

# Algorithm Reference - Java

Bipartite Match

Dijkstra

Edmonds Karp

Fenwick Tree

Kmp

Lca

Polygon Area

Prefix Tree

Priority Queue

Segment Tree

Sprague Grundy

Topological Sort

Union Find

# Bipartite Match

---

bipartite\_match.java

```
/*
Maximum bipartite matching using augmenting path algorithm.

Given a bipartite graph with left and right vertex sets, finds the maximum
number of edges such that no two edges share a vertex.

Key operations:
- addEdge(u, v): Add edge from left vertex u to right vertex v
- maxMatching(): Compute maximum matching size

Time complexity:  $O(V * E)$ 
Space complexity:  $O(V + E)$ 
*/

import java.util.*;

class bipartite_match {
    static class BipartiteMatch {
        private int leftSize;
        private int rightSize;
        private Map<Integer, List<Integer>> graph;
        private Map<Integer, Integer> match;
        private Set<Integer> visited;

        BipartiteMatch(int leftSize, int rightSize) {
            this.leftSize = leftSize;
            this.rightSize = rightSize;
            this.graph = new HashMap<>();
            for (int i = 0; i < leftSize; i++) {
                graph.put(i, new ArrayList<>());
            }
        }

        void addEdge(int u, int v) {
            graph.get(u).add(v);
        }

        int maxMatching() {
            match = new HashMap<>();
            int matchingSize = 0;

            for (int u = 0; u < leftSize; u++) {
                visited = new HashSet<>();
                if (dfs(u)) {
                    matchingSize++;
                }
            }

            return matchingSize;
        }

        private boolean dfs(int u) {
            for (int v : graph.get(u)) {
                if (visited.contains(v)) {
                    continue;
                }
                visited.add(v);

                // If v is not matched or we can find augmenting path from match[v]
                if (!match.containsKey(v) || dfs(match.get(v))) {
                    match.put(v, u);
                    return true;
                }
            }
            return false;
        }
    }
}
```

```

    Map<Integer, Integer> getMatching() {
        Map<Integer, Integer> result = new HashMap<>();
        for (Map.Entry<Integer, Integer> entry : match.entrySet()) {
            result.put(entry.getValue(), entry.getKey());
        }
        return result;
    }
}

static void testMain() {
    BipartiteMatch b = new BipartiteMatch(3, 3);
    b.addEdge(0, 0); // 1 -> X
    b.addEdge(1, 1); // 2 -> Y
    b.addEdge(2, 0); // 3 -> X
    b.addEdge(0, 2); // 1 -> Z
    b.addEdge(1, 2); // 2 -> Z
    b.addEdge(2, 1); // 3 -> Y

    int matching = b.maxMatching();
    if (matching != 3) throw new AssertionError("Expected 3, got " + matching);
    Map<Integer, Integer> matches = b.getMatching();
    if (matches.size() != 3) throw new AssertionError("Expected size 3, got " + matches.size());
}

```

# Dijkstra

---

dijkstra.java

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

Finds shortest paths from a source vertex to all other vertices in a graph
with non-negative edge weights.

Key operations:
- addEdge(u, v, weight): Add weighted directed edge
- shortestPaths(source): Compute shortest paths from source to all vertices
- shortestPath(source, target): Get shortest path between two vertices

Time complexity:  $O((V + E) \log V)$  with binary heap
Space complexity:  $O(V + E)$ 
*/

import java.util.*;

class dijkstra {
    static class Edge {
        int to;
        int weight;

        Edge(int to, int weight) {
            this.to = to;
            this.weight = weight;
        }
    }

    static class Node implements Comparable<Node> {
        int vertex;
        int distance;

        Node(int vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.distance, other.distance);
        }
    }

    static class Dijkstra {
        private int n;
        private Map<Integer, List<Edge>> graph;

        Dijkstra(int n) {
            this.n = n;
            this.graph = new HashMap<>();
            for (int i = 0; i < n; i++) {
                graph.put(i, new ArrayList<>());
            }
        }

        void addEdge(int u, int v, int weight) {
            graph.get(u).add(new Edge(v, weight));
        }

        Map<Integer, Integer> shortestPaths(int source) {
            Map<Integer, Integer> distances = new HashMap<>();
            for (int i = 0; i < n; i++) {
                distances.put(i, Integer.MAX_VALUE);
            }
            distances.put(source, 0);
        }
    }
}
```

```

PriorityQueue<Node> pq = new PriorityQueue<>();
pq.offer(new Node(source, 0));

while (!pq.isEmpty()) {
    Node current = pq.poll();
    int u = current.vertex;
    int dist = current.distance;

    if (dist > distances.get(u)) {
        continue;
    }

    for (Edge edge : graph.get(u)) {
        int v = edge.to;
        int newDist = dist + edge.weight;

        if (newDist < distances.get(v)) {
            distances.put(v, newDist);
            pq.offer(new Node(v, newDist));
        }
    }
}

return distances;
}

List<Integer> shortestPath(int source, int target) {
    Map<Integer, Integer> distances = new HashMap<>();
    Map<Integer, Integer> previous = new HashMap<>();

    for (int i = 0; i < n; i++) {
        distances.put(i, Integer.MAX_VALUE);
    }
    distances.put(source, 0);

    PriorityQueue<Node> pq = new PriorityQueue<>();
    pq.offer(new Node(source, 0));

    while (!pq.isEmpty()) {
        Node current = pq.poll();
        int u = current.vertex;
        int dist = current.distance;

        if (u == target) {
            break;
        }

        if (dist > distances.get(u)) {
            continue;
        }

        for (Edge edge : graph.get(u)) {
            int v = edge.to;
            int newDist = dist + edge.weight;

            if (newDist < distances.get(v)) {
                distances.put(v, newDist);
                previous.put(v, u);
                pq.offer(new Node(v, newDist));
            }
        }
    }

    if (!previous.containsKey(target) && target != source) {
        return null;
    }

