Hackpack

Team Fireball

Lara Aydin, Isaac Ehlers, Steven Petryk

General Programming

Comparable

Sorting from smallest to largest:

```
// a.compareTo(b) < 0   =>   a < b   =>   a - b < 0
public int compareTo (Thing other) {
  return value - other.value;
}
```

Geometry classes

```java
// Code modified from Arup Guha's geometry routines
// Found at http://www.cs.ucf.edu/~dmarino/progcontests/cop4516/samplecode/Test2DGeo.java

class Point {
  public double x, y, z;

  public Point(double _x, double _y) {
    this(_x, _y, 0);
  }

  public Point(double _x, double _y, double _z) {
    x = _x;
    y = _y;
    z = _z;
  }

  public boolean isStraightLineTo (Point mid, Point end) {
    Vector from = new Vector(this, mid);
    Vector to = new Vector(mid, end);

    return from.isStraightLineTo(to);
  }

  public boolean isRightTurn(Point mid, Point end) {
    Vector from = new Vector(this, mid);
    Vector to = new Vector(mid, end);

    return from.isLeftTurnTo(to);
  }

  public Vector getVector(Point to) {
    return new Vector(to.x - x, to.y - y, to.z - z);
  }

  public String toString () {
    return "<" + x + ", " + y + ">";
  }
}
```

```
class Vector {
  public double x, y, z;

  public Vector(double _x, double _y) {
    this(_x, _y, 0);
  }

  public Vector(double _x, double _y, double _z) {
    x = _x;
    y = _y;
    z = _z;
  }

  public Vector (Point start, Point end) {
    x = end.x - start.x;
    y = end.y - start.y;
  }

  public double dot (Vector other) {
    return this.x * other.x + this.y * other.y + this.z * other.z;
  }

  public Vector crossProduct(Vector other) {
  return new Vector((y * other.z) - (other.y * z), (z * other.x) - (other.z * x), (x * other.y)
  }

  public double magnitude() {
    return Math.sqrt((x * x) + (y * y) + (z * z));
  }

  public double angle(Vector other) {
    return Math.acos(this.dot(other) / magnitude() / other.magnitude());
  }

  public double signedCrossMag(Vector other) {
    return this.x * other.y - other.x * this.y;
  }

  public double crossProductMagnitude (Vector other) {
    return Math.abs(signedCrossMag(other));
  }

  public double referenceAngle () {
    return Math.atan2(y, x);
  }

  public boolean isStraightLineTo (Vector other) {
    return signedCrossMag(other) == 0;
  }

  public boolean isLeftTurnTo (Vector other) {
    return signedCrossMag(other) > 0;
  }
}
```

```
class Line {
  final public static double EPSILON = 1e-9;

  public Point p, end;
  public Vector dir;

  public Line(Point _start, Point _end) {
    p = _start;
    end = _end;
    dir = new Vector(p, end);
  }

  public Point intersect(Line other) {
    double den = det(dir.x, -other.dir.x, dir.y, -other.dir.y);
    if (Math.abs(den) < EPSILON) return null;

    double numLambda = det(other.p.x-p.x, -other.dir.x, other.p.y-p.y, -other.dir.y);
    return eval(numLambda/den);
  }

  public Point getPoint(double t) {
    return new Point(p.x + dir.x * t, p.y + dir.y * t, p.z + dir.z * t);
  }

  public double distance(Point other) {
    Vector toPt = new Vector(p, other);
    return dir.crossProductMagnitude(toPt) / dir.magnitude();
  }

  public Point eval(double lambda) {
    return new Point(p.x + lambda * dir.x, p.y + lambda * dir.y);
  }

  public static double det(double a, double b, double c, double d) {
    return a * d - b * c;
  }
}

class Plane {
  public Point a, b, c;
  public Vector normalVector;
  public double distanceToOrigin;

  public Plane(Point _a, Point _b, Point _c) {
    a = _a;
    b = _b;
    c = _c;
    Vector v1 = a.getVector(b);
    Vector v2 = a.getVector(c);
    normalVector = v1.crossProduct(v2);
    distanceToOrigin = (normalVector.x * a.x) + (normalVector.y * a.y) + (normalVector.z * a.z)
  }

  public boolean onPlane(Point p) {
    return (normalVector.x * p.x) + (normalVector.y * p.y) + (normalVector.z * p.z) == distance
  }
}
```

