```

```
    def find_sccs(self) -> list[list[NodeT]]:
```

```
        """
```

```
        Find all strongly connected components.
```

```
        Returns list of SCCs, where each SCC is a list of vertices.
```

```
        SCCs are returned in reverse topological order of the condensation graph.
```

```
        """
```

```
        # First DFS pass: compute finish order on original graph
```

```
        visited: set[NodeT] = set()
```

```
        finish_order: list[NodeT] = []
```

```
    def dfs1(node: NodeT) -> None:
```

```
        visited.add(node)
```

```
        if node in self.graph:
```

```
            for neighbor in self.graph[node]:
```

```
                if neighbor not in visited:
```

```
                    dfs1(neighbor)
```

```
        finish_order.append(node)
```

```
    for node in self.graph:
```

```
        if node not in visited:
```

```
            dfs1(node)
```

```
        # Second DFS pass: find SCCs on transpose graph in reverse finish order
```

```
        visited.clear()
```

```
        sccs: list[list[NodeT]] = []
```

```
    def dfs2(node: NodeT, scc: list[NodeT]) -> None:
```

```
        visited.add(node)
```

```
        scc.append(node)
```

```
        if node in self.transpose:
```

```

        for neighbor in self.transpose[node]:
            if neighbor not in visited:
                dfs2(neighbor, scc)

```

```

    for node in reversed(finish_order):
        if node not in visited:
            scc: list[NodeT] = []
            dfs2(node, scc)
            sccs.append(scc)

```

```

    return sccs

```

```

def test_main() -> None:

```

```

    g: KosarajuSCC[int] = KosarajuSCC()
    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)

```

```

    sccs = g.find_sccs()
    assert len(sccs) == 2
    assert sorted([sorted(scc) for scc in sccs]) == [[0, 1, 2], [3, 4, 5]]

```

Don't write tests below during competition.

```

def test_single_node() -> None:

```

```

    g: KosarajuSCC[str] = KosarajuSCC()
    g.add_edge("A", "A")

```

```

    sccs = g.find_sccs()
    assert len(sccs) == 1
    assert sccs[0] == ["A"]

```

```

def test_no_edges() -> None:

```

```

    g: KosarajuSCC[int] = KosarajuSCC()
    g.add_edge(1, 2)
    g.add_edge(3, 4)

```

```

    sccs = g.find_sccs()
    assert len(sccs) == 4
    assert sorted([sorted(scc) for scc in sccs]) == [[1], [2], [3], [4]]

```

```

def test_fully_connected() -> None:

```

```

    g: KosarajuSCC[int] = KosarajuSCC()
    # Create cycle: 0->1->2->3->0
    for i in range(4):
        g.add_edge(i, (i + 1) % 4)

```

```

    sccs = g.find_sccs()
    assert len(sccs) == 1
    assert sorted(sccs[0]) == [0, 1, 2, 3]

```

```

def test_linear_chain() -> None:

```

```

    g: KosarajuSCC[int] = KosarajuSCC()
    # Linear: 0->1->2->3->4
    for i in range(4):
        g.add_edge(i, i + 1)

```

```

    sccs = g.find_sccs()
    assert len(sccs) == 5
    # Each node is its own SCC
    assert sorted([sorted(scc) for scc in sccs]) == [[0], [1], [2], [3], [4]]

```

```

def test_multiple_components() -> None:

```

```

    g: KosarajuSCC[str] = KosarajuSCC()

```



```

# First SCC: A->B->C->A
g.add_edge("A", "B")
g.add_edge("B", "C")
g.add_edge("C", "A")
# Second SCC: D->E->D
g.add_edge("D", "E")
g.add_edge("E", "D")
# Connection between SCCs
g.add_edge("C", "D")

sccs = g.find_sccs()
assert len(sccs) == 2
assert sorted([sorted(scc) for scc in sccs]) == [["A", "B", "C"], ["D", "E"]]

```

```

def test_complex_graph() -> None:
    g: KosarajuSCC[int] = KosarajuSCC()
    # SCC1: 0->1->2->0
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    # SCC2: 3->4->3
    g.add_edge(3, 4)
    g.add_edge(4, 3)
    # SCC3: 5->6->7->5
    g.add_edge(5, 6)
    g.add_edge(6, 7)
    g.add_edge(7, 5)
    # Connections between SCCs
    g.add_edge(2, 3)
    g.add_edge(4, 5)

    sccs = g.find_sccs()
    assert len(sccs) == 3
    assert sorted([sorted(scc) for scc in sccs]) == [[0, 1, 2], [3, 4], [5, 6, 7]]

```

```

def test_single_node_no_self_loop() -> None:
    g: KosarajuSCC[int] = KosarajuSCC()
    g.graph[42] = []
    g.transpose[42] = []

    sccs = g.find_sccs()
    assert len(sccs) == 1
    assert sccs[0] == [42]

```

```

def test_bidirectional_edges() -> None:
    g: KosarajuSCC[int] = KosarajuSCC()
    # Bidirectional edges form SCC
    g.add_edge(1, 2)
    g.add_edge(2, 1)
    g.add_edge(2, 3)
    g.add_edge(3, 2)

    sccs = g.find_sccs()
    assert len(sccs) == 1
    assert sorted(sccs[0]) == [1, 2, 3]

```

```

def test_large_graph() -> None:
    g: KosarajuSCC[int] = KosarajuSCC()
    # Create 10 SCCs, each with 5 nodes in a cycle
    for scc_id in range(10):
        base = scc_id * 5
        for i in range(5):
            g.add_edge(base + i, base + (i + 1) % 5)
        # Connect to next SCC
        if scc_id < 9:
            g.add_edge(base + 4, (scc_id + 1) * 5)

    sccs = g.find_sccs()
    assert len(sccs) == 10
    # Each SCC should have 5 nodes
    for scc in sccs:

```

```
assert len(scc) == 5
```

```
def main() -> None:
    test_main()
    test_single_node()
    test_no_edges()
    test_fully_connected()
    test_linear_chain()
    test_multiple_components()
    test_complex_graph()
    test_single_node_no_self_loop()
    test_bidirectional_edges()
    test_large_graph()

if __name__ == "__main__":
    main()
```

Lca

lca.py

"""

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.

"""

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, TypeVar
```

```
NodeT = TypeVar("NodeT")
```

```
class LCA(Generic[NodeT]):
```

```
    def __init__(self, root: NodeT) -> None:
```

```
        self.root: Final = root
```

```
        self.graph: dict[NodeT, list[NodeT]] = {}
```

```
        self.depth: dict[NodeT, int] = {}
```

```
        self.parent: dict[NodeT, list[NodeT | None]] = {}
```

```
        self.max_log = 0
```

```
    def add_edge(self, u: NodeT, v: NodeT) -> None:
```

```
        """Add undirected edge between u and v."""
```

```
        if u not in self.graph:
```

```
            self.graph[u] = []
```

```
        if v not in self.graph:
```

```
            self.graph[v] = []
```

```
        self.graph[u].append(v)
```

```
        self.graph[v].append(u)
```

```
    def preprocess(self) -> None:
```

```
        """Build the binary lifting table. Call after adding all edges."""
```

```
        # Find max depth to determine log table size
```

```
        self._dfs_depth(self.root, None, 0)
```

```
        nodes = list(self.depth.keys())
```

```
        n = len(nodes)
```

```
        self.max_log = n.bit_length()
```

```
        # Initialize parent table
```

```
        for node in nodes:
```

```
            self.parent[node] = [None] * self.max_log
```

```
        # Fill first column (direct parents) and compute binary lifting
```

```
        self._dfs_parents(self.root, None)
```

```
        # Fill binary lifting table
```

```
        for j in range(1, self.max_log):
```

```
            for node in nodes:
```

```
                parent_j_minus_1 = self.parent[node][j - 1]
```

```
                if parent_j_minus_1 is not None:
```

```
                    self.parent[node][j] = self.parent[parent_j_minus_1][j - 1]
```

```
    def _dfs_depth(self, node: NodeT, par: NodeT | None, d: int) -> None:
```

```
        """Compute depths of all nodes."""
```

```
        self.depth[node] = d
```

```
        for neighbor in self.graph.get(node, []):
```

```
            if neighbor != par:
```

```
                self._dfs_depth(neighbor, node, d + 1)
```

```
    def _dfs_parents(self, node: NodeT, par: NodeT | None) -> None:
```

```
        """Set direct parents for all nodes."""
```

```
        self.parent[node][0] = par
```

```

for neighbor in self.graph.get(node, []):
    if neighbor != par:
        self._dfs_parents(neighbor, node)

def lca(self, u: NodeT, v: NodeT) -> NodeT:
    """Find lowest common ancestor of u and v."""
    if self.depth[u] < self.depth[v]:
        u, v = v, u

    # Bring u to same level as v
    diff = self.depth[u] - self.depth[v]
    for i in range(self.max_log):
        if (diff >> i) & 1:
            u_parent = self.parent[u][i]
            if u_parent is not None:
                u = u_parent

    if u == v:
        return u

    # Binary search for LCA
    for i in range(self.max_log - 1, -1, -1):
        if self.parent[u][i] != self.parent[v][i]:
            u_parent = self.parent[u][i]
            v_parent = self.parent[v][i]
            if u_parent is not None and v_parent is not None:
                u = u_parent
                v = v_parent

    result = self.parent[u][0]
    if result is None:
        msg = "LCA computation failed - invalid tree structure"
        raise ValueError(msg)
    return result

def distance(self, u: NodeT, v: NodeT) -> int:
    """Calculate distance between two nodes."""
    lca_node = self.lca(u, v)
    return self.depth[u] + self.depth[v] - 2 * self.depth[lca_node]

```

```

def test_main() -> None:
    lca = LCA(1)
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)]
    for u, v in 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

