Hackpack

Team #1 - Mars

Harrison Geiger, Ian Lewis and Kelton Williams

COP4516

Spring 2018

Table of Contents

**Table of Contents1**

**Permutation and Combination Generation2**

**Math Functions2**

Greatest Common Denominator (GCD)2

Lowest Common Multiple (LCM)2

**Graphing Algorithms3**

Depth First Search (DFS)4

Breadth First Search (BFS)4

Dijkstra’s Algorithm4

Bellman Ford’s Algorithm5

Floyd-Warshall’s Algorithm5

Topological Sort6

Network Flow (Ford Fulkerson)7

Kruskal’s Algorithm7 – 9

**Dynamic Programming Algorithms10**

Matrix Chain Multiplication DP10

Longest Common Sequence DP10

Knapsack DP11

**Geometry Algorithms11 - 12**

Line-Line Intersection12

Line-Plane Intersection12

Polygon Area13

Point in Polygon13

Convex Hull (Graham Scan)13

**Additional Helpful Code14**

Collections Notes14 - 15

Data Input14

End Case15

Binary Search16

Permutation and Combination Generation

1. public static void main(String[] args) {
2. int[] values = {
3. 1, 2, 3, 4
4. };
5. int[] perm = new int[values.length];
6. boolean[] used = new boolean[values.length]; // Also consider: Set<T>
7. perms(values, perm, used, 0);
8. }
9. public static void perms(int[] values, int[] perm, boolean[] used, int depth) {
10. if (depth == values.length) { // Process a permutation
11. System.out.println(java.util.Arrays.toString(perm));
12. return;
13. }
14. for (int i = 0; i < values.length; i++) { // Iterate over unused elements
15. if (!used[i]) { // for combinations, omit. For derrangements, check i != depth
16. used[i] = true; // Mark as used
17. perm[depth] = values[i]; // Use the element
18. perms(values, perm, used, depth + 1); // Recurse one level deeper
19. used[i] = false; } } // Unmark as used
20. }

Math Functions

1. public static void main(String[] args) {
2. System.out.println(gcd(960, 360)); // 120
3. System.out.println(lcm(134, 285));  // 38190
4. }

Greatest Common Denominator (GCD)

1. public static int gcd(int a, int b) {
2. if (b == 0) return a;
3. return gcd(b, a % b);
4. }
5. public static int gcd2(int a, int b) {
6. int temp;
7. while (b != 0) {
8. temp = a;
9. a = b;
10. b = temp % b;
11. }
12. return a;
13. }

Least Common Multiple (LCM)

1. public static int lcm(int a, int b) {
2. return a \* b / gcd(a, b);  // Beware of overflow
3. }

Graphing Algorithms

1. import java.util.\*;
2. import java.lang.Math;
3. class node {
4. public int data;
5. public Set < node > adj; // Set of adjacent nodes
6. public Map < node, Integer > adjW; // Map: adjacent node > weight of edge to that node
7. public node(int \_data) {
8. data = \_data;
9. adj = new HashSet < node > ();
10. adjW = new HashMap < node, Integer > ();
11. }
12. }
13. class edge implements Comparable < edge > {
14. public node target;
15. public int weight;
16. public edge(node \_target, int \_weight) {
17. target = \_target;
18. weight = \_weight;
19. }
20. public int compareTo(edge o) { // Smaller weights first
21. return weight - o.weight;
22. }
23. } // Also consider Integer.compare
24. public class Graph {
25. public static int oo = (int) 1e9; // Infinity (∞) placeholder
26. public static void main(String[] args) {
27. node[] graph = new node[5];
28. for (int i = 0; i < graph.length; i++) graph[i] = new node(i);
29. graph[0].adj.add(graph[1]);
30. graph[1].adj.add(graph[2]);
31. graph[0].adj.add(graph[3]);
32. graph[0].adj.add(graph[4]);
33. System.out.print("DFS: ");
34. dfs(graph[0]);
35. System.out.println(); // 3 4 2 1 0
36. System.out.print("BFS: ");
37. bfs(graph[0]);
38. System.out.println(); // 0 3 4 1 2
39. System.out.println("Dijkstra's: " + dijkstras(graph[0], graph[2])); // 2
40. System.out.println("Bellman-Ford");
41. bellmanFord(graph, graph[0]);
42. System.out.println("Floyd-Warshall");
43. floydWarshall(graph);
44. System.out.print("Topo Sort: ");
45. List < node > topo = topoSort(graph);
46. for (node n: topo) System.out.format("%d ", n.data);
47. System.out.println();
48. System.out.println("Ford Fulkerson: " + fordFulkerson(graph, graph[0], graph[2]));
49. }