    List<Integer> path = new ArrayList<>();
    int current = target;
    while (current != source) {
        path.add(current);
        current = previous.get(current);
    }
}

```

```

    }
    path.add(source);
    Collections.reverse(path);

    return path;
}

static void testMain() {
    Dijkstra d = new Dijkstra(4);
    d.addEdge(0, 1, 4);
    d.addEdge(0, 2, 2);
    d.addEdge(1, 2, 1);
    d.addEdge(1, 3, 5);
    d.addEdge(2, 3, 8);

    Map<Integer, Integer> distances = d.shortestPaths(0);
    assert distances.get(3) == 9;

    List<Integer> path = d.shortestPath(0, 3);
    assert path.equals(Arrays.asList(0, 1, 3));
}

```

# Edmonds Karp

---

edmonds\_karp.java

```
/*
Edmonds-Karp algorithm for computing maximum flow in a flow network.

Implementation of the Ford-Fulkerson method using BFS to find augmenting paths.
Guarantees  $O(V * E^2)$  time complexity.

Key operations:
- addEdge(u, v, capacity): Add a directed edge with given capacity
- maxFlow(source, sink): Compute maximum flow from source to sink

Space complexity:  $O(V^2)$  for adjacency matrix representation
*/

import java.util.*;

class edmonds_karp {
    static class EdmondsKarp {
        private int n;
        private int[][] capacity;
        private int[][] flow;

        EdmondsKarp(int n) {
            this.n = n;
            this.capacity = new int[n][n];
            this.flow = new int[n][n];
        }

        void addEdge(int u, int v, int cap) {
            capacity[u][v] += cap;
        }

        int maxFlow(int source, int sink) {
            // Reset flow
            for (int i = 0; i < n; i++) {
                Arrays.fill(flow[i], 0);
            }

            int totalFlow = 0;

            while (true) {
                // BFS to find augmenting path
                int[] parent = new int[n];
                Arrays.fill(parent, -1);
                parent[source] = source;

                Queue<Integer> queue = new LinkedList<>();
                queue.offer(source);

                while (!queue.isEmpty() && parent[sink] == -1) {
                    int u = queue.poll();

                    for (int v = 0; v < n; v++) {
                        if (parent[v] == -1 && capacity[u][v] - flow[u][v] > 0) {
                            parent[v] = u;
                            queue.offer(v);
                        }
                    }
                }

                // No augmenting path found
                if (parent[sink] == -1) {
                    break;
                }

                // Find minimum residual capacity along the path
                int pathFlow = Integer.MAX_VALUE;
            }
        }
    }
}
```

```

        int v = sink;
        while (v != source) {
            int u = parent[v];
            pathFlow = Math.min(pathFlow, capacity[u][v] - flow[u][v]);
            v = u;
        }

        // Update flow along the path
        v = sink;
        while (v != source) {
            int u = parent[v];
            flow[u][v] += pathFlow;
            flow[v][u] -= pathFlow;
            v = u;
        }

        totalFlow += pathFlow;
    }

    return totalFlow;
}

int getFlow(int u, int v) {
    return flow[u][v];
}

}

static void testMain() {
    EdmondsKarp e = new EdmondsKarp(4);
    e.addEdge(0, 1, 10);
    e.addEdge(0, 2, 8);
    e.addEdge(1, 2, 2);
    e.addEdge(1, 3, 5);
    e.addEdge(2, 3, 7);

    int maxFlow = e.maxFlow(0, 3);
    assert maxFlow == 12;
}

```



# Fenwick Tree

---

fenwick\_tree.java

/\*

*Fenwick Tree (Binary Indexed Tree) implementation.*

*A data structure for efficient prefix sum queries and point updates on an array.*

*Key operations:*

- *update(i, delta): Add delta to element at index i -  $O(\log n)$*
- *query(i): Get sum of elements from index 0 to i (inclusive) -  $O(\log n)$*
- *range\_query(l, r): Get sum from index l to r (inclusive) -  $O(\log n)$*

*Space complexity:  $O(n)$*

*Note: Uses 1-based indexing internally for simpler bit manipulation.*

\*/

```
import java.util.function.BinaryOperator;
```

```
class fenwick_tree {
    interface Summable<T> {
        T add(T other);

        T subtract(T other);
    }

    static class FenwickTree<T> {
        private Object[] tree;
        private int size;
        private T zero;
        private BinaryOperator<T> addOp;
        private BinaryOperator<T> subtractOp;

        FenwickTree(int n, T zero, BinaryOperator<T> addOp, BinaryOperator<T> subtractOp) {
            this.size = n;
            this.zero = zero;
            this.addOp = addOp;
            this.subtractOp = subtractOp;
            this.tree = new Object[n + 1];
            for (int i = 0; i <= n; i++) {
                tree[i] = zero;
            }
        }

        //  $O(n \log n)$  constructor from array
        FenwickTree(T[] values, T zero, BinaryOperator<T> addOp, BinaryOperator<T> subtractOp) {
            this(values.length, zero, addOp, subtractOp);
            for (int i = 0; i < values.length; i++) {
                update(i, values[i]);
            }
        }

        //  $O(n)$  constructor from array using prefix sums
        static <T> FenwickTree<T> fromArray(
            T[] arr, T zero, BinaryOperator<T> addOp, BinaryOperator<T> subtractOp) {
            int n = arr.length;
            FenwickTree<T> ft = new FenwickTree<>(n, zero, addOp, subtractOp);

            // Compute prefix sums
            Object[] prefix = new Object[n + 1];
            prefix[0] = zero;
            for (int i = 0; i < n; i++) {
                prefix[i + 1] = addOp.apply((T) prefix[i], arr[i]);
            }