```

Don't write tests below during competition.

```

def test_linear_chain() -> None:
    # Test on a simple linear chain: 1-2-3-4-5
    lca = LCA(1)
    edges = [(1, 2), (2, 3), (3, 4), (4, 5)]
    for u, v in 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

def test_single_node() -> None:
    lca = LCA("root")
    lca.preprocess()

    assert lca.lca("root", "root") == "root"
    assert lca.distance("root", "root") == 0

def test_star_graph() -> None:
    # Star graph: center connected to many leaves
    lca = LCA(0)
    for i in range(1, 6):
        lca.add_edge(0, i)

    lca.preprocess()

    # All leaf pairs should have LCA = center
    for i in range(1, 6):
        for j in range(i + 1, 6):
            assert lca.lca(i, j) == 0
            assert lca.distance(i, j) == 2 # Through center

    # Center to leaf
    for i in range(1, 6):
        assert lca.lca(0, i) == 0
        assert lca.distance(0, i) == 1

def test_deep_tree() -> None:
    # Deep binary tree
    lca = LCA(1)
    # Build tree: 1 -> 2,3 2 -> 4,5 3 -> 6,7 4 -> 8,9
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7), (4, 8), (4, 9)]
    for u, v in edges:
        lca.add_edge(u, v)

    lca.preprocess()

    # Test various LCA queries
    assert lca.lca(8, 9) == 4
    assert lca.lca(4, 5) == 2
    assert lca.lca(2, 3) == 1
    assert lca.lca(8, 5) == 2
    assert lca.lca(8, 6) == 1
    assert lca.lca(6, 7) == 3

    # Distance tests
    assert lca.distance(8, 9) == 2
    assert lca.distance(8, 5) == 3
    assert lca.distance(6, 7) == 2
    assert lca.distance(8, 6) == 5

def test_unbalanced_tree() -> None:
    # Highly unbalanced tree (essentially a path with some branches)
    lca = LCA("A")
    edges = [
        ("A", "B"), ("B", "C"), ("C", "D"), ("D", "E"),
        ("B", "X"), ("C", "Y"), ("D", "Z")
    ]
    for u, v in edges:
        lca.add_edge(u, v)

    lca.preprocess()

    assert lca.lca("E", "Z") == "D"
    assert lca.lca("X", "Y") == "B"
    assert lca.lca("X", "E") == "B"
    assert lca.lca("Y", "Z") == "C"

    # Distance in unbalanced tree

```

```
assert lca.distance("X", "E") == 4 # X->B->C->D->E
assert lca.distance("Y", "Z") == 3 # Y->C->D->Z
```

```
def test_large_balanced_tree() -> None:
    # Complete binary tree with 15 nodes (4 levels)
    lca = LCA(1)
    edges = []
    for i in range(1, 8): # Internal nodes
        left_child = 2 * i
        right_child = 2 * i + 1
        if left_child <= 15:
            edges.append((i, left_child))
        if right_child <= 15:
            edges.append((i, right_child))

    for u, v in 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
```

```
def test_string_nodes() -> None:
    # Test with string node labels
    lca = LCA("root")
    edges = [
        ("root", "left"), ("root", "right"),
        ("left", "left_left"), ("left", "left_right"),
        ("right", "right_left"), ("right", "right_right")
    ]
    for u, v in edges:
        lca.add_edge(u, v)

    lca.preprocess()

    assert lca.lca("left_left", "left_right") == "left"
    assert lca.lca("left_left", "right_left") == "root"
    assert lca.distance("left_left", "right_right") == 4
```

```
def test_complex_tree() -> None:
    # More complex tree structure
    lca = LCA(0)
    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 u, v in 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

```

```

def test_edge_cases() -> None:

```

```

# Test edge cases and boundary conditions

```

```

# Tree with only two nodes

```

```

lca = LCA("A")
lca.add_edge("A", "B")
lca.preprocess()

```

```

assert lca.lca("A", "B") == "A"
assert lca.lca("B", "A") == "A"
assert lca.distance("A", "B") == 1

```

```

# Same node queries

```

```

assert lca.lca("A", "A") == "A"
assert lca.lca("B", "B") == "B"

```

```

def test_large_star() -> None:

```

```

# Large star graph to test scalability

```

```

lca = LCA(0)
n = 100
for i in range(1, n + 1):
    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

```

```

def test_long_path() -> None:

```

```

# Very long path to test binary lifting efficiency

```

```

lca = LCA(0)
n = 64 # Power of 2 for clean binary lifting
for i in range(n):
    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

```

```

def test_fibonacci_tree() -> None:

```

```

# Tree based on Fibonacci structure

```

```

lca = LCA(1)
# Build: 1->2,3 2->4,5 3->6 4->7 5->8,9
edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (4, 7), (5, 8), (5, 9)]
for u, v in 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

```

```
def main() -> None:
    test_main()
    test_linear_chain()
    test_single_node()
    test_star_graph()
    test_deep_tree()
    test_unbalanced_tree()
    test_large_balanced_tree()
    test_string_nodes()
    test_complex_tree()
    test_edge_cases()
    test_large_star()
    test_long_path()
    test_fibonacci_tree()

if __name__ == "__main__":
    main()
```


Polygon Area

polygon_area.py

"""

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: $\text{Area} = 1/2 * |\sum(x_i * y_{(i+1)} - x_{(i+1)} * y_i)|$*

Time complexity: $O(n)$ where n is the number of vertices.

Space complexity: $O(1)$ additional space.

"""

```
from __future__ import annotations
```

```
def polygon_area(vertices: list[tuple[float, float]]) -> float:
```

"""

Calculate the area of a polygon using the Shoelace formula.

Args:

vertices: List of (x, y) coordinates in order (clockwise or counter-clockwise)

Returns:

The area of the polygon (always positive)

"""

```
if len(vertices) < 3:
```

```
    return 0.0
```

```
n = len(vertices)
```

```
area = 0.0
```

```
for i in range(n):
```

```
    j = (i + 1) % n
```

```
    area += vertices[i][0] * vertices[j][1]
```

```
    area -= vertices[j][0] * vertices[i][1]
```

```
return abs(area) / 2.0
```

```
def polygon_signed_area(vertices: list[tuple[float, float]]) -> float:
```

"""

Calculate the signed area of a polygon.

Returns positive area for counter-clockwise vertices, negative for clockwise.

Useful for determining polygon orientation.

Args:

vertices: List of (x, y) coordinates in order

Returns:

The signed area (positive for CCW, negative for CW)

"""

```
if len(vertices) < 3:
```

```
    return 0.0
```

```
n = len(vertices)
```

```
area = 0.0
```

```
for i in range(n):
```

```
    j = (i + 1) % n
```

```
    area += vertices[i][0] * vertices[j][1]
```

```
    area -= vertices[j][0] * vertices[i][1]
```

```
return area / 2.0
```

```
def is_clockwise(vertices: list[tuple[float, float]]) -> bool:
```

"""Check if polygon vertices are in clockwise order."""

```
return polygon_signed_area(vertices) < 0
```

```

def test_main() -> None:
    # Simple square with side length 2
    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
    triangle = [(0.0, 0.0), (3.0, 0.0), (1.5, 4.0)]
    assert polygon_area(triangle) == 6.0

    # Test orientation
    ccw_square = [(0.0, 0.0), (1.0, 0.0), (1.0, 1.0), (0.0, 1.0)]
    assert not is_clockwise(ccw_square)

    # Don't write tests below during competition.

def test_rectangle() -> None:
    # Rectangle 5 x 3
    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
    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

def test_triangle_variations() -> None:
    # Right triangle
    tri1 = [(0.0, 0.0), (4.0, 0.0), (0.0, 3.0)]
    assert polygon_area(tri1) == 6.0

    # Same triangle, different order
    tri2 = [(0.0, 3.0), (0.0, 0.0), (4.0, 0.0)]
    assert polygon_area(tri2) == 6.0

    # Equilateral-ish triangle
    tri3 = [(0.0, 0.0), (2.0, 0.0), (1.0, 1.732)]
    area = polygon_area(tri3)
    assert abs(area - 1.732) < 0.01

def test_pentagon() -> None:
    # Regular pentagon (approximate)
    import math
    n = 5
    radius = 1.0
    vertices = []
    for i in range(n):
        angle = 2 * math.pi * i / n
        x = radius * math.cos(angle)
        y = radius * math.sin(angle)
        vertices.append((x, y))

    area = polygon_area(vertices)
    # Area of regular pentagon with radius 1
    expected = 2.377 # approximately
    assert abs(area - expected) < 0.01

def test_concave_polygon() -> None:
    # L-shaped polygon (concave)
    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

def test_degenerate_cases() -> None:
    # 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

```

```

def test_floating_point() -> None:
    # Polygon with floating point coordinates
    poly = [(0.5, 0.5), (3.7, 0.5), (3.7, 2.8), (0.5, 2.8)]
    area = polygon_area(poly)
    expected = (3.7 - 0.5) * (2.8 - 0.5)
    assert abs(area - expected) < 1e-10

```

```

def test_signed_area() -> None:
    # Counter-clockwise square (positive area)
    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 not is_clockwise(ccw)

    # Clockwise square (negative area)
    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)

```

```

def test_large_polygon() -> None:
    # Polygon with many vertices (octagon)
    import math
    n = 8
    radius = 5.0
    vertices = []
    for i in range(n):
        angle = 2 * math.pi * i / n
        x = radius * math.cos(angle)
        y = radius * math.sin(angle)
        vertices.append((x, y))

    area = polygon_area(vertices)
    # Area of regular polygon: (n * r^2 * sin(2π/n)) / 2
    expected = (n * radius * radius * math.sin(2 * math.pi / n)) / 2
    assert abs(area - expected) < 0.01

```

```

def test_negative_coordinates() -> None:
    # Polygon with negative coordinates
    poly = [(-2.0, -1.0), (1.0, -1.0), (1.0, 2.0), (-2.0, 2.0)]
    area = polygon_area(poly)
    expected = 3.0 * 3.0
    assert area == expected

```

```

def test_diamond() -> None:
    # Diamond shape (rhombus)
    diamond = [(0.0, 2.0), (3.0, 0.0), (0.0, -2.0), (-3.0, 0.0)]
    area = polygon_area(diamond)
    # Area = (d1 * d2) / 2 where d1=6, d2=4
    expected = 12.0
    assert area == expected

```

```

def test_integer_coordinates() -> None:
    # Ensure integer coordinates work correctly
    poly = [(0.0, 0.0), (10.0, 0.0), (10.0, 5.0), (0.0, 5.0)]
    area = polygon_area(poly)
    assert area == 50.0

```

```

def main() -> None:
    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()
```

```
if __name__ == "__main__":
    main()
```