Combination Generation

Example: print out all alphabetic strings of a given length.

```java
class WordInventor {
  static List<String> results;

  public static List<String> generateCombinations (int length) {
    results = new ArrayList<String>();
    generateCombinations(length, "", 0);
    return results;
  }

  public static void generateCombinations (int length, String accumulator, int k) {
    if (k == length) {
      results.add(accumulator);
      return;
    }

    for (char c = 'a'; c <= 'z'; c++) {
      generateCombinations(length, accumulator + c, k + 1);
    }
  }
}
```

Permutation Generation

```java
class Permuter {
  public static <T> List<List<T>> permute (List<T> items) {
    return permute(items, new ArrayList<>());
  }

  public static <T> List<List<T>> permute (List<T> items, List<T> accumulator) {
    List<List<T>> results = new ArrayList<>();

    if (items.isEmpty()) {
      results.add(accumulator);
      return results;
    }

    for (T item : items) {
      List<T> itemsCopy = new ArrayList<>(items);
      List<T> accumulatorCopy = new ArrayList<>(accumulator);

      accumulatorCopy.add(item);
      itemsCopy.remove(item);
      results.addAll(permute(itemsCopy, accumulatorCopy));
    }

    return results;
  }
}
```

GCD

```java
public static int gcd (int a, int b) {
  return b == 0 ? a : gcd(b, a%b);
}
```

LCM

```java
public static int lcm (int a, int b) {
  return a * (b / gcd(a, b));
}
```

Graphs

```java
class Node {
  int value;
  public List<Edge<Node>> children;
  public Node () { this(0); }
  public Node (int _value) { value = _value; children = new ArrayList<Edge<Node>>(); }

  public Node addChild (Node child, int weight) {
    return addChild(child, weight, true);
  }

  public Node addChild (Node child, int weight, boolean reciprocate) {
    children.add(new Edge<>(this, child, weight));
    if (reciprocate) child.addChild(this, weight, false); // if undirected graph

    return this;
  }
}

class Edge<T> implements Comparable<Edge> {
  T node, from; int weight;
  Edge (T _node, int _weight) { this(null, _node, _weight); }
  Edge (T _from, T _node, int _weight) { from = _from; node = _node; weight = _weight; }

  @Override
  public int compareTo (Edge other) {
    return weight - other.weight;
  }
}
```

Kruskal's Algorithm

```
class Kruskal {
  public static int getMSTWeight (Node start, int numNodes) {
    Queue<Edge<Node>> edges = new PriorityQueue<>();
    edges.add(new Edge<Node>(null, start, 0));

    int result = 0;

    DisjointSet ds = new DisjointSet(5);

    int nodesReached = 0;

    while (!edges.isEmpty()) {
      Edge<Node> currentEdge = edges.poll();
      Node currentNode = currentEdge.node;

      boolean merged = true;
      if (currentEdge.from != null) {
        merged = ds.union(currentEdge.from.value, currentEdge.node.value);
      }

      if (!merged) continue;
      nodesReached++;
      edges.addAll(currentNode.children);
      result += currentEdge.weight;
    }

    return nodesReached == numNodes ? result : -1;
  }
}

class DisjointSet {
  int[] parent, rank;

  public DisjointSet (int n) {
    rank = new int[n]; parent = new int[n];

    for (int i = 0; i < n; i++) parent[i] = i;
  }

  public int find (int value) {
    if (parent[value] != value) parent[value] = find(parent[value]);
    return parent[value];
  }

  public boolean union (int a, int b) {
    int aRoot = find(a);
    int bRoot = find(b);
    if (aRoot == bRoot) return false;

    if      (rank[aRoot] < rank[bRoot]) parent[aRoot] = bRoot;
    else if (rank[aRoot] > rank[bRoot]) parent[bRoot] = aRoot;
    else {
      parent[bRoot] = aRoot;
      rank[aRoot]++;
    }

    return true;
  }
}
```