Depth First Search (DFS)

1. public static void dfs(node currNode) {
2. dfs(currNode, new HashSet < node > ());
3. }
4. public static void dfs(node currNode, Set < node > visited) {
5. visited.add(currNode); // Mark start node visited
6. for (node adjNode: currNode.adj)
7. if (!visited.contains(adjNode))
8. dfs(adjNode, visited); // Recurse on all adjacent unvisited nodes
9. System.out.print(currNode.data + " ");
10. } // Finally, process node

Breadth First Search (BFS)

1. public static void bfs(node startNode) {
2. Queue<node> queue = new ArrayDeque<>();//Or ArrayList/LinkedList/PriorityQueue
3. Set <node> visited = new HashSet < > (); //Or node.visited or boolean[] visited
4. queue.offer(startNode);
5. while (!queue.isEmpty()) {
6. node currNode = queue.poll(); // Poll next node in queue
7. System.out.print(currNode.data + " "); // Process node
8. for (node adjNode: currNode.adj)
9. if (!visited.contains(adjNode)) {
10. visited.add(adjNode);
11. queue.offer(adjNode);  } } }

Dijkstra’s Algorithm

1. public static int dijkstras(node source,node dest) {//O(|E|+|V|log|V|) Dist from source node to dest node
2. Set <node> visited = new HashSet < > ();  // No negative edges
3. PriorityQueue<edge> pq = new PriorityQueue<>();//PriorityQueue improves runtime
4. pq.offer(new edge(source, 0));
5. while (!pq.isEmpty()) {
6. edge e = pq.poll(); // Poll for next edge of smallest cost
7. if (visited.contains(e.target)) continue; // Skip edges to already visited nodes
8. visited.add(e.target); // Visit node
9. if (e.target == dest) return e.weight; // Stop if we found the target node
10. for (node adjNode: e.target.adj) // Enqueue edges to unvisited nodes, add to weight
11. if (!visited.contains(adjNode))
12. pq.add(new edge(adjNode, e.weight + e.target.adjW.getOrDefault(adjNode, 1)));
13. }
14. return oo;
15. } // Destination not found, return infinite cost (impossible)

Bellman Ford’s Algorithm

1. //O(|V|\*|E|) Dist from source node to all nodes (no neg. cycles)
2. public static void bellmanFord(node[] graph, node source) {
3. Map<node,Integer> dist= new HashMap<node,Integer>(); // Distance from source node to given node
4. Map<node,node> pred=new HashMap<node,node>(); // Predecessor nodes in the path to a dest. node
5. for (node n: graph) dist.put(n, oo);
6. dist.put(source, 0); // Initialize distances to infinity, except source (0)
7. for (int i = 0; i < graph.length - 1; i++) {
8. boolean stop = true;
9. for (node u: graph)
10. for (node v: u.adj) { // Iterate over all edges
11. int w = u.adjW.getOrDefault(v, 1);
12. if (dist.get(u) + w < dist.get(v)) {
13. stop = false;
14. dist.put(v, dist.get(u) + w);
15. pred.put(v, u);
16. }
17. }
18. if (stop) break;
19. }
20. } // Optimisation: Stop early if no change was made