Prefix Tree

prefix_tree.py

```
"""
```

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: $O(\text{ALPHABET_SIZE} * N * M)$ in the worst case, where N is the number of strings and M is the average length of strings.*

```
"""
```

```
from __future__ import annotations
```

```
import bisect
```

```
class PrefixTree:
```

```
    def __init__(self) -> None:
```

```
        self.keys: list[str] = []
```

```
        self.values: list[PrefixTree | None] = []
```

```
    def pp(self, indent: int = 0) -> None:
```

```
        """Pretty-print tree structure for debugging."""
```

```
        for key, value in zip(self.keys, self.values): # Note: add strict=False for Python 3.10+
```

```
            print(" " * indent + key + ": " + ("-" if value is None else ""))
```

```
            if value is not None:
```

```
                value.pp(indent + 2)
```

```
    def find_all(self, s: str, offset: int, append_to: list[int]) -> None:
```

```
        """Find all strings in tree that are prefixes of s[offset:]. Appends end positions."""
```

```
        if self.keys and self.keys[0] == "":
```

```
            append_to.append(offset)
```

```
        index = bisect.bisect_left(self.keys, s[offset : offset + 1])
```

```
        if index == len(self.keys):
```

```
            return
```

```
        if s[offset : offset + len(self.keys[index])] == self.keys[index]:
```

```
            pt = self.values[index]
```

```
            if pt is None:
```

```
                append_to.append(offset + len(self.keys[index]))
```

```
            else:
```

```
                pt.find_all(s, offset + len(self.keys[index]), append_to)
```

```
    def max_len(self) -> int:
```

```
        """Return length of longest string in tree."""
```

```
        result = 0
```

```
        for key, value in zip(self.keys, self.values): # Note: add strict=False for Python 3.10+
```

```
            result = max(result, len(key) + (0 if value is None else value.max_len()))
```

```
        return result
```

```
    def add(self, s: str) -> None:
```

```
        """Add string to tree."""
```

```
        if not s or not self.keys:
```

```
            self.keys.insert(0, s)
```

```
            self.values.insert(0, None)
```

```
            return
```

```
        pos = bisect.bisect_left(self.keys, s)
```

```
        if pos and self.keys[pos - 1] and self.keys[pos - 1][0] == s[0]:
```

```
            pos -= 1
```

```
        if pos < len(self.keys) and self.keys[pos][:1] == s[:1]:
```

```
            # Merge
```

```
            if s.startswith(self.keys[pos]):
```

```
                pt = self.values[pos]
```

```
                if pt is None:
```

```
                    child = PrefixTree()
```

```
                    child.keys.append("")
```

```
                    child.values.append(None)
```

```
                    self.values[pos] = pt = child
```

```
                pt.add(s[len(self.keys[pos]) :])
```

```

    elif self.keys[pos].startswith(s):
        child = PrefixTree()
        child.keys.append("")
        child.values.append(None)
        child.keys.append(self.keys[pos][len(s) :])
        child.values.append(self.values[pos])
        self.keys[pos] = s
        self.values[pos] = child
    else:
        prefix = 1
        while s[prefix] == self.keys[pos][prefix]:
            prefix += 1
        child = PrefixTree()
        if s < self.keys[pos]:
            child.keys.append(s[prefix:])
            child.values.append(None)
        child.keys.append(self.keys[pos][prefix:])
        child.values.append(self.values[pos])
        if s >= self.keys[pos]:
            child.keys.append(s[prefix:])
            child.values.append(None)
        self.keys[pos] = s[:prefix]
        self.values[pos] = child
    else:
        self.keys.insert(pos, s)
        self.values.insert(pos, None)

```

```

def test_main() -> None:
    p = PrefixTree()
    p.add("cat")
    p.add("car")
    p.add("card")
    l: list[int] = []
    p.find_all("card", 0, l)
    assert l == [3, 4]
    assert p.max_len() == 4

```

Don't write tests below during competition.

```

def test_empty_tree() -> None:
    p = PrefixTree()
    l: list[int] = []
    p.find_all("test", 0, l)
    assert l == []
    assert p.max_len() == 0

```

```

def test_single_string() -> None:
    p = PrefixTree()
    p.add("hello")
    l: list[int] = []
    p.find_all("hello world", 0, l)
    assert l == [5]
    assert p.max_len() == 5

```

```

def test_empty_string() -> None:
    p = PrefixTree()
    p.add("")
    l: list[int] = []
    p.find_all("anything", 0, l)
    assert l == [0] # Empty string matches at position 0

```

```

def test_no_match() -> None:
    p = PrefixTree()
    p.add("cat")
    p.add("car")
    l: list[int] = []
    p.find_all("dog", 0, l)
    assert l == []

```

```

def test_partial_match() -> None:
    p = PrefixTree()
    p.add("catalog")
    l: list[int] = []
    p.find_all("cat", 0, l)
    assert l == [] # "catalog" is not a prefix of "cat"

def test_overlapping_strings() -> None:
    p = PrefixTree()
    p.add("a")
    p.add("ab")
    p.add("abc")
    l: list[int] = []
    p.find_all("abcdef", 0, l)
    assert l == [1, 2, 3]

def test_different_offsets() -> None:
    p = PrefixTree()
    p.add("test")
    l: list[int] = []
    p.find_all("xxtest", 2, l)
    assert l == [6] # "test" found starting at offset 2, ends at 6

def test_multiple_words() -> None:
    p = PrefixTree()
    words = ["the", "then", "there", "answer", "any", "by", "bye", "their"]
    for word in words:
        p.add(word)

    l: list[int] = []
    p.find_all("their", 0, l)
    # "the", "their" are prefixes of "their"
    assert 3 in l
    assert 5 in l

def test_common_prefix() -> None:
    p = PrefixTree()
    p.add("pre")
    p.add("prefix")
    p.add("prepare")

    l: list[int] = []
    p.find_all("prefix", 0, l)
    assert l == [3, 6] # "pre" and "prefix"

def test_max_len() -> None:
    p = PrefixTree()
    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

def test_duplicate_add() -> None:
    p = PrefixTree()
    p.add("test")
    p.add("test") # Add same string again

    l: list[int] = []
    p.find_all("test", 0, l)
    # Should still work correctly
    assert 4 in l

```

```
def main() -> None:
    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()

if __name__ == "__main__":
    main()
```


Priority Queue

priority_queue.py

"""

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 Python's heapq module 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.

"""

```
from __future__ import annotations
```

```
import heapq
import itertools
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar, cast
```

```
from typing_extensions import Self
```

```
class Comparable(Protocol):
    def __lt__(self, other: Self, /) -> bool: ...
```

```
KeyT = TypeVar("KeyT")
PriorityT = TypeVar("PriorityT", bound=Comparable)
```

```
class PriorityQueue(Generic[KeyT, PriorityT]):
    _REMOVED: Final = object() # placeholder for a removed task

    def __init__(self) -> None:
        self._size = 0
        # list of entries arranged in a heap
        self._pq: list[list[object]] = []
        # mapping of tasks to entries
        self._entry_finder: dict[KeyT, list[object]] = {}
        self._counter: Final = itertools.count() # unique sequence count

    def __setitem__(self, key: KeyT, priority: PriorityT) -> None:
        """Add new task or update priority. Lower priority = popped first."""
        if key in self._entry_finder:
            self.remove(key)
        self._size += 1
        entry = [priority, key]
        self._entry_finder[key] = entry
        heapq.heappush(self._pq, entry)

    def remove(self, key: KeyT) -> None:
        """Remove task by key. Raises KeyError if not found."""
        entry = self._entry_finder.pop(key)
        entry[-1] = PriorityQueue._REMOVED
        self._size -= 1

    def pop(self) -> tuple[KeyT, PriorityT]:
        """Remove and return (key, priority) with lowest priority. Raises KeyError if empty."""
        while self._pq:
            priority, task = heapq.heappop(self._pq)
            if task is not PriorityQueue._REMOVED:
                del self._entry_finder[cast("KeyT", task)]
                self._size -= 1
                return cast("tuple[KeyT, PriorityT]", (task, priority))
```

```

        raise KeyError("pop from an empty priority queue")

def peek(self) -> tuple[KeyT, PriorityT] | None:
    """Return (key, priority) with lowest priority without removing. Returns None if empty."""
    while self._pq:
        priority, task = self._pq[0]
        if task is not PriorityQueue._REMOVED:
            return cast("tuple[KeyT, PriorityT]", (task, priority))
        # Remove the REMOVED sentinel from the top
        heapq.heappop(self._pq)
    return None

def __contains__(self, key: KeyT) -> bool:
    """Check if key exists in queue."""
    return key in self._entry_finder

def __len__(self) -> int:
    return self._size

def test_main() -> None:
    p: PriorityQueue[str, int] = PriorityQueue()
    p["x"] = 15
    p["y"] = 23
    p["z"] = 8
    assert p.peek() == ("z", 8)
    assert p.pop() == ("z", 8)
    assert p.pop() == ("x", 15)

# Don't write tests below during competition.

def test_basic_operations() -> None:
    """Test basic add, pop, and len operations."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    # Test empty queue
    assert len(pq) == 0
    assert pq.peek() is None

    # Add items
    pq["task1"] = 10
    pq["task2"] = 5
    pq["task3"] = 15

    assert len(pq) == 3
    assert pq.peek() == ("task2", 5)

    # Pop in priority order
    assert pq.pop() == ("task2", 5)
    assert len(pq) == 2
    assert pq.pop() == ("task1", 10)
    assert pq.pop() == ("task3", 15)

    assert len(pq) == 0

def test_update_priority() -> None:
    """Test updating the priority of existing tasks."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    pq["task1"] = 10
    pq["task2"] = 5

    # Update task1 to have higher priority
    pq["task1"] = 3
    assert pq.peek() == ("task1", 3)
    assert len(pq) == 2

    # Pop should now give task1 first
    assert pq.pop() == ("task1", 3)
    assert pq.pop() == ("task2", 5)

```

```

def test_remove() -> None:
    """Test removing tasks from the queue."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    pq["task1"] = 10
    pq["task2"] = 5
    pq["task3"] = 15

    # Remove middle priority task
    pq.remove("task1")
    assert len(pq) == 2
    assert "task1" not in pq

    # Verify correct items remain
    assert pq.pop() == ("task2", 5)
    assert pq.pop() == ("task3", 15)

def test_contains() -> None:
    """Test membership checking."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    pq["task1"] = 10
    pq["task2"] = 5

    assert "task1" in pq
    assert "task2" in pq
    assert "task3" not in pq

    pq.remove("task1")
    assert "task1" not in pq

def test_empty_operations() -> None:
    """Test operations on empty queue."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    # Test peek on empty queue
    assert pq.peek() is None

    # Test pop on empty queue
    try:
        pq.pop()
        assert False, "Should raise KeyError"
    except KeyError:
        pass

def test_remove_nonexistent() -> None:
    """Test removing a key that doesn't exist."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    pq["task1"] = 10

    try:
        pq.remove("nonexistent")
        assert False, "Should raise KeyError"
    except KeyError:
        pass

def test_single_element() -> None:
    """Test queue with single element."""
    pq: PriorityQueue[str, int] = PriorityQueue()

    pq["only"] = 42
    assert len(pq) == 1
    assert pq.peek() == ("only", 42)
    assert pq.pop() == ("only", 42)
    assert len(pq) == 0

def test_duplicate_priorities() -> None:
    """Test tasks with the same priority."""
    pq: PriorityQueue[str, int] = PriorityQueue()

```

```
pq["task1"] = 10
pq["task2"] = 10
pq["task3"] = 10

assert len(pq) == 3

# All should pop eventually
results = [pq.pop(), pq.pop(), pq.pop()]
assert len(results) == 3
assert all(priority == 10 for _, priority in results)
```

```
def test_with_floats() -> None:
    """Test priority queue with floating point priorities."""
    pq: PriorityQueue[str, float] = PriorityQueue()

    pq["a"] = 1.5
    pq["b"] = 0.5
    pq["c"] = 2.3

    assert pq.pop() == ("b", 0.5)
    assert pq.pop() == ("a", 1.5)
    assert pq.pop() == ("c", 2.3)
```

```
def main() -> None:
    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()
    print("All priority queue tests passed!")
```

```
if __name__ == "__main__":
    main()
```

Segment Tree

segment_tree.py

"""

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.