Prim's Algorithm

```java
class Prim {
  public static int getMSTWeight (Node start, int numNodes) {
    Queue<Edge<Node>> pq = new PriorityQueue<>();
    Set<Node> visited = new HashSet<>();

    int result = 0;

    pq.add(new Edge<Node>(start, 0));

    while (!pq.isEmpty()) {
      Edge<Node> current = pq.poll();
      Node currentNode = current.node;
      if (!visited.add(currentNode)) continue;

      result += current.weight;

      pq.addAll(currentNode.children);
    }

    if (visited.size() == numNodes) {
      return result;
    } else {
      return -1;
    }
  }
}
```

Depth First Search

```
class DFS {
  public static boolean canReachNode (Node start, Node target) {
    Set<Node> visited = new HashSet<>();
    Deque<Node> queue = new ArrayDeque<>();
    queue.push(start);

    while (!queue.isEmpty()) {
      Node current = queue.pop();

      if (!visited.add(current)) continue;
      if (current == target) return true;

      for (Edge<Node> edge : current.children) {
        queue.push(edge.node);
      }
    }

    return false;
  }
}
```

Breadth First Search

```java
class BFS {
  public static int distanceToNode (Node start, Node target) {
    Set<Node> visited = new HashSet<>();
    Deque<NodeWithDistance> queue = new ArrayDeque<>();
    queue.add(new NodeWithDistance(start, 0));

    while (!queue.isEmpty()) {
      NodeWithDistance current = queue.poll();

      if (!visited.add(current.node)) continue;
      if (current.node == target) return current.distance;

      for (Edge<Node> edge : current.node.children) {
        queue.add(new NodeWithDistance(edge.node, current.distance + 1));
      }
    }

    return -1;
  }

  static class NodeWithDistance {
    Node node; int distance;
    public NodeWithDistance (Node _node, int _distance) { node = _node; distance = _distance; }
  }
}
```

Topological Sort

```
class TopologicalSort {
  public static ArrayList<Integer> sort(ArrayList<ArrayList<Node>> adjList) {
    ArrayList<Integer> sorted = new ArrayList<Integer>();
    int[] inDegrees = new int[adjList.size()];
    Arrays.fill(inDegrees, 0);
    Queue<Integer> q = new LinkedList<Integer>();

    for(int i = 0; i < adjList.size(); i++)
      for(int j = 0; j < adjList.get(i).size(); j++)
        inDegrees[adjList.get(i).get(j).value]++;

    for(int i = 0; i < inDegrees.length; i++)
      if(inDegrees[i] == 0)
        q.offer(i);

    while(!q.isEmpty()) {
      int currNodeVal = q.poll();
      sorted.add(currNodeVal);
      for(Node n : adjList.get(currNodeVal)) {
        inDegrees[n.value]--;
        if(inDegrees[n.value] == 0)
          q.offer(n.value);
      }
    }

    if(sorted.size() < adjList.size()) {
      System.out.println("Warning: Graph contains a cycle!");
      return sorted;
    }
    else
      return sorted;
  }
}
```

Floyd-Warshall's Algorithm

```java

class FloydWarshalls {
  public static int[][] floydwarshalls(int[][] matrix) {
    int n = matrix.length;
    int[][] sp = new int[n][n];

    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        sp[i][j] = (i == j) ? 0 : matrix[i][j];

    // Floyd-Warshall's
    for (int k = 1; k <= n; k++)
      for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
          sp[i][j] = Math.min(sp[i][j], sp[i][k-1] + sp[k-1][j]);

    // Negative cycle detection.
    for (int i = 0; i < n; i++)
      if (sp[i][i] < 0)
        return new int[1][1];

    return sp;
  }
}
```

Bellman Ford's Algorithm

```java
class BellmanFord {
  final public static int oo = (int)10e9;

  public static Map<Node, Integer> distances(List<Edge<Node>> graph, int numVertices, Node sour
    Map<Node, Integer> estimates = new HashMap<>(numVertices);
    estimates.put(source, 0);

    for (int i = 0; i < numVertices - 1; i++) {
      for (Edge<Node> edge : graph) {
        if (estimates.getOrDefault(edge.from, oo) + edge.weight < estimates.getOrDefault(edge.n
          estimates.put(edge.node, estimates.get(edge.from) + edge.weight);
        }
      }

    }

    return estimates;
  }
}
```