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

```

```

        ft.tree[i] = subtractOp.apply((T) prefix[i], (T) prefix[rangeStart - 1]);
    }

    return ft;
}

@SuppressWarnings("unchecked")
void update(int i, T delta) {
    if (i < 0 || i >= size) {
        throw new IndexOutOfBoundsException(
            "Index " + i + " out of bounds for size " + size);
    }
    i++; // Convert to 1-based indexing
    while (i <= size) {
        tree[i] = addOp.apply((T) tree[i], delta);
        i += i & (-i);
    }
}

@SuppressWarnings("unchecked")
T query(int i) {
    if (i < 0 || i >= size) {
        throw new IndexOutOfBoundsException(
            "Index " + i + " out of bounds for size " + size);
    }
    i++; // Convert to 1-based indexing
    T sum = zero;
    while (i > 0) {
        sum = addOp.apply(sum, (T) tree[i]);
        i -= i & (-i);
    }
    return sum;
}

T rangeQuery(int l, int r) {
    if (l > r || l < 0 || r >= size) {
        return zero;
    }
    if (l == 0) {
        return query(r);
    }
    return subtractOp.apply(query(r), query(l - 1));
}

// Optional functionality (not always needed during competition)

T getValue(int i) {
    if (i < 0 || i >= size) {
        throw new IndexOutOfBoundsException(
            "Index " + i + " out of bounds for size " + size);
    }
    if (i == 0) {
        return query(0);
    }
    return subtractOp.apply(query(i), query(i - 1));
}

// Find smallest index >= startIndex with value > zero
// REQUIRES: all updates are non-negative, T must be comparable
@SuppressWarnings("unchecked")
Integer firstNonzeroIndex(int startIndex, java.util.Comparator<T> comparator) {
    startIndex = Math.max(startIndex, 0);
    if (startIndex >= size) {
        return null;
    }

    T prefixBefore = startIndex > 0 ? query(startIndex - 1) : zero;
    T total = query(size - 1);
    if (comparator.compare(total, prefixBefore) == 0) {
        return null;
    }
}

```

```

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

while (bit > 0) {
    int nxt = idx + bit;
    if (nxt <= size) {
        T cand = addOp.apply(cur, (T) tree[nxt]);
        if (comparator.compare(cand, prefixBefore)
            <= 0) { // move right while prefix <= target
            cur = cand;
            idx = nxt;
        }
    }
    bit >>= 1;
}

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

}

static void testMain() {
    FenwickTree<Long> f = new FenwickTree<>(5, 0L, (a, b) -> a + b, (a, b) -> a - b);
    f.update(0, 7L);
    f.update(2, 13L);
    f.update(4, 19L);
    assert f.query(4) == 39L;
    assert f.rangeQuery(1, 3) == 13L;

    // Optional functionality (not always needed during competition)

    assert f.getValue(2) == 13L;
    FenwickTree<Long> g =
        FenwickTree.fromArray(
            new Long[] {1L, 2L, 3L, 4L, 5L}, 0L, (a, b) -> a + b, (a, b) -> a - b);
    assert g.query(4) == 15L;
}

```

# Kmp

---

kmp.java

/\*

*Knuth-Morris-Pratt (KMP) string matching algorithm.*

*Efficiently finds all occurrences of a pattern in a text string.*

*Key operations:*

- *computeLPS(pattern): Compute Longest Proper Prefix which is also Suffix array*

- *search(text, pattern): Find all starting positions where pattern occurs in text*

*Time complexity:  $O(n + m)$  where  $n$  is text length and  $m$  is pattern length*

*Space complexity:  $O(m)$  for the LPS array*

\*/

```
import java.util.*;
```

```
class kmp {
```

```
    static int[] computeLPS(String pattern) {
```

```
        int m = pattern.length();
```

```
        int[] lps = new int[m];
```

```
        int len = 0;
```

```
        int i = 1;
```

```
        lps[0] = 0;
```

```
        while (i < m) {
```

```
            if (pattern.charAt(i) == pattern.charAt(len)) {
```

```
                len++;
```

```
                lps[i] = len;
```

```
                i++;
```

```
            } else {
```

```
                if (len != 0) {
```

```
                    len = lps[len - 1];
```

```
                } else {
```

```
                    lps[i] = 0;
```

```
                    i++;
```

```
                }
```

```
            }
```

```
        }
```

```
        return lps;
```

```
    }
```

```
    static List<Integer> search(String text, String pattern) {
```

```
        List<Integer> result = new ArrayList<>();
```

```
        if (pattern.isEmpty()) {
```

```
            return result;
```

```
        }
```

```
        int n = text.length();
```

```
        int m = pattern.length();
```

```
        int[] lps = computeLPS(pattern);
```

```
        int i = 0; // index for text
```

```
        int j = 0; // index for pattern
```

```
        while (i < n) {
```

```
            if (text.charAt(i) == pattern.charAt(j)) {
```

```
                i++;
```

```
                j++;
```

```
            }
```

```
            if (j == m) {
```

```
                result.add(i - j);
```

```
                j = lps[j - 1];
```

```
            } else if (i < n && text.charAt(i) != pattern.charAt(j)) {
```

```

        if (j != 0) {
            j = lps[j - 1];
        } else {
            i++;
        }
    }

    return result;
}

static void testMain() {
    String text = "ababcbababab";
    String pattern = "aba";
    List<Integer> matches = search(text, pattern);
    assert matches.equals(Arrays.asList(0, 5, 7));
    assert matches.size() == 3;

    // Test failure function
    int[] failure = computeLPS("abcbabcbab");
    assert Arrays.equals(failure, new int[] {0, 0, 0, 1, 2, 3, 4, 5});
}

```

# Lca

---

lca.java

```
/*  
Lowest Common Ancestor (LCA) using Binary Lifting.  
  