"""

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Final, Generic, Protocol, TypeVar
```

```
from typing_extensions import Self
```

```
class Summable(Protocol):
```

```
    def __add__(self, other: Self, /) -> Self: ...
```

```
ValueT = TypeVar("ValueT", bound=Summable)
```

```
class SegmentTree(Generic[ValueT]):
```

```
    def __init__(self, arr: list[ValueT], zero: ValueT) -> None:
```

```
        self.n: Final = len(arr)
```

```
        self.zero: Final = zero
```

```
        # Tree needs 4*n space for worst case
```

```
        self.tree: list[ValueT] = [zero] * (4 * self.n)
```

```
        if arr:
```

```
            self._build(arr, 1, 0, self.n - 1)
```

```
    def _build(self, arr: list[ValueT], node: int, start: int, end: int) -> None:
```

```
        if start == end:
```

```
            self.tree[node] = arr[start]
```

```
        else:
```

```
            mid = (start + end) // 2
```

```
            self._build(arr, 2 * node, start, mid)
```

```
            self._build(arr, 2 * node + 1, mid + 1, end)
```

```
            self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]
```

```
    def update(self, idx: int, val: ValueT) -> None:
```

```
        """Update value at index idx to val."""
```

```
        if not (0 <= idx < self.n):
```

```
            msg = f"Index {idx} out of bounds for size {self.n}"
```

```
            raise IndexError(msg)
```

```
        self._update(1, 0, self.n - 1, idx, val)
```

```
    def _update(self, node: int, start: int, end: int, idx: int, val: ValueT) -> None:
```

```
        if start == end:
```

```
            self.tree[node] = val
```

```
        else:
```

```
            mid = (start + end) // 2
```

```
            if idx <= mid:
```

```
                self._update(2 * node, start, mid, idx, val)
```

```
            else:
```

```
                self._update(2 * node + 1, mid + 1, end, idx, val)
```

```
            self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]
```

```
    def query(self, left: int, right: int) -> ValueT:
```

```
        """Query sum of range [left, right] inclusive."""
```

```
        if not (0 <= left <= right < self.n):
```

```
            msg = f"Invalid range [{left}, {right}] for size {self.n}"
```

```
            raise IndexError(msg)
```

```
        return self._query(1, 0, self.n - 1, left, right)
```

```

def _query(self, node: int, start: int, end: int, left: int, right: int) -> ValueT:
    if right < start or left > end:
        return self.zero
    if left <= start and end <= right:
        return self.tree[node]
    mid = (start + end) // 2
    left_sum = self._query(2 * node, start, mid, left, right)
    right_sum = self._query(2 * node + 1, mid + 1, end, left, right)
    return left_sum + right_sum

```

```

def test_main() -> None:
    st = SegmentTree([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.

```

def test_large_array() -> None:
    # Test with large array
    arr = list(range(1000))
    st = SegmentTree(arr, 0)

    # Test various range queries
    assert st.query(0, 99) == sum(range(100))
    assert st.query(500, 599) == sum(range(500, 600))
    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

```

```

def test_edge_cases() -> None:
    # Single element
    st = SegmentTree([42], 0)
    assert st.query(0, 0) == 42
    st.update(0, 100)
    assert st.query(0, 0) == 100

    # Empty array
    st_empty = SegmentTree([], 0)
    assert len(st_empty.tree) == 0

    # All zeros
    st_zeros = SegmentTree([0, 0, 0, 0], 0)
    assert st_zeros.query(0, 3) == 0
    st_zeros.update(2, 5)
    assert st_zeros.query(0, 3) == 5

```

```

def test_negative_values() -> None:
    arr = [-5, 3, -2, 8, -1]
    st = SegmentTree(arr, 0)

    assert st.query(0, 4) == 3 # -5 + 3 + (-2) + 8 + (-1)
    assert st.query(1, 3) == 9 # 3 + (-2) + 8

    st.update(0, -10)
    assert st.query(0, 1) == -7 # -10 + 3

```

```

def test_stress_updates() -> None:
    # Test many updates
    arr = [1] * 100
    st = SegmentTree(arr, 0)

    # Initial sum should be 100
    assert st.query(0, 99) == 100

```

```

# Update every element
for i in range(100):
    st.update(i, i + 1)

# Sum should now be 1 + 2 + ... + 100 = 5050
assert st.query(0, 99) == sum(range(1, 101))

# Test various ranges
assert st.query(0, 9) == sum(range(1, 11))
assert st.query(50, 59) == sum(range(51, 61))

```

```

def test_invalid_operations() -> None:
    st = SegmentTree([1, 2, 3], 0)

    # Test invalid indices
    try:
        st.update(-1, 5)
        assert False, "Should raise IndexError"
    except IndexError:
        pass

    try:
        st.update(3, 5)
        assert False, "Should raise IndexError"
    except IndexError:
        pass

    try:
        st.query(-1, 2)
        assert False, "Should raise IndexError"
    except IndexError:
        pass

    try:
        st.query(0, 3)
        assert False, "Should raise IndexError"
    except IndexError:
        pass

    try:
        st.query(2, 1) # left > right
        assert False, "Should raise IndexError"
    except IndexError:
        pass

```

```

def test_string_summation() -> None:
    # Test with strings as summable values
    arr = ["a", "b", "c", "d"]
    st = SegmentTree(arr, "")

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

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

```

```

def main() -> None:
    test_main()
    test_large_array()
    test_edge_cases()
    test_negative_values()
    test_stress_updates()
    test_invalid_operations()
    test_string_summation()

```

```

if __name__ == "__main__":
    main()

```

Skiplist

skiplist.py

"""

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.

"""

Don't use annotations during contest

from __future__ import annotations

from typing import TYPE_CHECKING, Generic, TypeVar

if TYPE_CHECKING:
 from typing_extensions import Self

import random

T = TypeVar("T")

class SkipListNode(Generic[T]):

def __init__(self, value: T | None, level: int) -> None:
 self.value: T | None = value
 self.forward: list[SkipListNode[T] | None] = [None] * (level + 1)

class SkipList(Generic[T]):

"""Probabilistic data structure for sorted elements with $O(\log n)$ operations."""

def __init__(self, max_level: int = 16, p: float = 0.5) -> None:
 self.max_level = max_level
 self.p = p
 self.level = 0
 self.header: SkipListNode[T] = SkipListNode(None, max_level)

def _random_level(self) -> int:
 level = 0
 while random.random() < self.p and level < self.max_level: # noqa: S311
 level += 1
 return level

def insert(self, value: T) -> Self:
 update: list[SkipListNode[T] | None] = [None] * (self.max_level + 1)
 current: SkipListNode[T] = self.header

for i in range(self.level, -1, -1):
 while (
 current.forward[i] is not None and current.forward[i].value < value # type:
 ignore[union-attr, operator]
):
 current = current.forward[i] # type: ignore[assignment]
 update[i] = current

level = self._random_level()
 if level > self.level:
 for i in range(self.level + 1, level + 1):
 update[i] = self.header
 self.level = level

new_node: SkipListNode[T] = SkipListNode(value, level)


```

for i in range(level + 1):
    new_node.forward[i] = update[i].forward[i] # type: ignore[union-attr]
    update[i].forward[i] = new_node # type: ignore[union-attr]

return self

def search(self, value: T) -> bool:
    current: SkipListNode[T] = self.header
    for i in range(self.level, -1, -1):
        while (
            current.forward[i] is not None and current.forward[i].value < value # type:
            ignore[union-attr, operator]
        ):
            current = current.forward[i] # type: ignore[assignment]
    current = current.forward[0] # type: ignore[assignment]
    return current is not None and current.value == value

def delete(self, value: T) -> bool:
    update: list[SkipListNode[T] | None] = [None] * (self.max_level + 1)
    current: SkipListNode[T] = self.header

    for i in range(self.level, -1, -1):
        while (
            current.forward[i] is not None and current.forward[i].value < value # type:
            ignore[union-attr, operator]
        ):
            current = current.forward[i] # type: ignore[assignment]
            update[i] = current

    current = current.forward[0] # type: ignore[assignment]
    if current is None or current.value != value:
        return False

    for i in range(self.level + 1):
        if update[i].forward[i] != current: # type: ignore[union-attr]
            break
        update[i].forward[i] = current.forward[i] # type: ignore[union-attr]

    while self.level > 0 and self.header.forward[self.level] is None:
        self.level -= 1

    return True

# Optional functionality (not always needed during competition)

def __len__(self) -> int:
    count = 0
    current = self.header.forward[0]
    while current is not None:
        count += 1
        current = current.forward[0]
    return count

def to_list(self) -> list[T]:
    result: list[T] = []
    current = self.header.forward[0]
    while current is not None:
        result.append(current.value) # type: ignore[arg-type]
        current = current.forward[0]
    return result

def __contains__(self, value: T) -> bool:
    return self.search(value)

def test_main() -> None:
    random.seed(42)
    sl: SkipList[int] = SkipList()
    sl.insert(10).insert(20).insert(5).insert(15)
    assert sl.search(10)
    assert sl.search(20)
    assert not sl.search(25)
    assert sl.delete(10)
    assert not sl.search(10)
    assert not sl.delete(30)

```

Optional functionality (not always needed during competition)

```
random.seed(42)
sl2: SkipList[int] = SkipList()
sl2.insert(3).insert(1).insert(4).insert(1).insert(5)
assert len(sl2) == 5
assert sl2.to_list() == [1, 1, 3, 4, 5]
assert 3 in sl2
assert 7 not in sl2
```

Don't write tests below during competition.

```
def test_basic_operations() -> None:
```

```
    random.seed(123)
    sl: SkipList[int] = SkipList()
    assert not sl.search(1)
    sl.insert(5)
    assert sl.search(5)
    assert not sl.search(4)
```

```
def test_multiple_inserts() -> None:
```

```
    random.seed(456)
    sl: SkipList[int] = SkipList()
    values = [10, 5, 15, 3, 7, 12, 20]
    for v in values:
        sl.insert(v)
    for v in values:
        assert sl.search(v)
    assert not sl.search(1)
    assert not sl.search(100)
```

```
def test_delete_operations() -> None:
```

```
    random.seed(789)
    sl: SkipList[int] = SkipList()
    sl.insert(10).insert(20).insert(30)
    assert sl.delete(20)
    assert not sl.search(20)
    assert sl.search(10)
    assert sl.search(30)
    assert not sl.delete(20)
    assert not sl.delete(40)
```

```
def test_duplicate_values() -> None:
```

```
    random.seed(101)
    sl: SkipList[int] = SkipList()
    sl.insert(5).insert(5).insert(5)
    assert len(sl) == 3
    result = sl.to_list()
    assert result == [5, 5, 5]
```

```
def test_ordered_insertion() -> None:
```

```
    random.seed(202)
    sl: SkipList[int] = SkipList()
    for i in range(1, 11):
        sl.insert(i)
    assert sl.to_list() == list(range(1, 11))
```

```
def test_reverse_insertion() -> None:
```

```
    random.seed(303)
    sl: SkipList[int] = SkipList()
    for i in range(10, 0, -1):
        sl.insert(i)
    assert sl.to_list() == list(range(1, 11))
```

```
def test_empty_skiplist() -> None:
```

```
    random.seed(404)
    sl: SkipList[int] = SkipList()
```

```
assert len(sl) == 0
assert sl.to_list() == []
assert not sl.delete(5)
```

```
def test_strings() -> None:
    random.seed(505)
    sl: SkipList[str] = SkipList()
    sl.insert("dog").insert("cat").insert("bird").insert("ant")
    assert sl.search("cat")
    assert sl.to_list() == ["ant", "bird", "cat", "dog"]
```

```
def main() -> None:
    test_basic_operations()
    test_multiple_inserts()
    test_delete_operations()
    test_duplicate_values()
    test_ordered_insertion()
    test_reverse_insertion()
    test_empty_skiplist()
    test_strings()
    test_main()
```

```
if __name__ == "__main__":
    main()
```