Floyd-Warshall’s Algorithm

1. public static void floydWarshall(node[] graph){ // O(|V|^3) (SLOW!) Dist from all nodes to all node
2. Map < node, Map < node, Integer >> dist = new HashMap < node, Map < node, Integer >> ();
3. Map < node, Map < node, node >> next = new HashMap < node, Map < node, node >> ();
4. for (node n: graph) {
5. Map < node, Integer > map = new HashMap < node, Integer > ();
6. dist.put(n, map);
7. for (node n2: graph) map.put(n2, oo); // Initialize distances to infinity
8. next.put(n, new HashMap < node, node > ());
9. } // Initialize next pointers to null
10. for (node n: graph) // Initialize edges
11. for (node adjNode: n.adj) {
12. dist.get(n).put(adjNode, n.adjW.getOrDefault(adjNode, 1));
13. next.get(n).put(adjNode, adjNode);
14. }
15. for (node k: graph) // Floyd-warshall DP - O(n^3)
16. for (node i: graph)
17. for (node j: graph) {
18. int sum = dist.get(i).get(k) + dist.get(k).get(j);
19. if (sum < dist.get(i).get(j)) { // Update if shorter path found
20. dist.get(i).put(j, sum);
21. next.get(i).put(j, next.get(i).get(k));
22. }
23. }
24. }

Topological Sort

1. public static List<node> topoSort(node[] graph) {//Node ordering that does not violate edge directions
2. Map < node, Integer > numIncEdges = new HashMap < node, Integer > ();
3. for (node n: graph) { // Count incoming edges for each node
4. numIncEdges.put(n, numIncEdges.getOrDefault(n, 0));
5. for (node adj: n.adj)
6. numIncEdges.put(adj, numIncEdges.getOrDefault(adj, 0) + 1);
7. }
8. Queue < node > queue = new ArrayDeque < node > (); // Also consider PriorityQueue
9. for (node n: numIncEdges.keySet()) // Initialize with nodes w/ 0 incoming edges
10. if (numIncEdges.get(n) == 0) queue.offer(n);
11. List < node > topoSort = new ArrayList < node > ();
12. while (!queue.isEmpty()) { // Process nodes until we have none left
13. node n = queue.poll(); // Choose next node with 0 incoming edges
14. topoSort.add(n); // Process this node (add to topo sort)
15. for (node adj: n.adj) { // Remove edges - decrement incoming edge count
16. numIncEdges.put(adj, numIncEdges.get(adj) - 1);
17. if (numIncEdges.get(adj) == 0) queue.offer(adj);
18. }
19. } // No more edges - enqueue
20. return topoSort.size() == graph.length ? topoSort : null;
21. } // If all nodes processed, we have valid toposort

Network Flow (Ford Fulkerson)

1. public static int fordFulkerson(node[] graph, node source, node sink) {
2. Set < node > visited = new HashSet < node > ();
3. Map<node,Map <node,Integer>> cap = new HashMap<>(); // Initialize cap with edge weights (max flow)
4. for (node n: graph) {
5. cap.put(n, new HashMap < node, Integer > ());
6. for (node adj: n.adj)
7. cap.get(n).put(adj, n.adjW.getOrDefault(adj, 1));
8. }
9. int flow = 0;
10. while (true) {
11. visited.clear();
12. int res = fulkDFS(cap, source, sink, visited, oo);
13. if (res == 0) break;
14. flow += res;
15. }
16. return flow;
17. }
18. public static int fulkDFS(Map<node,Map<node,Integer>> cap,node node,node sink,Set<node> visited,int min)
19. {
20. if (node == sink) return min;
21. if (visited.contains(node)) return 0;
22. visited.add(node);
23. int flow = 0;
24. for (node adj: node.adj) {
25. flow = fulkDFS(cap, adj, sink, visited, Math.min(cap.get(node).getOrDefault(adj, 0), min));
26. if (flow > 0) {
27. cap.get(node).put(adj, cap.get(node).getOrDefault(adj, 0) - flow); // cap[v][i] -= flow
28. cap.get(adj).put(node, cap.get(adj).getOrDefault(node, 0) + flow); // cap[i][v] += flow
29. return flow;
30. }
31. }
32. return 0;
33. }

Kruskal’s Algorithm (Team Mar’s Version)

NOTE: This is our team’s implementation of Kruskal’s. However, we will be using Columbia University’s implementation for the final exam. Their implementation is listed immediately below.