Preprocesses a tree to answer LCA queries efficiently.  
  
Key operations:  
- addEdge(u, v): Add undirected edge to tree  
- build(root): Preprocess tree with given root -  $O(n \log n)$   
- query(u, v): Find LCA of nodes u and v -  $O(\log n)$   
- distance(u, v): Find distance between two nodes -  $O(\log n)$ 
```

Space complexity:  $O(n \log n)$

Binary lifting allows us to "jump" up the tree in powers of 2, enabling efficient LCA queries.

```
*/  
  
import java.util.*;  
  
class lca {  
    static class LCA {  
        private int n;  
        private int maxLog;  
        private Map<Integer, List<Integer>> graph;  
        private int[] depth;  
        private Map<Integer, Map<Integer, Integer>> up;  
  
        LCA(int n) {  
            this.n = n;  
            this.maxLog = (int) Math.ceil(Math.log(n) / Math.log(2)) + 1;  
            this.graph = new HashMap<>();  
            this.depth = new int[n];  
            this.up = new HashMap<>();  
  
            for (int i = 0; i < n; i++) {  
                graph.put(i, new ArrayList<>());  
                up.put(i, new HashMap<>());  
            }  
        }  
  
        void addEdge(int u, int v) {  
            graph.get(u).add(v);  
            graph.get(v).add(u);  
        }  
  
        void build(int root) {  
            Arrays.fill(depth, 0);  
            dfs(root, -1, 0);  
        }  
  
        private void dfs(int node, int parent, int d) {  
            depth[node] = d;  
  
            if (parent != -1) {  
                up.get(node).put(0, parent);  
            }  
  
            for (int i = 1; i < maxLog; i++) {  
                if (up.get(node).containsKey(i - 1)) {  
                    int ancestor = up.get(node).get(i - 1);  
                    if (up.get(ancestor).containsKey(i - 1)) {  
                        up.get(node).put(i, up.get(ancestor).get(i - 1));  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    for (int child : graph.get(node)) {
        if (child != parent) {
            dfs(child, node, d + 1);
        }
    }
}

int query(int u, int v) {
    if (depth[u] < depth[v]) {
        int temp = u;
        u = v;
        v = temp;
    }

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

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

    // Binary search for LCA
    for (int i = maxLog - 1; i >= 0; i--) {
        if (up.get(u).containsKey(i) && up.get(v).containsKey(i)) {
            int uAncestor = up.get(u).get(i);
            int vAncestor = up.get(v).get(i);
            if (uAncestor != vAncestor) {
                u = uAncestor;
                v = vAncestor;
            }
        }
    }

    return up.get(u).getOrDefault(0, u);
}

int distance(int u, int v) {
    int lcaNode = query(u, v);
    return depth[u] + depth[v] - 2 * depth[lcaNode];
}

static void testMain() {
    LCA lca = new LCA(6);
    lca.addEdge(0, 1); // 1-2
    lca.addEdge(0, 2); // 1-3
    lca.addEdge(1, 3); // 2-4
    lca.addEdge(1, 4); // 2-5
    lca.addEdge(2, 5); // 3-6

    lca.build(0);

    assert lca.query(3, 4) == 1; // LCA(4, 5) = 2
    assert lca.query(3, 5) == 0; // LCA(4, 6) = 1
    assert lca.distance(3, 5) == 4; // distance(4, 6) = 4
}

```

# Polygon Area

---

polygon\_area.java

```
/*  
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.  
*/
```

```
class polygon_area {  
    static class Point {  
        double x, y;  
  
        Point(double x, double y) {  
            this.x = x;  
            this.y = y;  
        }  
    }  
  
    static double polygonArea(Point[] vertices) {  
        if (vertices.length < 3) {  
            return 0.0;  
        }  
  
        int n = vertices.length;  
        double area = 0.0;  
  
        for (int i = 0; i < n; i++) {  
            int j = (i + 1) % n;  
            area += vertices[i].x * vertices[j].y;  
            area -= vertices[j].x * vertices[i].y;  
        }  
  
        return Math.abs(area) / 2.0;  
    }  
  
    static double polygonSignedArea(Point[] vertices) {  
        if (vertices.length < 3) {  
            return 0.0;  
        }  
  
        int n = vertices.length;  
        double area = 0.0;  
  
        for (int i = 0; i < n; i++) {  
            int j = (i + 1) % n;  
            area += vertices[i].x * vertices[j].y;  
            area -= vertices[j].x * vertices[i].y;  
        }  
  
        return area / 2.0;  
    }  
  
    static boolean isClockwise(Point[] vertices) {  
        return polygonSignedArea(vertices) < 0;  
    }  
  
    static void testMain() {  
        // Simple square with side length 2  
        Point[] square = {  
            new Point(0.0, 0.0), new Point(2.0, 0.0), new Point(2.0, 2.0), new Point(0.0, 2.0)  
        };  
        assert Math.abs(polygonArea(square) - 4.0) < 1e-9;  
    }  
}
```



```
// Triangle with base 3 and height 4
Point[] triangle = {new Point(0.0, 0.0), new Point(3.0, 0.0), new Point(1.5, 4.0)};
assert Math.abs(polygonArea(triangle) - 6.0) < 1e-9;

// Test orientation
Point[] ccwSquare = {
    new Point(0.0, 0.0), new Point(1.0, 0.0), new Point(1.0, 1.0), new Point(0.0, 1.0)
};
assert !isClockwise(ccwSquare);
}
```

# Prefix Tree

---

prefix\_tree.java

```
/*
Prefix Tree (Trie) implementation for efficient string prefix operations.