Sprague Grundy

sprague_grundy.py

```
"""
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(get_valid_moves): 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 tuple representation.
```

```
Requirements:
- State must be hashable and canonically represented (e.g., tuple/sorted tuple).
- get_valid_moves(state) must not create cycles.
"""
```

```
# Don't use annotations during contest
from __future__ import annotations
```

```
from collections.abc import Hashable, Iterable
from functools import cache
from typing import Callable, Final
```

```
State = Hashable
MovesFn = Callable[[State], Iterable[State]]
```

```
def mex(values: Iterable[int]) -> int:
    """Minimum EXcludant: smallest non-negative integer not occurring in 'values'."""
    s = set(values)
    g = 0
    while g in s:
        g += 1
    return g
```

```
class GrundyEngine:
    """Wrapper that binds a move function and provides grundy(), XOR operations, etc."""
    def __init__(self, get_valid_moves: MovesFn) -> None:
        self._moves: Final = get_valid_moves

        @cache
        def _grundy_cached(state: State) -> int:
            nxt = tuple(self._moves(state))
            if not nxt:
                return 0
            return mex(_grundy_cached(s) for s in nxt)

        self._grundy_cached = _grundy_cached

    def grundy(self, state: State) -> int:
        return self._grundy_cached(state)

    def grundy_multi(self, states: Iterable[State]) -> int:
        g = 0
        for s in states:
            g ^= self._grundy_cached(s)
        return g

    def is_winning_position(self, states: Iterable[State]) -> bool:
        return self.grundy_multi(states) != 0
```

Optional functionality (not always needed during competition)

```
def detect_period(seq: list[int], min_period: int = 1, max_period: int | None = None) -> int | None:
    """Find smallest period p such that seq repeats (completely) with period p."""
    n = len(seq)
    if max_period is None:
        max_period = n // 2
    for p in range(min_period, max_period + 1):
        ok = True
        for i in range(n):
            if seq[i] != seq[i % p]:
                ok = False
                break
        if ok:
            return p
    return None
```

```
def nim_moves_single_heap(n: Hashable) -> Iterable[Hashable]:
    """Nim: single heap. Move: take 1..n stones."""
    assert isinstance(n, int)
    yield from range(n) # leave 0..n-1
```

```
def subtraction_game_moves_factory(allowed: set[int]) -> Callable[[Hashable], Iterable[Hashable]]:
    """Subtraction game: state = int; allowed moves = move sizes in 'allowed'."""
    allowed_sorted = tuple(sorted(allowed))
    def moves(n: Hashable) -> Iterable[Hashable]:
        assert isinstance(n, int)
        for d in allowed_sorted:
            if d <= n:
                yield n - d
    return moves
```

```
def kayles_moves(segments: Hashable) -> Iterable[Hashable]:
    """
    Kayles (bowling pins):
    State: sorted tuple of segment lengths. A move removes 1 pin or 2 adjacent pins in one segment,
    thus splitting it into up to two new segments. Return new canonical (sorted) state.

    Args:
        segments: A tuple[int, ...] representing segment lengths (sorted).

    Returns:
        Set of new states (each a tuple[int, ...]).
    """
    assert isinstance(segments, tuple), "segments must be a tuple of ints"
    res: set[tuple[int, ...]] = set()
    for idx, n in enumerate(segments):
        if n <= 0:
            continue

        # Remove one pin at position i (0..n-1)
        for i in range(n):
            left = i
            right = n - i - 1
            new_seg = [*segments[:idx]]
            if left > 0:
                new_seg.append(left)
            if right > 0:
                new_seg.append(right)
            new_seg.extend(segments[idx + 1 :])
            res.add(tuple(sorted(new_seg)))

        # Remove two adjacent pins at position i,i+1 (0..n-2)
        for i in range(n - 1):
            left = i
            right = n - i - 2
            new_seg = [*segments[:idx]]
            if left > 0:
                new_seg.append(left)
            if right > 0:
                new_seg.append(right)
```

```
new_seg.extend(segments[idx + 1 :])
res.add(tuple(sorted(new_seg)))
```

```
return res
```

```
def test_main() -> None:
```

```
# Test Nim with larger values
eng = GrundyEngine(nim_moves_single_heap)
assert eng.grundy(42) == 42
assert eng.grundy_multi([17, 23, 31]) == 25 #  $17 \oplus 23 \oplus 31 = 25$ 
assert eng.is_winning_position([15, 27, 36]) is True #  $15 \oplus 27 \oplus 36 = 48 \neq 0$ 
```

```
# Test subtraction game {1,3,4} with period 7
eng2 = GrundyEngine(subtraction_game_moves_factory({1, 3, 4}))
assert eng2.grundy(14) == 0 #  $14 \% 7 = 0 \rightarrow \text{grundy} = 0$ 
assert eng2.grundy(15) == 1 #  $15 \% 7 = 1 \rightarrow \text{grundy} = 1$ 
assert eng2.grundy(18) == 2 #  $18 \% 7 = 4 \rightarrow \text{grundy} = 2$ 
```

```
# Test Kayles
eng3 = GrundyEngine(kayles_moves)
assert eng3.grundy((7,)) == 2 #  $K(7) = 2$ 
assert eng3.grundy((3, 5)) == 7 #  $K(3) \oplus K(5) = 3 \oplus 4 = 7$ 
```

```
# Don't write tests below during competition.
```

```
def test_nim_extended() -> None:
```

```
eng = GrundyEngine(nim_moves_single_heap)
# Known:  $\text{grundy}(n) = n$  for all  $n$  in Nim
for n in range(64):
    assert eng.grundy(n) == n
```

```
def test_subtraction_game_period() -> None:
```

```
# Allowed moves = {1,3,4}. Classic periodic sequence.
moves = subtraction_game_moves_factory({1, 3, 4})
eng = GrundyEngine(moves)

seq = [eng.grundy(n) for n in range(200)]
# For {1,3,4} the period is 7: [0,1,0,1,2,3,2] ...
p = detect_period(seq, min_period=1, max_period=50)
assert p == 7
```

```
base = seq[:p]
# Check repetition
for i, g in enumerate(seq):
    assert g == base[i % p]
```

```
# Winning N: those with  $\text{grundy}(n) \neq 0$ 
wins = [n for n, g in enumerate(seq[:30]) if g != 0]
assert wins[:10] == [1, 3, 4, 5, 6, 8, 10, 11, 12, 13]
```

```
def test_sum_of_independent_subgames() -> None:
```

```
# Same subtraction game. Combined position = multiple independent heaps (ints).
eng = GrundyEngine(subtraction_game_moves_factory({1, 3, 4}))

# Build some positions
A = [5, 7] #  $\text{grundy}(5)=3, \text{grundy}(7)=2 \rightarrow \text{XOR}=1 \rightarrow \text{winning}$ 
B = [8, 9] # Let's compute what  $g(8)$  and  $g(9)$  are:
GA = eng.grundy_multi(A)
GB = eng.grundy_multi(B)
assert GA != 0
assert (eng.grundy(8) ^ eng.grundy(9)) == GB
assert eng.is_winning_position(A) is True
assert eng.is_winning_position(B) == (GB != 0)
```

```
def test_kayles_small() -> None:
```

```
eng = GrundyEngine(kayles_moves)

# Known first values for  $K(n)$  (reasonably small  $n$ )
# Not all precisely known by heart, but we validate consistency/monotone checks.
```

```

vals = [eng.grundy((n,)) for n in range(15)]
# Not trivial pattern; we check a few hand-picked facts (from direct computation):
assert vals[:10] == [0, 1, 2, 3, 1, 4, 3, 2, 1, 4]
# Splits: (n,) can end in (a,b) → XOR rule implicit in recursion.
# Extra sanity: composite segments
assert eng.grundy((2, 2)) == (eng.grundy((2,)) ^ eng.grundy((2,)))

```

```

def test_long_application_scan() -> None:

```

```

    """
    Typical competition application:
    - Given a parameter N, derive for which N the position is winning.
    - Use period if it exists.
    Here we use subtraction game {1,3,4}.
    """
    eng = GrundyEngine(subtraction_game_moves_factory({1, 3, 4}))
    N = 500
    seq = [eng.grundy(n) for n in range(N + 1)]
    period = detect_period(seq, min_period=1, max_period=100)
    assert period == 7

    # Winning N up to 60:
    winning_N = [n for n in range(61) if seq[n] != 0]
    # Spot-check the first few values
    assert winning_N[:12] == [1, 3, 4, 5, 6, 8, 10, 11, 12, 13, 15, 17]

```

```

def test_cycle_guard_note() -> None:

```

```

    """
    Theory requirement: no cycles. This test is 'meta' and documents the assumption.
    We do NOT build a cyclic moves function here; we just note the requirement.
    """
    assert True

```

```

def test() -> None:

```

```

    test_nim_extended()
    test_subtraction_game_period()
    test_sum_of_independent_subgames()
    test_kayles_small()
    test_long_application_scan()
    test_cycle_guard_note()

```

```

def main() -> None:

```

```

    test_main()
    test()
    print("All Sprague-Grundy tests passed!")

```

```

if __name__ == "__main__":

```

```

    main()

```

Suffix Array

suffix_array.py

"""

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

A suffix array is a sorted array of all suffixes of a string. The LCP array stores the length of the longest common prefix between consecutive suffixes in the suffix array. These structures enable efficient string pattern matching and various string algorithms.

Time complexity: $O(n \log n)$ for suffix array construction, $O(n)$ for LCP array.

Space complexity: $O(n)$ for suffix array and LCP array.