Dijkstra's Algorithm

```java
class Dijkstras {
  public static LinkedList<Vertex> dijkstras(int source, int[][] matrix) {
    int[] dist = new int[matrix.length];
    boolean[] visited = new boolean[matrix.length];
    int numVisited = 0;
    PriorityQueue<Vertex> queue = new PriorityQueue<>();
    LinkedList<Vertex> path = new LinkedList<>();

    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[source] = 0;


    for(int i = 0; i < matrix.length; i++)
      queue.add(new Vertex(i, dist[i]));

    while (!queue.isEmpty() && numVisited < matrix.length) {
      Vertex vertex = queue.remove();
      if(visited[vertex.id]) continue;
      visited[vertex.id] = true;
      path.add(vertex);

      for(int i = 0; i < matrix.length; i++) {
        if(matrix[vertex.id][i] > 0 && !visited[i] && dist[vertex.id] + matrix[vertex.id][i] <
          dist[i] = dist[vertex.id] + matrix[vertex.id][i];
          queue.add(new Vertex(i, dist[i]));
        }
      }
    }

    return path;
  }

  static class Vertex {
    int id; int distance;
    public Vertex (int _id, int _distance) {
      id = _id; distance = _distance;
    }
  }
}
```

Network Flow

```java
class NetworkFlow {
  static int numNodes;
  static int[][] capMat;
  static int source;
  static int sink;

  // Takes pre-filled adjacency matrix denoting capacities with source node
  // at n  and sink node at n - 1.
  public static int edmondsKarp(int[][] capacityMatrix) {
    numNodes = capacityMatrix.length;
    capMat = capacityMatrix;
    source = numNodes - 2;
    sink = numNodes - 1;

    return ek();
  }

  public static int ek() {
    int flow = 0;
    while(true) {
      int residual = ekBFS();
      if(residual == 0)
        break;

      flow += residual;
    }
    return flow;
  }

  // Need tailored BFS for Edmond Karp algorithm.
  // Used to find shortest augmenting path.
  public static int ekBFS() {
    int[] min = new int[numNodes];
    int[] previous = new int[numNodes];
    Queue<Integer> q = new LinkedList<Integer>();
    min[source] = (int) 1e9;
    Arrays.fill(previous, -1);
    previous[source] = source;
    q.offer(source);

    while(!q.isEmpty()) {
      int currNode = q.poll();
      if(currNode == sink)
        break;

      for(int i = 0; i < numNodes - 2; i++) {
        if(previous[i] == -1 && capMat[currNode][i] > 0) {
          previous[i] = currNode;
          min[i] = Math.min(capMat[currNode][i], min[currNode]);
          q.offer(i);
        }
      }
    }

    if(min[sink] == 0)
      return 0;

    int node1 = previous[sink];
    int node2 = sink;
    int flow = min[sink];

    while(node2 != source) {
      capMat[node1][node2] -= flow;
      capMat[node2][node1] += flow;
      node2 = node1;
      node1 = previous[node1];
    }

    return flow;
  }
}
```

Dynamic Programming

Matrix Chain Multiplication

```
class MCM {
  static int[][] memo;

  // matrices array of form {a, b, c, d} (n = 4) such that
  // there are n - 1 = 3 matrices represented with dimensions:
  // (a x b), (b x c), (c x d) -- start initially 1, end = n - 1.
  public static int minMults(int[] matrices) {
    memo = new int[matrices.length][matrices.length];
    for(int i = 0; i < matrices.length - 1; i++) {
      Arrays.fill(memo[i], -1);
    }

    return minMults(matrices, 1, matrices.length - 1);
  }

  public static int minMults(int[] matrices, int start, int end) {
    int dim = matrices[start] * 100 + matrices[end];
    if(memo[start][end] != -1)
      return memo[start][end];

    if(start == end)
      return 0;

    int min = (int) 1e9;
    for(int i = start; i < end; i++) {
      int currCount = minMults(matrices, start, i) +
              minMults(matrices, i + 1, end) +
              matrices[start - 1] * matrices[i] * matrices[end];

      if(currCount < min)
        min = currCount;
    }

    memo[start][end] = min;
    return min;
  }
}
```