Supports:
- insert(word): Add a word to the trie - O(m) where m is word length
- search(word): Check if exact word exists - O(m)
- startsWith(prefix): Check if any word starts with prefix - O(m)
- delete(word): Remove a word from the trie - O(m)

Space complexity: O(ALPHABET_SIZE * N * M) where N is number of words and M is average length
*/

import java.util.*;

class prefix_tree {
    static class TrieNode {
        Map<Character, TrieNode> children;
        boolean isEndOfWord;

        TrieNode() {
            children = new HashMap<>();
            isEndOfWord = false;
        }
    }

    static class PrefixTree {
        private TrieNode root;

        PrefixTree() {
            root = new TrieNode();
        }

        void insert(String word) {
            TrieNode node = root;
            for (char c : word.toCharArray()) {
                node.children.putIfAbsent(c, new TrieNode());
                node = node.children.get(c);
            }
            node.isEndOfWord = true;
        }

        boolean search(String word) {
            TrieNode node = root;
            for (char c : word.toCharArray()) {
                if (!node.children.containsKey(c)) {
                    return false;
                }
                node = node.children.get(c);
            }
            return node.isEndOfWord;
        }

        boolean startsWith(String prefix) {
            TrieNode node = root;
            for (char c : prefix.toCharArray()) {
                if (!node.children.containsKey(c)) {
                    return false;
                }
                node = node.children.get(c);
            }
            return true;
        }

        boolean delete(String word) {
            return deleteHelper(root, word, 0);
        }
    }
}
```

```

private boolean deleteHelper(TrieNode node, String word, int depth) {
    if (node == null) {
        return false;
    }

    if (depth == word.length()) {
        if (!node.isEndOfWord) {
            return false;
        }
        node.isEndOfWord = false;
        return node.children.isEmpty();
    }

    char c = word.charAt(depth);
    if (!node.children.containsKey(c)) {
        return false;
    }

    TrieNode child = node.children.get(c);
    boolean shouldDeleteChild = deleteHelper(child, word, depth + 1);

    if (shouldDeleteChild) {
        node.children.remove(c);
        return !node.isEndOfWord && node.children.isEmpty();
    }

    return false;
}

static void testMain() {
    PrefixTree trie = new PrefixTree();
    trie.insert("cat");
    trie.insert("car");
    trie.insert("card");

    assert trie.search("car");
    assert !trie.search("ca");
    assert trie.startsWith("car");
}

```

# Priority Queue

---

priority\_queue.java

```
/*  
Generic priority queue (min-heap) with update and remove operations.
```

```
Supports:
```

- *push(item): Add item to heap -  $O(\log n)$*
- *pop(): Remove and return minimum item -  $O(\log n)$*
- *peek(): View minimum item without removing -  $O(1)$*
- *update(old\_item, new\_item): Update item in heap -  $O(n)$*
- *remove(item): Remove specific item -  $O(n)$*

```
Space complexity:  $O(n)$   
*/
```

```
import java.util.*;
```

```
class priority_queue {  
    static class PriorityQueue<T extends Comparable<T>> {  
        private List<T> heap;  
  
        PriorityQueue() {  
            this.heap = new ArrayList<>();  
        }  
  
        void push(T item) {  
            heap.add(item);  
            siftUp(heap.size() - 1);  
        }  
  
        T pop() {  
            if (heap.isEmpty()) {  
                throw new IllegalStateException("Heap is empty");  
            }  
            T item = heap.get(0);  
            T last = heap.remove(heap.size() - 1);  
            if (!heap.isEmpty()) {  
                heap.set(0, last);  
                siftDown(0);  
            }  
            return item;  
        }  
  
        T peek() {  
            if (heap.isEmpty()) {  
                return null;  
            }  
            return heap.get(0);  
        }  
  
        boolean contains(T item) {  
            return heap.contains(item);  
        }  
  
        void update(T oldItem, T newItem) {  
            int idx = heap.indexOf(oldItem);  
            if (idx == -1) {  
                throw new IllegalArgumentException("Item not in heap");  
            }  
            heap.set(idx, newItem);  
            if (newItem.compareTo(oldItem) < 0) {  
                siftUp(idx);  
            } else {  
                siftDown(idx);  
            }  
        }  
  
        void remove(T item) {
```

```

        int idx = heap.indexOf(item);
        if (idx == -1) {
            throw new IllegalArgumentException("Item not in heap");
        }
        T last = heap.remove(heap.size() - 1);
        if (idx < heap.size()) {
            T oldItem = heap.get(idx);
            heap.set(idx, last);
            if (last.compareTo(oldItem) < 0) {
                siftUp(idx);
            } else {
                siftDown(idx);
            }
        }
    }

    int size() {
        return heap.size();
    }

    boolean isEmpty() {
        return heap.isEmpty();
    }

    private void siftUp(int idx) {
        while (idx > 0) {
            int parent = (idx - 1) / 2;
            if (heap.get(idx).compareTo(heap.get(parent)) >= 0) {
                break;
            }
            Collections.swap(heap, idx, parent);
            idx = parent;
        }
    }

    private void siftDown(int idx) {
        while (true) {
            int smallest = idx;
            int left = 2 * idx + 1;
            int right = 2 * idx + 2;

            if (left < heap.size() && heap.get(left).compareTo(heap.get(smallest)) < 0) {
                smallest = left;
            }
            if (right < heap.size() && heap.get(right).compareTo(heap.get(smallest)) < 0) {
                smallest = right;
            }
            if (smallest == idx) {
                break;
            }
            Collections.swap(heap, idx, smallest);
            idx = smallest;
        }
    }

    static void testMain() {
        PriorityQueue<Integer> p = new PriorityQueue<>();
        p.push(15);
        p.push(23);
        p.push(8);
        assert p.peek() == 8;
        assert p.pop() == 8;
        assert p.pop() == 15;
    }
}

```

# Segment Tree

---

segment\_tree.java

```
/*
Segment Tree for range queries and point updates.