"""

```
from __future__ import annotations
```

```
# Don't use annotations during contest
```

```
from typing import Final
```

```
class SuffixArray:
```

```
    def __init__(self, text: str) -> None:
```

```
        self.text: Final[str] = text
```

```
        self.n: Final[int] = len(text)
```

```
        self.sa: list[int] = self._build_suffix_array()
```

```
        self.lcp: list[int] = self._build_lcp_array()
```

```
    def _build_suffix_array(self) -> list[int]:
```

```
        """Build suffix array using Python's sort with custom key."""
```

```
        suffixes = list(range(self.n))
```

```
        suffixes.sort(key=lambda i: self.text[i:])
```

```
        return suffixes
```

```
    def _build_lcp_array(self) -> list[int]:
```

```
        """
```

```
        Build LCP array using Kasai's algorithm.
```

```
        lcp[i] = length of longest common prefix between sa[i] and sa[i-1].
```

```
        lcp[0] is defined as 0.
```

```
        """
```

```
        if self.n == 0:
```

```
            return []
```

```
        # Build rank array: rank[i] = position of suffix i in suffix array
```

```
        rank = [0] * self.n
```

```
        for i in range(self.n):
```

```
            rank[self.sa[i]] = i
```

```
        lcp = [0] * self.n
```

```
        h = 0 # Length of current LCP
```

```
        for i in range(self.n):
```

```
            if rank[i] > 0:
```

```
                j = self.sa[rank[i] - 1] # Previous suffix in sorted order
```

```
                # Extend LCP from previous calculation
```

```
                while i + h < self.n and j + h < self.n and self.text[i + h] == self.text[j + h]:
```

```
                    h += 1
```

```
                lcp[rank[i]] = h
```

```
                if h > 0:
```

```
                    h -= 1
```

```
        return lcp
```

```
    def find_pattern(self, pattern: str) -> list[int]:
```

```
        """
```

```
        Find all occurrences of pattern in text.
```

```
        Returns list of starting positions where pattern occurs, in sorted order.
```

```
        """
```

```
        if not pattern:
```

```
            return []
```



```

m = len(pattern)
# Binary search for first occurrence
left, right = 0, self.n

# Find leftmost position where suffix >= pattern
while left < right:
    mid = (left + right) // 2
    suffix = self.text[self.sa[mid]:]
    if suffix < pattern:
        left = mid + 1
    else:
        right = mid

start = left

# Find rightmost position where suffix starts with pattern
left, right = start, self.n
while left < right:
    mid = (left + right) // 2
    suffix = self.text[self.sa[mid]:self.sa[mid] + m]
    if suffix <= pattern:
        left = mid + 1
    else:
        right = mid

end = left

# Collect all matching positions
result = [
    self.sa[i]
    for i in range(start, end)
    if self.text[self.sa[i]:self.sa[i] + m] == pattern
]
return sorted(result)

```

```

def test_main() -> None:
    # Test suffix array and LCP for "banana"
    sa = SuffixArray("banana")
    # Suffixes in sorted order: "a", "ana", "anana", "banana", "na", "nana"
    # Corresponding starting positions: 5, 3, 1, 0, 4, 2
    assert sa.sa == [5, 3, 1, 0, 4, 2]
    # LCP: [0, 1, 3, 0, 0, 2]
    assert sa.lcp == [0, 1, 3, 0, 0, 2]

    # Test pattern finding
    positions = sa.find_pattern("ana")
    assert positions == [1, 3]

```

Don't write tests below during competition.

```

def test_empty_string() -> None:
    sa = SuffixArray("")
    assert sa.sa == []
    assert sa.lcp == []
    assert sa.find_pattern("a") == []

```

```

def test_single_char() -> None:
    sa = SuffixArray("a")
    assert sa.sa == [0]
    assert sa.lcp == [0]

```

```

def test_repeated_chars() -> None:
    sa = SuffixArray("aaaa")
    assert sa.sa == [3, 2, 1, 0]
    # LCP: all have LCP with previous (except first)
    assert sa.lcp == [0, 1, 2, 3]

```

```

def test_pattern_not_found() -> None:
    sa = SuffixArray("hello")

```

```

assert sa.find_pattern("world") == []

def test_pattern_at_end() -> None:
    sa = SuffixArray("hello")
    assert sa.find_pattern("lo") == [3]

def test_overlapping_patterns() -> None:
    sa = SuffixArray("aabaabaa")
    positions = sa.find_pattern("aa")
    assert sorted(positions) == [0, 3, 6]

def test_entire_string() -> None:
    text = "programming"
    sa = SuffixArray(text)
    assert sa.find_pattern(text) == [0]

def test_lcp_calculation() -> None:
    sa = SuffixArray("abcbab")
    # Suffixes sorted: "ab", "abcbab", "b", "bcab", "cab"
    # Positions: 3, 0, 4, 1, 2
    assert sa.sa == [3, 0, 4, 1, 2]
    # LCP between consecutive suffixes
    assert sa.lcp[0] == 0 # First has no predecessor
    assert sa.lcp[1] == 2 # "ab" and "abcbab" share "ab"
    assert sa.lcp[2] == 0 # "abcbab" and "b" share nothing
    assert sa.lcp[3] == 1 # "b" and "bcab" share "b"
    assert sa.lcp[4] == 0 # "bcab" and "cab" share nothing

def test_all_unique_chars() -> None:
    sa = SuffixArray("abcd")
    assert sa.sa == [0, 1, 2, 3]
    assert sa.lcp == [0, 0, 0, 0]

def test_palindrome() -> None:
    sa = SuffixArray("racecar")
    positions = sa.find_pattern("r")
    assert sorted(positions) == [0, 6]

def test_long_pattern() -> None:
    text = "thequickbrownfoxjumpsoverthelazydog"
    sa = SuffixArray(text)
    assert sa.find_pattern("jumps") == [16]
    assert sa.find_pattern("the") == [0, 25]

def main() -> None:
    test_main()
    test_empty_string()
    test_single_char()
    test_repeated_chars()
    test_pattern_not_found()
    test_pattern_at_end()
    test_overlapping_patterns()
    test_entire_string()
    test_lcp_calculation()
    test_all_unique_chars()
    test_palindrome()
    test_long_pattern()

if __name__ == "__main__":
    main()

```

Topological Sort

topological_sort.py

```
"""
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.
"""
```

```
from __future__ import annotations
```

```
from collections import deque
```

```
# Don't use annotations during contest
from typing import Generic, TypeVar
```

```
NodeT = TypeVar("NodeT")
```

```
class TopologicalSort(Generic[NodeT]):
```

```
    def __init__(self) -> None:
        self.graph: dict[NodeT, list[NodeT]] = {}
        self.in_degree: dict[NodeT, int] = {}
```

```
    def add_edge(self, u: NodeT, v: NodeT) -> None:
        """Add directed edge from u to v."""
```

```
        if u not in self.graph:
            self.graph[u] = []
            self.in_degree[u] = 0
        if v not in self.in_degree:
            self.in_degree[v] = 0
            self.graph[v] = []
```

```
        self.graph[u].append(v)
        self.in_degree[v] += 1
```

```
    def kahn_sort(self) -> list[NodeT] | None:
```

```
        """
        Topological sort using Kahn's algorithm (BFS-based).
```

```
Returns the topological ordering, or None if the graph has a cycle.
        """
```

```
        in_deg = self.in_degree.copy()
        queue = deque([node for node, deg in in_deg.items() if deg == 0])
        result = []
```

```
        while queue:
            node = queue.popleft()
            result.append(node)
```

```
            for neighbor in self.graph[node]:
                in_deg[neighbor] -= 1
                if in_deg[neighbor] == 0:
                    queue.append(neighbor)
```

```
        # Check if all nodes are processed (no cycle)
        if len(result) != len(self.in_degree):
            return None
```

```
        return result
```

```
    def dfs_sort(self) -> list[NodeT] | None:
```

```
        """
        Topological sort using DFS.
```

```
Returns the topological ordering, or None if the graph has a cycle.
        """
```

```

WHITE, GRAY, BLACK = 0, 1, 2
color = dict.fromkeys(self.in_degree, WHITE)
result = []

def dfs(node: NodeT) -> bool:
    if color[node] == GRAY: # Back edge (cycle)
        return False
    if color[node] == BLACK: # Already processed
        return True

    color[node] = GRAY
    for neighbor in self.graph[node]:
        if not dfs(neighbor):
            return False

    color[node] = BLACK
    result.append(node)
    return True

for node in self.in_degree:
    if color[node] == WHITE and not dfs(node):
        return None

return result[::-1]

def has_cycle(self) -> bool:
    """Check if the graph contains a cycle."""
    return self.kahn_sort() is None

def longest_path(self) -> dict[NodeT, int]:
    """
    Find longest path from each node in the DAG.

    Returns a dictionary mapping each node to its longest path length.
    """
    topo_order = self.kahn_sort()
    if topo_order is None:
        msg = "Graph contains a cycle"
        raise ValueError(msg)

    dist = dict.fromkeys(self.in_degree, 0)

    for node in topo_order:
        for neighbor in self.graph[node]:
            dist[neighbor] = max(dist[neighbor], dist[node] + 1)

    return dist

def test_main() -> None:
    ts: TopologicalSort[int] = TopologicalSort()
    edges = [(5, 2), (5, 0), (4, 0), (4, 1), (2, 3), (3, 1)]
    for u, v in edges:
        ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()

# Test with cycle
    ts_cycle: TopologicalSort[int] = TopologicalSort()
    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.

```

def test_empty_graph() -> None:
    ts: TopologicalSort[int] = TopologicalSort()

```

```
# Empty graph should return empty list
```

```
assert ts.kahn_sort() == []
```

```
assert ts.dfs_sort() == []
```

```
assert not ts.has_cycle()
```

```
def test_single_node() -> None:
```

```
ts: TopologicalSort[str] = TopologicalSort()
```

```
ts.add_edge("A", "A") # This creates a self-loop (cycle)
```

```
assert ts.has_cycle()
```

```
assert ts.kahn_sort() is None
```

```
assert ts.dfs_sort() is None
```

```
# Single node without self-loop
```

```
ts2: TopologicalSort[str] = TopologicalSort()
```

```
ts2.in_degree["A"] = 0
```

```
ts2.graph["A"] = []
```

```
result = ts2.kahn_sort()
```

```
assert result == ["A"]
```

```
assert not ts2.has_cycle()
```

```
def test_simple_chain() -> None:
```

```
# Linear chain: A -> B -> C -> D
```

```
ts: TopologicalSort[str] = TopologicalSort()
```

```
edges = [("A", "B"), ("B", "C"), ("C", "D")]
```

```
for u, v in edges:
```

```
    ts.add_edge(u, v)
```

```
kahn_result = ts.kahn_sort()
```

```
dfs_result = ts.dfs_sort()
```

```
assert kahn_result == ["A", "B", "C", "D"]
```

```
assert dfs_result == ["A", "B", "C", "D"]
```

```
assert not ts.has_cycle()
```

```
def test_simple_cycle() -> None:
```

```
# Simple 3-cycle: A -> B -> C -> A
```

```
ts: TopologicalSort[str] = TopologicalSort()
```

```
edges = [("A", "B"), ("B", "C"), ("C", "A")]
```

```
for u, v in edges:
```

```
    ts.add_edge(u, v)
```

```
assert ts.has_cycle()
```

```
assert ts.kahn_sort() is None
```

```
assert ts.dfs_sort() is None
```

```
def test_disconnected_components() -> None:
```

```
# Two disconnected chains: A->B and C->D
```

```
ts: TopologicalSort[str] = TopologicalSort()
```

```
edges = [("A", "B"), ("C", "D")]
```

```
for u, v in edges:
```

```
    ts.add_edge(u, v)
```

```
kahn_result = ts.kahn_sort()
```

```
dfs_result = ts.dfs_sort()
```

```
# Both should work, order might vary for disconnected components
```

```
assert kahn_result is not None
```

```
assert dfs_result is not None
```

```
assert len(kahn_result) == 4
```

```
assert len(dfs_result) == 4
```

```
assert not ts.has_cycle()
```

```
# Check that ordering constraints are satisfied
```

```
assert kahn_result.index("A") < kahn_result.index("B")
```

```
assert kahn_result.index("C") < kahn_result.index("D")
```

```
def test_diamond_dag() -> None:
```

```

# Diamond: A -> B,C B,C -> D
ts: TopologicalSort[str] = TopologicalSort()
edges = [("A", "B"), ("A", "C"), ("B", "D"), ("C", "D")]
for u, v in edges:
    ts.add_edge(u, v)