Longest Common Subsequence

```java
class LCS {
  public static int longestCommonSubsequenceLength (String x, String y) {
    int lengths[][] = new int[x.length() + 1][y.length() + 1];

    Arrays.fill(lengths[0], 0);
    for (int i = 0; i < lengths.length; i++) lengths[i][0] = 0;

    for (int i = 1; i < lengths.length; i++) {
      for (int j = 1; j < lengths[0].length; j++) {
        if (x.charAt(i - 1) == y.charAt(j - 1)) {
          lengths[i][j] = lengths[i - 1][j - 1] + 1;
        } else {
          lengths[i][j] = Math.max(lengths[i - 1][j], lengths[i][j - 1]);
        }
      }
    }

    return lengths[lengths.length - 1][lengths[0].length - 1];
  }
}
```

Knapsack

```java
class Knapsack {
  public static int knapsack (int capacity, int[] weights, int[] values, boolean allowDups) {
    int n = weights.length;
    int[] dp = new int[capacity + 1];

    for (int i = 0; i < n; i++) {
      for (
        int w = allowDups ? weights[i] : capacity;
        allowDups ? w <= capacity : w >= weights[i];
      ) {
        dp[w] = Math.max(dp[w], dp[w-weights[i]] + values[i] );

        if (allowDups) w++; else w--;
      }
    }

    return dp[capacity];
  }
}
```

"Dinner" Example

```java
class dinner {
  static long[] memo;

  // public static void main(String[] args) {
  //   ...fills up the memo table
  // }

  public static long numSols (int total) {
    if (total < 0) {
      return 0;
    }

    if (memo[total] != -1) {
      return memo[total];
    }

    if (total == 0) {
      return 1;
    }

    long solsWith2 = numSols(total - 2);
    long solsWith5 = numSols(total - 5);
    long solsWith10 = numSols(total - 10);

    return memo[total] = solsWith2 + solsWith5 + solsWith10;
  }
}
```

"Stick" example

```java
class sticks {
  public static int[] subSticks;
  public static int[][] joinSizes;
  public static int[][] memo;

  // public static void main (String[] args) {
  //   ...
  //
  //   for (int i = 0; i < numSubsticks; i++) {
  //     joinSizes[i][i] = subSticks[i];
  //
  //     for (int j = i + 1; j < numSubsticks; j++) {
  //       joinSizes[i][j] = joinSizes[i][j-1] + subSticks[j];
  //     }
  //   }
  //
  //   ...
  // }

  public static int solve(int start, int end) {
    if (start == end) return 0;
    if (memo[start][end] != -1) return memo[start][end];

    int res = Integer.MAX_VALUE;

    for (int split = start; split < end; split++) {
      int leftCost = solve(start, split);
      int rightCost = solve(split + 1, end);

      int leftSize = joinSizes[start][split];
      int rightSize = joinSizes[split + 1][end];

      res = Math.min(res, leftCost + rightCost + leftSize + rightSize);
    }

    return memo[start][end] = res;
  }
}
```

Intersection tests

Line-Line Intersection

```
class LineLineIntersection {

  public static Point intersection(Line line1, Line line2) {
    return line1.intersect(line2);
  }
}
```

Line-Plane Intersection

```
class LinePlaneIntersection {
  final public static double EPSILON = 1e-9;

  public static Point intersection(Plane p, Line l) {

    double t = (p.normalVector.x * l.dir.x) +
               (p.normalVector.y * l.dir.y) +
               (p.normalVector.z * l.dir.z);

    if(Math.abs(t) < EPSILON)
      return null;

    double parameter = p.distanceToOrigin -
                       (p.normalVector.x * l.p.x) -
                       (p.normalVector.y * l.p.y) -
                       (p.normalVector.z * l.p.z);

    return l.getPoint(parameter / t);
  }
}
```