Supports efficient range queries (sum, min, max, etc.) and point updates on an array.

Key operations:
- update(i, value): Update element at index i -  $O(\log n)$ 
- query(l, r): Query range [l, r] -  $O(\log n)$ 

Space complexity:  $O(4n) = O(n)$ 

This implementation supports sum queries but can be modified for min/max/gcd/etc.
*/

import java.util.*;
import java.util.function.BinaryOperator;

class segment_tree {
    static class SegmentTree<T> {
        private Object[] tree;
        private int n;
        private T zero;
        private BinaryOperator<T> combineOp;

        SegmentTree(T[] arr, T zero, BinaryOperator<T> combineOp) {
            this.n = arr.length;
            this.zero = zero;
            this.combineOp = combineOp;
            this.tree = new Object[4 * n];
            if (n > 0) {
                build(arr, 0, 0, n - 1);
            }
        }

        private void build(T[] arr, int node, int start, int end) {
            if (start == end) {
                tree[node] = arr[start];
            } else {
                int mid = (start + end) / 2;
                int leftChild = 2 * node + 1;
                int rightChild = 2 * node + 2;

                build(arr, leftChild, start, mid);
                build(arr, rightChild, mid + 1, end);

                tree[node] = combineOp.apply((T) tree[leftChild], (T) tree[rightChild]);
            }
        }

        void update(int idx, T value) {
            update(0, 0, n - 1, idx, value);
        }

        @SuppressWarnings("unchecked")
        private void update(int node, int start, int end, int idx, T value) {
            if (start == end) {
                tree[node] = value;
            } else {
                int mid = (start + end) / 2;
                int leftChild = 2 * node + 1;
                int rightChild = 2 * node + 2;

                if (idx <= mid) {
                    update(leftChild, start, mid, idx, value);
                } else {
                    update(rightChild, mid + 1, end, idx, value);
                }
            }
        }
    }
}
```

```

        }

        tree[node] = combineOp.apply((T) tree[leftChild], (T) tree[rightChild]);
    }
}

T query(int l, int r) {
    if (l < 0 || r >= n || l > r) {
        throw new IllegalArgumentException("Invalid range");
    }
    return query(0, 0, n - 1, l, r);
}

@SuppressWarnings("unchecked")
private T query(int node, int start, int end, int l, int r) {
    if (r < start || l > end) {
        return zero;
    }

    if (l <= start && end <= r) {
        return (T) tree[node];
    }

    int mid = (start + end) / 2;
    int leftChild = 2 * node + 1;
    int rightChild = 2 * node + 2;

    T leftSum = query(leftChild, start, mid, l, r);
    T rightSum = query(rightChild, mid + 1, end, l, r);

    return combineOp.apply(leftSum, rightSum);
}

static void testMain() {
    Long[] arr = {1L, 3L, 5L, 7L, 9L};
    SegmentTree<Long> st = new SegmentTree<>(arr, 0L, (a, b) -> a + b);
    assert st.query(1, 3) == 15L;
    st.update(2, 10L);
    assert st.query(1, 3) == 20L;
    assert st.query(0, 4) == 30L;
}

```

# Sprague Grundy

---

sprague\_grundy.java

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

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

```
API:
```

- GrundyEngine(moveFunction): makes it easy to plug in any game.
- grundy(state): compute nimber for a state (must be hashable).
- grundyMulti(states): XOR of nimbers for independent subgames.
- isWinningPosition(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 array representation.

```
Requirements:
```

- State must be hashable and canonically represented (e.g., sorted arrays).
- moveFunction must not create cycles.

```
*/
```

```
import java.util.*;  
import java.util.function.Function;
```

```
public class sprague_grundy {
```

```
    // Minimum EXcludant: smallest non-negative integer not occurring in 'values'
```

```
    public static int mex(Collection<Integer> values) {  
        Set<Integer> s = new HashSet<>(values);  
        int g = 0;  
        while (s.contains(g)) {  
            g++;  
        }  
        return g;  
    }  
}
```

```
    public static class GrundyEngine<T> {  
        protected final Function<T, Collection<T>> moves;  
        private final Map<T, Integer> cache = new HashMap<>();
```

```
        public GrundyEngine(Function<T, Collection<T>> moveFunction) {  
            this.moves = moveFunction;  
        }
```

```
        public int grundy(T state) {  
            if (cache.containsKey(state)) {  
                return cache.get(state);  
            }
```

```
            Collection<T> nextStates = moves.apply(state);  
            if (nextStates.isEmpty()) {  
                cache.put(state, 0);  
                return 0;  
            }
```

```
            List<Integer> nimbers = new ArrayList<>();  
            for (T nextState : nextStates) {  
                nimbers.add(grundy(nextState));  
            }
```

```
            int result = mex(nimbers);  
            cache.put(state, result);  
            return result;  
        }
```



```

    public int GrundyMulti(Collection<T> states) {
        int result = 0;
        for (T state : states) {
            result ^= Grundy(state);
        }
        return result;
    }

    public boolean isWinningPosition(Collection<T> states) {
        return GrundyMulti(states) != 0;
    }
}

// Wrapper class for Kayles segments with proper equals/hashCode
public static class KaylesState {
    private final int[] segments;
    private final int hashCode;

    public KaylesState(int[] segments) {
        this.segments = segments.clone();
        Arrays.sort(this.segments); // Ensure canonical form
        this.hashCode = Arrays.hashCode(this.segments);
    }

    public KaylesState(List<Integer> segments) {
        this(segments.stream().mapToInt(Integer::intValue).toArray());
    }

    public int[] getSegments() {
        return segments.clone();
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        KaylesState that = (KaylesState) obj;
        return Arrays.equals(segments, that.segments);
    }