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

assert kahn_result is not None
assert dfs_result is not None
assert not ts.has_cycle()

# Check ordering constraints
for result in [kahn_result, dfs_result]:
    assert result.index("A") < result.index("B")
    assert result.index("A") < result.index("C")
    assert result.index("B") < result.index("D")
    assert result.index("C") < result.index("D")

def test_complex_dag() -> None:
    # More complex DAG with multiple valid orderings
    ts: TopologicalSort[int] = TopologicalSort()
    edges = [(1, 2), (1, 3), (2, 4), (3, 4), (3, 5), (4, 6), (5, 6)]
    for u, v in edges:
        ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()

    # Verify all ordering constraints
    constraints = [(1, 2), (1, 3), (2, 4), (3, 4), (3, 5), (4, 6), (5, 6)]
    for result in [kahn_result, dfs_result]:
        for u, v in constraints:
            assert result.index(u) < result.index(v)

def test_large_cycle() -> None:
    # Large cycle: 0 -> 1 -> 2 -> ... -> 99 -> 0
    ts: TopologicalSort[int] = TopologicalSort()
    n = 100
    for i in range(n):
        ts.add_edge(i, (i + 1) % n)

    assert ts.has_cycle()
    assert ts.kahn_sort() is None
    assert ts.dfs_sort() is None

def test_tree_structure() -> None:
    # Binary tree structure (DAG)
    ts: TopologicalSort[int] = TopologicalSort()
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7)]
    for u, v in edges:
        ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()

    # Check tree constraints
    for result in [kahn_result, dfs_result]:
        assert result.index(1) < result.index(2)
        assert result.index(1) < result.index(3)
        assert result.index(2) < result.index(4)
        assert result.index(2) < result.index(5)
        assert result.index(3) < result.index(6)

```

```
assert result.index(3) < result.index(7)
```

```
def test_longest_path() -> None:
    # Test longest path functionality
    ts: TopologicalSort[str] = TopologicalSort()
    edges = [("A", "B"), ("A", "C"), ("B", "D"), ("C", "D"), ("D", "E")]
    for u, v in edges:
        ts.add_edge(u, v)

    longest_paths = ts.longest_path()

    # Expected longest paths from each node
    assert longest_paths["A"] == 0
    assert longest_paths["B"] == 1
    assert longest_paths["C"] == 1
    assert longest_paths["D"] == 2
    assert longest_paths["E"] == 3
```

```
def test_longest_path_with_cycle() -> None:
    # Should raise error for graphs with cycles
    ts: TopologicalSort[int] = TopologicalSort()
    edges = [(1, 2), (2, 3), (3, 1)] # Cycle
    for u, v in edges:
        ts.add_edge(u, v)

    try:
        ts.longest_path()
        assert False, "Should raise ValueError for cyclic graph"
    except ValueError:
        pass
```

```
def test_multiple_sources() -> None:
    # Graph with multiple sources (nodes with in-degree 0)
    ts: TopologicalSort[int] = TopologicalSort()
    edges = [(1, 3), (2, 3), (3, 4), (5, 6)]
    for u, v in edges:
        ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()

    # Sources (1, 2, 5) should come before their dependents
    for result in [kahn_result, dfs_result]:
        assert result.index(1) < result.index(3)
        assert result.index(2) < result.index(3)
        assert result.index(3) < result.index(4)
        assert result.index(5) < result.index(6)
```

```
def test_course_prerequisites() -> None:
    # Real-world example: course prerequisites
    ts: TopologicalSort[str] = TopologicalSort()
    # Math1 -> Math2 -> Math3, Physics1 -> Physics2, Math2 -> Physics2
    edges = [
        ("Math1", "Math2"), ("Math2", "Math3"),
        ("Physics1", "Physics2"), ("Math2", "Physics2")
    ]
    for u, v in edges:
        ts.add_edge(u, v)

    kahn_result = ts.kahn_sort()
    assert kahn_result is not None
    assert not ts.has_cycle()

    # Verify prerequisites
    assert kahn_result.index("Math1") < kahn_result.index("Math2")
    assert kahn_result.index("Math2") < kahn_result.index("Math3")
    assert kahn_result.index("Physics1") < kahn_result.index("Physics2")
```

```
assert kahn_result.index("Math2") < kahn_result.index("Physics2")
```

```
def test_complex_dependencies() -> None:
    # Complex dependency graph
    ts: TopologicalSort[str] = TopologicalSort()
    edges = [
        ("underwear", "pants"), ("underwear", "shoes"),
        ("pants", "belt"), ("pants", "shoes"),
        ("shirt", "belt"), ("shirt", "tie"),
        ("tie", "jacket"), ("belt", "jacket"),
        ("socks", "shoes")
    ]
    for u, v in edges:
        ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()

    # Check some key dependencies
    for result in [kahn_result, dfs_result]:
        assert result.index("underwear") < result.index("pants")
        assert result.index("pants") < result.index("shoes")
        assert result.index("socks") < result.index("shoes")
        assert result.index("shirt") < result.index("tie")
        assert result.index("tie") < result.index("jacket")
```

```
def test_stress_large_dag() -> None:
    # Large DAG: layered graph
    ts: TopologicalSort[int] = TopologicalSort()
    layers = 10
    nodes_per_layer = 10

    # Connect each node in layer i to all nodes in layer i+1
    for layer in range(layers - 1):
        for i in range(nodes_per_layer):
            for j in range(nodes_per_layer):
                u = layer * nodes_per_layer + i
                v = (layer + 1) * nodes_per_layer + j
                ts.add_edge(u, v)

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

    assert kahn_result is not None
    assert dfs_result is not None
    assert not ts.has_cycle()
    assert len(kahn_result) == layers * nodes_per_layer
```

```
def main() -> None:
    test_main()
    test_empty_graph()
    test_single_node()
    test_simple_chain()
    test_simple_cycle()
    test_disconnected_components()
    test_diamond_dag()
    test_complex_dag()
    test_large_cycle()
    test_tree_structure()
    test_longest_path()
    test_longest_path_with_cycle()
    test_multiple_sources()
    test_course_prerequisites()
    test_complex_dependencies()
    test_stress_large_dag()
```



```
if __name__ == "__main__":  
    main()
```

Two Sat

two_sat.py

"""

2-SAT solver using Kosaraju's SCC algorithm on implication graph.

2-SAT (Boolean Satisfiability with 2 literals per clause) determines if a Boolean formula in CNF with at most 2 literals per clause is satisfiable. Uses implication graph where each variable x has nodes x and $\neg x$, and clause $(a \text{ OR } b)$ creates edges $\neg a \rightarrow b$ and $\neg b \rightarrow a$.

The formula is satisfiable iff no variable x has x and $\neg x$ in the same SCC.

Time complexity: $O(n + m)$ where n is variables and m is clauses.

Space complexity: $O(n + m)$ for the implication graph.

"""

from __future__ import annotations

Don't use annotations during contest

from typing import Final

class TwoSAT:

def __init__(self, n: int) -> None:

"""Initialize 2-SAT solver for n Boolean variables (indexed 0 to $n-1$)."""

self.n: Final[int] = n

Implication graph: node $2*i$ is x_i , node $2*i+1$ is $\neg x_i$

self.graph: list[list[int]] = [[] for _ in range(2 * n)]

self.transpose: list[list[int]] = [[] for _ in range(2 * n)]

def add_clause(self, a: int, b: int, *, a_neg: bool, b_neg: bool) -> None:

"""

Add clause $(a \text{ OR } b)$ where a and b are variable indices.

$a_neg=True$ means $\neg a$, $a_neg=False$ means a .

Creates implications: $\neg a \rightarrow b$ and $\neg b \rightarrow a$.

"""

Map to graph nodes: $2*i = x_i$, $2*i+1 = \neg x_i$

a_node = 2 * a + (1 if a_neg else 0) # If a_neg, use $\neg a$ ($2*a+1$); else use a ($2*a$)

b_node = 2 * b + (1 if b_neg else 0) # If b_neg, use $\neg b$ ($2*b+1$); else use b ($2*b$)

na_node = 2 * a + (0 if a_neg else 1) # Negation of a_node

nb_node = 2 * b + (0 if b_neg else 1) # Negation of b_node

$\neg a \rightarrow b$ and $\neg b \rightarrow a$

self.graph[na_node].append(b_node)

self.graph[nb_node].append(a_node)

self.transpose[b_node].append(na_node)

self.transpose[a_node].append(nb_node)

def solve(self) -> list[bool] | None:

"""

Solve 2-SAT problem.

Returns assignment $[x_0, x_1, \dots, x_{n-1}]$ if satisfiable, None otherwise.

If variable x and $\neg x$ are in same SCC, formula is unsatisfiable.

"""

Kosaraju's algorithm for SCCs

visited: list[bool] = [False] * (2 * self.n)

finish_order: list[int] = []

def dfs1(node: int) -> None:

visited[node] = True

for neighbor in self.graph[node]:

if not visited[neighbor]:

dfs1(neighbor)

finish_order.append(node)

for node in range(2 * self.n):

if not visited[node]:

dfs1(node)

Second DFS pass on transpose

```

visited = [False] * (2 * self.n)
scc_id: list[int] = [-1] * (2 * self.n)
current_scc = 0

def dfs2(node: int, scc: int) -> None:
    visited[node] = True
    scc_id[node] = scc
    for neighbor in self.transpose[node]:
        if not visited[neighbor]:
            dfs2(neighbor, scc)

for node in reversed(finish_order):
    if not visited[node]:
        dfs2(node, current_scc)
        current_scc += 1

# Check satisfiability: x and ¬x must not be in same SCC
for i in range(self.n):
    if scc_id[2 * i] == scc_id[2 * i + 1]:
        return None

# Construct assignment: if SCC(x) > SCC(not-x), set x=True (reverse topo order)
return [scc_id[2 * i] > scc_id[2 * i + 1] for i in range(self.n)]

```

```

def test_main() -> None:
    # Test: (x0 ∨ x1) ∧ (¬x0 ∨ x1) ∧ (x0 ∨ ¬x1)
    # Simplifies to: x1 ∧ x0, so both must be True
    sat: TwoSAT = TwoSAT(2)
    sat.add_clause(0, 1, a_neg=False, b_neg=False) # x0 ∨ x1
    sat.add_clause(0, 1, a_neg=True, b_neg=False)  # ¬x0 ∨ x1
    sat.add_clause(0, 1, a_neg=False, b_neg=True)  # x0 ∨ ¬x1

    result = sat.solve()
    assert result is not None

    # Verify solution satisfies all clauses
    assert result[0] or result[1] # x0 ∨ x1
    assert (not result[0]) or result[1] # ¬x0 ∨ x1
    assert result[0] or (not result[1]) # x0 ∨ ¬x1

```

Don't write tests below during competition.