Geometry

Polygon Area

```java
class PolygonArea {
  // Shape must be made of points in either clockwise or
  // counter-clockwise order (cannot be self-intersecting).
  public static double getArea2D(ArrayList<Point> shape) {
    double area = 0;
    Point curr;
    Point next;

    for(int i = 0; i < shape.size(); i++) {
      curr = shape.get(i);
      if(i == shape.size() - 1)
        next = shape.get(0);
      else
        next = shape.get(i + 1);

      area += 0.5 * (next.x - curr.x) * (next.y + curr.y);
    }
    return Math.abs(area);
  }
}
```

Convex Hull

```
class ConvexHullSolver {
  int numPoints;
  Queue<Point> initialPoints;
  Queue<Point> sortedPoints;
  Point firstPoint;

  public static Comparator<Point> getLowerLeftComparator() {
    return new Comparator<Point>() {
      @Override
      public int compare(Point o1, Point o2) {
        if (o1.y != o2.y) return Double.compare(o1.y, o2.y);

        return Double.compare(o1.x, o2.x);
      }
    };
  }

  public static Comparator<Point> getReferenceAngleComparator (final Point initialPoint) {
    return new Comparator<Point>() {
      @Override
      public int compare(Point p1, Point p2) {
        if (p1 == initialPoint) return -1;
        if (p2 == initialPoint) return 1;

        Vector v1 = new Vector(initialPoint, p1);
        Vector v2 = new Vector(initialPoint, p2);

        if (Math.abs(v1.referenceAngle() - v2.referenceAngle()) < 1e-4) {
          return Double.compare(v1.magnitude(), v2.magnitude());
        }

        return Double.compare(v1.referenceAngle(), v2.referenceAngle());
      }
    };
  }

  public ConvexHullSolver (int _numPoints) {
    numPoints = _numPoints;
    initialPoints = new PriorityQueue<>(numPoints, getLowerLeftComparator());
  }

  public void addPoint (Point point) {
    initialPoints.add(point);
  }

  public Stack<Point> solve () {
    sortPoints();

    Stack<Point> pointStack = new Stack<>();

    if (sortedPoints.size() <= 3) {
      List<Point> points = new ArrayList<>(sortedPoints);

      if (points.get(0).isStraightLineTo(points.get(1), points.get(2))) {
        pointStack.add(points.get(0));
        pointStack.add(points.get(1));
      } else {
        pointStack.addAll(sortedPoints);
      }

      return pointStack;
    }

    pointStack.push(sortedPoints.poll());
    pointStack.push(sortedPoints.poll());

    while (!sortedPoints.isEmpty()) {
      Point endPoint = sortedPoints.poll();
      Point midPoint = pointStack.pop();
      Point prevPoint = pointStack.pop();

      while (!prevPoint.isRightTurn(midPoint, endPoint)) {
        if (pointStack.isEmpty()) {
          midPoint = endPoint;
          endPoint = sortedPoints.poll();
        } else {
          midPoint = prevPoint;
          prevPoint = pointStack.pop();
        }
      }

      pointStack.push(prevPoint);
      pointStack.push(midPoint);
      pointStack.push(endPoint);
    }

    return pointStack;
  }

  public void sortPoints () {
    firstPoint = initialPoints.peek();

    sortedPoints = new PriorityQueue<>(numPoints, getReferenceAngleComparator(firstPoint));
    sortedPoints.addAll(initialPoints);
  }
}
```

Point in Polygon

```java
class PointInPolygon {
  // Shape must be made of points in either clockwise or
  // counter-clockwise order (cannot be self-intersecting).
  public static int inPolygon(Point p, ArrayList<Point> shape) {
    double errorFactor = 1e-7;
    double angleTotal = 0;
    Vector curr;
    Vector next;

    for(int i = 0; i < shape.size(); i++) {
      if(p.equals(shape.get(i)))
        return 1; // Point on vertex of polygon

      curr = new Vector(p, shape.get(i));
      if(i == shape.size() - 1)
        next = new Vector(p, shape.get(0));
      else
        next = new Vector(p, shape.get(i + 1));

      double angle = curr.angle(next);
      if(!(Math.abs(angle - Math.PI) < errorFactor))
        angleTotal += angle;
    }
    angleTotal = Math.abs(angleTotal);

    if(Math.abs(angleTotal - (2 * Math.PI)) < errorFactor)
      return 0; // Point in polygon
    else if(Math.abs(angleTotal - (Math.PI)) < errorFactor)
      return 1; // Point on edge of polygon
    else
      return 2; // Point outside of polygon

  }
}
```

Tests

Dr. Guha, if you'd like to test the hackpack, all of this code can be found on GitHub.