    @Override
    public int hashCode() {
        return hashCode;
    }

    @Override
    public String toString() {
        return Arrays.toString(segments);
    }
}

// Optional functionality (not always needed during competition)
public static Integer detectPeriod(List<Integer> seq, int minPeriod, Integer maxPeriod) {
    int n = seq.size();
    if (maxPeriod == null) {
        maxPeriod = n / 2;
    }
    for (int p = minPeriod; p <= maxPeriod; p++) {
        boolean ok = true;
        for (int i = 0; i < n; i++) {
            if (!seq.get(i).equals(seq.get(i % p))) {
                ok = false;
                break;
            }
        }
        if (ok) {
            return p;
        }
    }
    return null;
}

```

```

}

public static Collection<Integer> nimMovesSingleHeap(int n) {
    List<Integer> moves = new ArrayList<>();
    for (int k = 0; k < n; k++) {
        moves.add(k); // leave 0..n-1
    }
    return moves;
}

public static Function<Integer, Collection<Integer>> subtractionGameMovesFactory(
    Set<Integer> allowed) {
    List<Integer> allowedSorted = new ArrayList<>(allowed);
    Collections.sort(allowedSorted);

    return n -> {
        List<Integer> moves = new ArrayList<>();
        for (int d : allowedSorted) {
            if (d <= n) {
                moves.add(n - d);
            }
        }
        return moves;
    };
}

public static Collection<KaylesState> kaylesMovesHelper(KaylesState state) {
    Set<KaylesState> resultSet = new HashSet<>();
    int[] segments = state.getSegments();

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

        // Remove one pin at position i (0..n-1)
        for (int i = 0; i < n; i++) {
            int left = i;
            int right = n - i - 1;
            List<Integer> newSeg = new ArrayList<>();

            for (int j = 0; j < idx; j++) {
                newSeg.add(segments[j]);
            }
            if (left > 0) newSeg.add(left);
            if (right > 0) newSeg.add(right);
            for (int j = idx + 1; j < segments.length; j++) {
                newSeg.add(segments[j]);
            }

            resultSet.add(new KaylesState(newSeg));
        }

        // Remove two adjacent pins at position i,i+1 (0..n-2)
        for (int i = 0; i < n - 1; i++) {
            int left = i;
            int right = n - i - 2;
            List<Integer> newSeg = new ArrayList<>();

            for (int j = 0; j < idx; j++) {
                newSeg.add(segments[j]);
            }
            if (left > 0) newSeg.add(left);
            if (right > 0) newSeg.add(right);
            for (int j = idx + 1; j < segments.length; j++) {
                newSeg.add(segments[j]);
            }

            resultSet.add(new KaylesState(newSeg));
        }
    }

    return new ArrayList<>(resultSet);
}

```

```

}

public static Function<KaylesState, Collection<KaylesState>> kaylesMovesFactory() {
    return sprague_grundy::kaylesMovesHelper;
}

public static void testMain() {
    // Test Nim with larger values
    GrundyEngine<Integer> eng = new GrundyEngine<>(sprague_grundy::nimMovesSingleHeap);
    assert eng.grundy(42) == 42;
    assert eng.grundyMulti(Arrays.asList(17, 23, 31)) == 25; //  $17 \oplus 23 \oplus 31 = 25$ 
    assert eng.isWinningPosition(Arrays.asList(15, 27, 36)) == true; //  $15 \oplus 27 \oplus 36 = 48 \neq 0$ 

    // Test subtraction game {1,3,4} with period 7
    GrundyEngine<Integer> eng2 =
        new GrundyEngine<>(subtractionGameMovesFactory(Set.of(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
    GrundyEngine<KaylesState> eng3 = new GrundyEngine<>(kaylesMovesFactory());
    assert eng3.grundy(new KaylesState(new int[] {7})) == 2; //  $K(7) = 2$ 
    assert eng3.grundy(new KaylesState(new int[] {3, 5})) == 7; //  $K(3) \oplus K(5) = 3 \oplus 4 = 7$ 
}

```

# Topological Sort

---

topological\_sort.java

```
/*
Topological sorting algorithms for Directed Acyclic Graphs (DAG).

Provides two implementations:
1. Kahn's algorithm (BFS-based) - detects cycles
2. DFS-based algorithm - also detects cycles

Both return a topological ordering of vertices if the graph is a DAG,
or null if a cycle is detected.

Time complexity:  $O(V + E)$ 
Space complexity:  $O(V + E)$ 
*/

import java.util.*;

class topological_sort {
    static class TopologicalSort {
        private int n;
        private Map<Integer, List<Integer>> graph;

        TopologicalSort(int n) {
            this.n = n;
            this.graph = new HashMap<>();
            for (int i = 0; i < n; i++) {
                graph.put(i, new ArrayList<>());
            }
        }

        void addEdge(int u, int v) {
            graph.get(u).add(v);
        }

        List<Integer> kahSort() {
            int[] inDegree = new int[n];

            for (int u = 0; u < n; u++) {
                for (int v : graph.get(u)) {
                    inDegree[v]++;
                }
            }

            Queue<Integer> queue = new LinkedList<>();
            for (int i = 0; i < n; i++) {
                if (inDegree[i] == 0) {
                    queue.offer(i);
                }
            }

            List<Integer> result = new ArrayList<>();

            while (!queue.isEmpty()) {
                int u = queue.poll();
                result.add(u);

                for (int v : graph.get(u)) {
                    inDegree[v]--;
                    if (inDegree[v] == 0) {
                        queue.offer(v);
                    }
                }
            }

            if (result.size() != n) {
                return null; // Cycle detected
            }
        }
    }
}
```

