Algorithm Reference - Python

Bellman Ford

bellman_ford.py

```python
"""
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
```

```python
                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"
```

Bipartite Match

bipartite_match.py

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

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

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

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"}
```

Convex Hull

convex_hull.py

```python
"""
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
```

```python
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
```

Dijkstra

dijkstra.py

```python
"""
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
```

```python
            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"]
```

Edmonds Karp

edmonds_karp.py

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

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

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

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,
```

```python
        )


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
```

Fenwick Tree

fenwick_tree.py

```python
"""
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:
```

```python
            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
```

```python
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
```

Kmp

kmp.py

```python
"""
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
```

```python
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("abcabcab")
    assert failure == [0, 0, 0, 1, 2, 3, 4, 5]
```

Kosaraju Scc

kosaraju_scc.py

```python
"""
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:
```

```python
            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]]
```

Lca

lca.py

```python
"""
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)
```

```python
    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
```

Polygon Area

polygon_area.py

```python
"""
Shoelace formula (Gauss's area formula) for computing the area of a polygon.

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

The formula: Area = 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
```

```python
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)
```

Prefix Tree

prefix_tree.py

```python
"""
Write-only prefix tree (trie) for efficient string storage and retrieval.

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

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

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
```

Priority Queue

priority_queue.py

```python
"""
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:
```

```python
                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)
```

Segment Tree

segment_tree.py

```python
"""
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}"
```

```python
            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
```

Skiplist

skiplist.py

```python
"""
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
```

```python
            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)
```

```python
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
```

Sprague Grundy

sprague_grundy.py

```python
"""
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
```

```python
    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
```

```python
            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  # 17^23^31 = 25
    assert eng.is_winning_position([15, 27, 36]) is True  # 15^27^36 = 48 != 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 → grundy = 0
    assert eng2.grundy(15) == 1  # 15 % 7 = 1 → grundy = 1
    assert eng2.grundy(18) == 2  # 18 % 7 = 4 → grundy = 2

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

Suffix Array

suffix_array.py

```python
"""
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.
        """
```

```python
        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]
```

Topological Sort

topological_sort.py

```python
"""
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.
```

```python
        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()
```

Two Sat

two_sat.py

```python
"""
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 not-x, and clause (a OR b) creates edges not-a -> b
and not-b -> a.

The formula is satisfiable iff no variable x has x and not-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 ¬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 OR b) where a and b are variable indices.
        a_neg=True means not-a, a_neg=False means a.
        Creates implications: not-a -> b and not-b -> a.
        """
        # Map to graph nodes: 2*i = xi, 2*i+1 = ¬xi
        a_node = 2 * a + (1 if a_neg else 0)  # If a_neg, use ¬a (2*a+1); else use a (2*a)
        b_node = 2 * b + (1 if b_neg else 0)  # If b_neg, use ¬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

        # ¬a → b and ¬b → 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, ..., x_{n-1}] if satisfiable, None otherwise.
        If variable x and ¬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]:
```

```python
                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
```

Union Find

union_find.py

```python
"""
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
```

```python
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
```