To run these tests, you can clone the repository, and simply run `make test`. This will compile the example code in the hackpack (it runs a shell script that strips away everything other than Java code, and then compiles the result). This is how we tested our code along the way.

```java
public class hackpack {
  public static boolean failures = false;

  public static void main (String args[]) {
    testCombinationGeneration();
    testPermutationGeneration();
    testGCD();
    testLCM();
    testDisjointSet();
    testKruskals();
    testPrims();
    testDFS();
    testBFS();
    testFloydWarshalls();
    testDijkstras();
    testLCS();
    testKnapsack();
    testConvexHull();

    if (!failures) {
      handleSuccess();
    }
  }

  public static void testCombinationGeneration () {
    List<String> results = WordInventor.generateCombinations(3);
    assertEqual(results.size(), (int)Math.pow(26, 3));
  }

  public static void testPermutationGeneration () {
    List<Integer> items = new ArrayList<>();
    items.add(1); items.add(2); items.add(3); items.add(4); items.add(5);

    List<List<Integer>> results = Permuter.permute(items);
    assertEqual(results.size(), 120); // 5!
  }

  public static void testGCD () {
    assertEqual(MathUtils.gcd(1, 1), 1);
    assertEqual(MathUtils.gcd(5, 10), 5);
    assertEqual(MathUtils.gcd(15, 3), 3);
  }

  public static void testLCM () {
    assertEqual(MathUtils.lcm(1, 1), 1);
    assertEqual(MathUtils.lcm(5, 10), 10);
    assertEqual(MathUtils.lcm(8, 3), 24);
  }

  public static void testDisjointSet () {
    DisjointSet set = new DisjointSet(5);

    set.union(1, 2);
    set.union(1, 3);
    assertEqual(set.find(2), 1);
    assertEqual(set.find(3), 1);
    assertEqual(set.find(4), 4);
  }

  public static void testKruskals () {
    Node a = new Node(0), b = new Node(1), c = new Node(2), d = new Node(3), e = new Node(4);

    a.addChild(b, 1);
    a.addChild(c, 2);
    c.addChild(e, 3);
    e.addChild(a, 4);

    assertEqual(Kruskal.getMSTWeight(a, 5), -1);

    e.addChild(d, 5);
    assertEqual(Kruskal.getMSTWeight(a, 5), 11);
  }

  public static void testPrims () {
    Node a = new Node(0), b = new Node(1), c = new Node(2), d = new Node(3), e = new Node(4);

    a.addChild(b, 1);
    a.addChild(c, 2);
    c.addChild(e, 3);
    e.addChild(a, 4);

    assertEqual(Prim.getMSTWeight(a, 5), -1);

    e.addChild(d, 5);
    assertEqual(Prim.getMSTWeight(a, 5), 11);
  }

  public static void testDFS () {
    Node start = new Node();
    Node reachable = new Node();
    Node unreachable = new Node();

    start
      .addChild(new Node(), 1)
      .addChild(new Node(), 1)
      .addChild(
        new Node()
          .addChild(reachable, 1)
          .addChild(new Node(), 1),
        1
      );

    assertTrue(DFS.canReachNode(start, reachable), "Expected node to be reachable");
    refute(DFS.canReachNode(start, unreachable), "Expected node to be unreachable");
  }

  public static void testBFS () {
    Node start = new Node();
    Node reachable = new Node();
    Node unreachable = new Node();

    start
      .addChild(new Node(), 1)
      .addChild(new Node(), 1)
      .addChild(
```

```java
        new Node()
          .addChild(reachable, 1)
          .addChild(new Node(), 1),
        1
      );

    assertEqual(BFS.distanceToNode(start, reachable), 2);
    assertEqual(BFS.distanceToNode(start, unreachable), -1);
  }

  public static void testFloydWarshalls() {
    // int[][] matrix = new int[4][4];
    // int[][] result = new int[4][4];
    //FloydWarshalls fw = new FloydWarshalls(4, matrix);

    int[][] matrix = {{1000000000, 1000000000, -2, 1000000000},