```

        return result;
    }

    List<Integer> dfsSort() {
        Set<Integer> visited = new HashSet<>();
        Set<Integer> recStack = new HashSet<>();
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < n; i++) {
            if (!visited.contains(i)) {
                if (!dfsVisit(i, visited, recStack, stack)) {
                    return null; // Cycle detected
                }
            }
        }

        List<Integer> result = new ArrayList<>();
        while (!stack.isEmpty()) {
            result.add(stack.pop());
        }

        return result;
    }

    private boolean dfsVisit(
        int u, Set<Integer> visited, Set<Integer> recStack, Stack<Integer> stack) {
        visited.add(u);
        recStack.add(u);

        for (int v : graph.get(u)) {
            if (!visited.contains(v)) {
                if (!dfsVisit(v, visited, recStack, stack)) {
                    return false;
                }
            } else if (recStack.contains(v)) {
                return false; // Cycle detected
            }
        }

        recStack.remove(u);
        stack.push(u);
        return true;
    }

    boolean hasCycle() {
        return kahnSort() == null;
    }

    List<Integer> longestPath(int source) {
        List<Integer> topoOrder = kahnSort();
        if (topoOrder == null) {
            return null; // Has cycle
        }

        Map<Integer, Integer> dist = new HashMap<>();
        Map<Integer, Integer> parent = new HashMap<>();

        for (int i = 0; i < n; i++) {
            dist.put(i, Integer.MIN_VALUE);
        }
        dist.put(source, 0);

        for (int u : topoOrder) {
            if (dist.get(u) != Integer.MIN_VALUE) {
                for (int v : graph.get(u)) {
                    if (dist.get(u) + 1 > dist.get(v)) {
                        dist.put(v, dist.get(u) + 1);
                        parent.put(v, u);
                    }
                }
            }
        }
    }

```

```

    }

    // Find vertex with maximum distance
    int maxDist = Integer.MIN_VALUE;
    int endVertex = -1;
    for (int i = 0; i < n; i++) {
        if (dist.get(i) > maxDist) {
            maxDist = dist.get(i);
            endVertex = i;
        }
    }

    if (endVertex == -1 || maxDist == Integer.MIN_VALUE) {
        return Arrays.asList(source);
    }

    // Reconstruct path
    List<Integer> path = new ArrayList<>();
    int current = endVertex;
    while (current != source) {
        path.add(current);
        current = parent.get(current);
    }
    path.add(source);
    Collections.reverse(path);

    return path;
}

static void testMain() {
    TopologicalSort ts = new TopologicalSort(6);
    int[][] edges = {{5, 2}, {5, 0}, {4, 0}, {4, 1}, {2, 3}, {3, 1}};
    for (int[] edge : edges) {
        ts.addEdge(edge[0], edge[1]);
    }

    List<Integer> kahnResult = ts.kahnSort();
    List<Integer> dfsResult = ts.dfsSort();

    assert kahnResult != null;
    assert dfsResult != null;
    assert !ts.hasCycle();

    // Test with cycle
    TopologicalSort tsCycle = new TopologicalSort(3);
    tsCycle.addEdge(0, 1);
    tsCycle.addEdge(1, 2);
    tsCycle.addEdge(2, 0);
    assert tsCycle.hasCycle();
}

```

# Union Find

---

union\_find.java

```
/*  
Union-Find (Disjoint Set Union) data structure with path compression and union by rank.
```

Supports:

- `find(x)`: Find the representative of the set containing `x` -  $O(\alpha(n))$  amortized
- `union(x, y)`: Merge the sets containing `x` and `y` -  $O(\alpha(n))$  amortized
- `connected(x, y)`: Check if `x` and `y` are in the same set -  $O(\alpha(n))$  amortized

Space complexity:  $O(n)$

$\alpha(n)$  is the inverse Ackermann function, which is effectively constant for all practical values of  $n$ .  
\*/

```
class union_find {  
    static class UnionFind {  
        private int[] parent;  
        private int[] rank;  
  
        UnionFind(int n) {  
            parent = new int[n];  
            rank = new int[n];  
            for (int i = 0; i < n; i++) {  
                parent[i] = i;  
                rank[i] = 0;  
            }  
        }  
  
        int find(int x) {  
            if (parent[x] != x) {  
                parent[x] = find(parent[x]); // Path compression  
            }  
            return parent[x];  
        }  
  
        int union(int x, int y) {  
            int rootX = find(x);  
            int rootY = find(y);  
  
            if (rootX == rootY) {  
                return rootX;  
            }  
  
            // Union by rank  
            if (rank[rootX] < rank[rootY]) {  
                parent[rootX] = rootY;  
                merge(rootY, rootX);  
                return rootY;  
            } else if (rank[rootX] > rank[rootY]) {  
                parent[rootY] = rootX;  
                merge(rootX, rootY);  
                return rootX;  
            } else {  
                parent[rootY] = rootX;  
                merge(rootX, rootY);  
                rank[rootX]++;  
                return rootX;  
            }  
        }  
  
        boolean connected(int x, int y) {  
            return find(x) == find(y);  
        }  
  
        void merge(int root, int child) {  
            // Override to define custom merge behavior when sets are united  
        }  
    }  
}
```

```

}

static class Test extends UnionFind {
    private int[] size;

    Test(int n) {
        super(n);
        size = new int[n];
        for (int i = 0; i < n; i++) {
            size[i] = 1;
        }
    }

    @Override
    void merge(int root, int child) {
        size[root] += size[child];
    }

    int getSize(int x) {
        return size[find(x)];
    }
}

static void testMain() {
    Test a = new Test(3);
    int d = a.union(0, 1);
    int e = a.union(d, 2);
    assert a.getSize(e) == 3;
    assert a.getSize(a.find(0)) == 3;
}

```