```

def test_unsatisfiable() -> None:
    # Test: (x ∨ y) ∧ (x ∨ ¬y) ∧ (¬x ∨ y) ∧ (¬x ∨ ¬y)
    # This is equivalent to x ∧ ¬x, which is unsatisfiable
    sat: TwoSAT = TwoSAT(2)
    sat.add_clause(0, 1, a_neg=False, b_neg=False) # x ∨ y
    sat.add_clause(0, 1, a_neg=False, b_neg=True)  # x ∨ ¬y
    sat.add_clause(0, 1, a_neg=True, b_neg=False)  # ¬x ∨ y
    sat.add_clause(0, 1, a_neg=True, b_neg=True)   # ¬x ∨ ¬y

    result = sat.solve()
    assert result is None

```

```

def test_single_variable() -> None:
    # Test: (x ∨ x) which is just x
    sat: TwoSAT = TwoSAT(1)
    sat.add_clause(0, 0, a_neg=False, b_neg=False) # x ∨ x

    result = sat.solve()
    assert result is not None
    assert result[0] is True

```

```

def test_trivial_satisfiable() -> None:
    # Test: (x ∨ ¬x) which is always true
    sat: TwoSAT = TwoSAT(1)
    sat.add_clause(0, 0, a_neg=False, b_neg=True) # x ∨ ¬x

    result = sat.solve()
    assert result is not None # Can be either True or False

```

```

def test_implication_chain() -> None:
    # Test:  $(\neg x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ 
    # This creates chain:  $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3$ 
    # Satisfiable with  $x_0=False$  or all True
    sat: TwoSAT = TwoSAT(4)
    sat.add_clause(0, 1, a_neg=True, b_neg=False) #  $\neg x_0 \vee x_1$  ( $x_0 \rightarrow x_1$ )
    sat.add_clause(1, 2, a_neg=True, b_neg=False) #  $\neg x_1 \vee x_2$  ( $x_1 \rightarrow x_2$ )
    sat.add_clause(2, 3, a_neg=True, b_neg=False) #  $\neg x_2 \vee x_3$  ( $x_2 \rightarrow x_3$ )

    result = sat.solve()
    assert result is not None
    # Verify implications
    if result[0]:
        assert result[1]
    if result[1]:
        assert result[2]
    if result[2]:
        assert result[3]

def test_mutual_implication() -> None:
    # Test:  $(\neg x \vee y) \wedge (\neg y \vee x)$ 
    # This means  $x \leftrightarrow y$  ( $x$  and  $y$  must have same value)
    sat: TwoSAT = TwoSAT(2)
    sat.add_clause(0, 1, a_neg=True, b_neg=False) #  $\neg x \vee y$  ( $x \rightarrow y$ )
    sat.add_clause(1, 0, a_neg=True, b_neg=False) #  $\neg y \vee x$  ( $y \rightarrow x$ )

    result = sat.solve()
    assert result is not None
    assert result[0] == result[1]

def test_large_satisfiable() -> None:
    # Test with 10 variables, random satisfiable clauses
    sat: TwoSAT = TwoSAT(10)

    # Add clauses that form a satisfiable system
    for i in range(9):
        sat.add_clause(i, i + 1, a_neg=False, b_neg=False) #  $x_i \vee x_{i+1}$ 

    result = sat.solve()
    assert result is not None
    # At least one variable in each pair should be True
    for i in range(9):
        assert result[i] or result[i + 1]

def test_contradictory_implications() -> None:
    # Test:  $x \rightarrow y$  and  $x \rightarrow \neg y$ , which means  $\neg x$  must be True
    sat: TwoSAT = TwoSAT(2)
    sat.add_clause(0, 1, a_neg=True, b_neg=False) #  $\neg x \vee y$  ( $x \rightarrow y$ )
    sat.add_clause(0, 1, a_neg=True, b_neg=True) #  $\neg x \vee \neg y$  ( $x \rightarrow \neg y$ )

    result = sat.solve()
    assert result is not None
    assert result[0] is False #  $x$  must be False

def test_complex_system() -> None:
    # 5 variables with multiple constraints
    sat: TwoSAT = TwoSAT(5)
    sat.add_clause(0, 1, a_neg=False, b_neg=False) #  $x_0 \vee x_1$ 
    sat.add_clause(1, 2, a_neg=True, b_neg=False) #  $\neg x_1 \vee x_2$ 
    sat.add_clause(2, 3, a_neg=True, b_neg=True) #  $\neg x_2 \vee \neg x_3$ 
    sat.add_clause(3, 4, a_neg=False, b_neg=False) #  $x_3 \vee x_4$ 
    sat.add_clause(4, 0, a_neg=True, b_neg=True) #  $\neg x_4 \vee \neg x_0$ 

    result = sat.solve()
    assert result is not None

    # Verify all clauses
    assert result[0] or result[1]
    assert (not result[1]) or result[2]

```

```
assert (not result[2]) or (not result[3])
assert result[3] or result[4]
assert (not result[4]) or (not result[0])
```

```
def test_xor_constraint() -> None:
    # Test XOR:  $x \oplus y$  (exactly one of  $x, y$  is True)
    #  $XOR = (x \vee y) \wedge (\neg x \vee \neg y)$ 
    sat: TwoSAT = TwoSAT(2)
    sat.add_clause(0, 1, a_neg=False, b_neg=False) #  $x \vee y$ 
    sat.add_clause(0, 1, a_neg=True, b_neg=True)  #  $\neg x \vee \neg y$ 

    result = sat.solve()
    assert result is not None
    # Exactly one should be True
    assert (result[0] and not result[1]) or (not result[0] and result[1])
```

```
def main() -> None:
    test_main()
    test_unsatisfiable()
    test_single_variable()
    test_trivial_satisfiable()
    test_implication_chain()
    test_mutual_implication()
    test_large_satisfiable()
    test_contradictory_implications()
    test_complex_system()
    test_xor_constraint()
```

```
if __name__ == "__main__":
    main()
```

Union Find

union_find.py

```
"""
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).
"""
```

```
from __future__ import annotations
```

```
# Don't use annotations during contest
from typing import TYPE_CHECKING, cast
```

```
if TYPE_CHECKING:
    from typing_extensions import Self
```

```
class UnionFind:
```

```
    def __init__(self) -> None:
        self.parent = self
        self.rank = 0
```

```
    def merge(self, other: Self) -> None:
        """Override to define custom merge behavior when sets are united."""
```

```
    def find(self) -> Self:
        """Return root of this set with path compression."""
        if self.parent == self:
            return self
        self.parent = self.parent.find()
        return cast("Self", self.parent)
```

```
    def union(self, other: Self) -> Self:
        """Unite sets containing self and other. Returns the new root."""
        x = self.find()
        y = other.find()
        if x is 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 += 1
        return y
```

```
class Test(UnionFind):
```

```
    """Better to modify copy of UnionFind class and avoid having to type cast everywhere."""
```

```
    def __init__(self) -> None:
        super().__init__()
        self.size = 1
```

```
    def merge(self, other: Self) -> None:
        assert isinstance(other, Test)
        self.size += other.size
```

```
def test_main() -> None:
    a, b, c = Test(), Test(), Test()
    d = a.union(b)
```

```
e = d.union(c)
assert e.find().size == 3
assert a.find().size == 3
```

Don't write tests below during competition.

```
def test_single_element() -> None:
```

```
    a = Test()
    assert a.find() is a
    assert a.size == 1
```

```
def test_union_same_set() -> None:
```

```
    a = Test()
    b = Test()
    a.union(b)
    # Unioning again should be safe
    root = a.union(b)
    assert a.find() is b.find()
    assert root.size == 2
```

```
def test_multiple_unions() -> None:
```

```
    nodes = [Test() for _ in range(10)]
    # Chain union: 0-1-2-3-4-5-6-7-8-9
    for i in range(9):
        nodes[i].union(nodes[i + 1])
```

```
    # All should have same root
    root = nodes[0].find()
    for node in nodes:
        assert node.find() is root
```

```
    assert root.size == 10
```

```
def test_union_order_independence() -> None:
```

```
    # Test that union order doesn't affect final result
    a1, b1, c1 = Test(), Test(), Test()
    a1.union(b1).union(c1)
    root1 = a1.find()
```

```
    a2, b2, c2 = Test(), Test(), Test()
    c2.union(b2).union(a2)
    root2 = a2.find()
```

```
    assert root1.size == root2.size == 3
```

```
def test_disconnected_sets() -> None:
```

```
    # Create two separate sets
```

```
    a, b = Test(), Test()
    c, d = Test(), Test()
```

```
    a.union(b)
    c.union(d)
```

```
    assert a.find() is b.find()
    assert c.find() is d.find()
    assert a.find() is not c.find()
```

```
    assert a.find().size == 2
    assert c.find().size == 2
```

```
def test_large_set() -> None:
```

```
    # Create a large union-find structure
    nodes = [Test() for _ in range(100)]
```

```
    # Union in pairs
```

```
    for i in range(0, 100, 2):
        nodes[i].union(nodes[i + 1])
```

```
# Now we have 50 sets of size 2
roots = set()
for node in nodes:
    roots.add(id(node.find()))
assert len(roots) == 50

# Union all pairs together
for i in range(0, 100, 4):
    if i + 2 < 100:
        nodes[i].union(nodes[i + 2])

# Now we have 25 sets of size 4
roots = set()
for node in nodes:
    roots.add(id(node.find()))
assert len(roots) == 25
```

```
def main() -> None:
    test_single_element()
    test_union_same_set()
    test_multiple_unions()
    test_union_order_independence()
    test_disconnected_sets()
    test_large_set()
    test_main()
```

```
if __name__ == "__main__":
    main()
```