              {4, 1000000000, 3, 1000000000},
              {1000000000, 1000000000, 1000000000, 2}, {1000000000, -1, 1000000000, 1000000000
    int[][] result = {{0, -1, -2, 0},
              {4, 0, 2, 4},
              {5, 1, 0, 2},
              {3, -1, 1, 0}};

    assertArraysEqual(FloydWarshalls.floydwarshalls(matrix), result);
  }

  public static void testDijkstras() {
    int[][] matrix = new int[9][9];
  }

  public static void testLCS () {
    String x = "123456789";
    String y = "13597341234569";
                    // ^^^^^^^

    assertEqual(LCS.longestCommonSubsequenceLength(x, y), 7);
  }

  public static void testKnapsack () {
    int weights[] = new int[] { 3, 2, 6, 8, 1, 3 };
    int values[] = new int[] { 7, 5, 12, 20, 3, 6 };

    assertEqual(Knapsack.knapsack(1, weights, values, false), 3);
    assertEqual(Knapsack.knapsack(2, weights, values, false), 5);
    assertEqual(Knapsack.knapsack(10, weights, values, false), 25);
    assertEqual(Knapsack.knapsack(23, weights, values, false), 53);
  }

  public static void testConvexHull () {
    ConvexHullSolver solver = new ConvexHullSolver(5);
    List<Point> points = new ArrayList<>();

    Point topLeft     = new Point(2, 0), topRight   = new Point(2, 2),
            lowerLeft = new Point(0, 0), lowerRight = new Point(0, 2),
            middle    = new Point(1, 1);

    solver.addPoint(lowerLeft);
    solver.addPoint(lowerRight);
    solver.addPoint(topLeft);
    solver.addPoint(topRight);
    solver.addPoint(middle);

    Stack<Point> hull = solver.solve();

    assertContains(hull, lowerLeft);
    assertContains(hull, lowerRight);
    assertContains(hull, topLeft);
    assertContains(hull, topRight);
    refuteContains(hull, middle);
  }

  /*
   * Low-level test code. Don't worry about this too much.
   */

  public static final String ANSI_RESET = "\u001B[0m";
  public static final String ANSI_RED = "\u001B[31m";
  public static final String ANSI_GREEN = "\u001B[32m";

  private static void handleSuccess () {
    System.out.println(ANSI_GREEN + "✓ All tests passed" + ANSI_RESET);
  }

  private static void handleTestFailure (TestFailure e) {
    failures = true;

    String failingTest = "";

    outer: for (StackTraceElement element : e.getStackTrace()) {
      String name = element.getMethodName();

      if (!name.startsWith("assert") && !name.startsWith("throwOn")) {
        failingTest = name;
        break outer;
      }
    }

    System.out.println(ANSI_RED + "× " + failingTest + " failed: " + e.getMessage() + ANSI_RES

    e.printStackTrace();
  }

  private static <T> void assertEqual (T a, T b) {
    assertTrue(a.equals(b), String.format("Expected %s to equal %s", a, b));
  }

  private static <T> void assertArraysEqual (T[] a, T[] b) {
    assertTrue(
      Arrays.deepEquals(a, b),
      String.format("Expected %s to match %s", Arrays.deepToString(a), Arrays.deepToString(b))
    );
  }

  private static <T> void assertContains (List<T> haystack, T needle) {
    assertTrue(haystack.contains(needle), String.format("Expected %s to contain %s", haystack,
  }

  private static <T> void refuteContains (List<T> haystack, T needle) {
    refute(haystack.contains(needle), String.format("Expected %s not to contain %s", haystack,
  }

  private static void assertTrue (boolean thing, String message) {
    try {
      if (!thing) {
        throw new TestFailure(message);
      }
```

```java
    } catch (TestFailure e) {
      handleTestFailure(e);
    }
  }

  private static void refute (boolean thing, String message) {
    assertTrue(!thing, message);
  }
}

class TestFailure extends Exception {
  public TestFailure (String message) {
    super(message);
  }
}
```