1. public static int kruskals(int size, edge[] graph) {
2. int totalWeight = 0;
3. int edges = 0;
4. ArrayList < edge > mst = new ArrayList < > ();
5. Arrays.sort(graph); // Sort the edge weights in increasing order.
6. for (int i = 0; i < size; i++) { // Check every edge to see if they are selected.
7. if (!mst.contains(graph[i])) { // If the given edge is not selected,
8. mst.add(graph[i]);
9. edges++; // add it to the mst,
10. totalWeight += graph[i].weight; // and tally their weight.
11. if (edges == (graph.length - 1)) return -1;
12. }
13. } // If we ever breach our size limit, we no longer have an MST. Signify with a -1 "weight".
14. return totalWeight;
15. } // Report cost of all selected edge weights.
16. }

Kruskal’s Algorithm (Columbia University’s Version)

NOTE: Sourced from: <http://www.cs.columbia.edu/~gskc/Code/AdvancedInternetServices/MinimalSpanningTree/Kruskal.java>

1. import java.io.\*;
2. import java.util.\*;
3. public class Kruskal {
4. private final int MAX\_NODES = 21;
5. private HashSet nodes[]; // Array of connected components
6. private TreeSet allEdges; // Priority queue of Edge objects
7. private Vector allNewEdges; // Edges in Minimal-Spanning Tree
8. public static void main(String args[]) {
9. System.out.println("Running [Kruskal] - 2000");
10. if (args.length != 1) {
11. System.out.println("Usage: java Kruskal <fileName>");
12. System.exit(0);
13. }
14. Kruskal k = new Kruskal();
15. k.readInGraphData(args[0]);
16. k.performKruskal();
17. k.printFinalEdges();
18. }
19. Kruskal() {
20. // Constructor
21. nodes = new HashSet[MAX\_NODES]; // Create array for components
22. allEdges = new TreeSet(new Edge()); // Create empty priority queue
23. allNewEdges = new Vector(MAX\_NODES); // Create vector for MST edges
24. }
25. private void readInGraphData(String fileName) {
26. try {
27. FileReader file = new FileReader(fileName);
28. BufferedReader buff = new BufferedReader(file);
29. String line = buff.readLine();
30. while (line != null) {
31. StringTokenizer tok = new StringTokenizer(line, " ");
32. int from = Integer.valueOf(tok.nextToken()).intValue();
33. int to = Integer.valueOf(tok.nextToken()).intValue();
34. int cost = Integer.valueOf(tok.nextToken()).intValue();
36. allEdges.add(new Edge(from, to, cost)); // Update priority queue
37. if (nodes[from] == null) {
38. // Create set of connect components [singleton] for this node
39. nodes[from] = new HashSet(2 \* MAX\_NODES);
40. nodes[from].add(new Integer(from));
41. }
43. if (nodes[to] == null) {
44. // Create set of connect components [singleton] for this node
45. nodes[to] = new HashSet(2 \* MAX\_NODES);
46. nodes[to].add(new Integer(to));
47. }
49. line = buff.readLine();
50. }
51. buff.close();
52. } catch (IOException e) {
53. //
54. }
55. }
56. private void performKruskal() {
57. int size = allEdges.size();
58. for (int i = 0; i < size; i++) {
59. Edge curEdge = (Edge) allEdges.first();
60. if (allEdges.remove(curEdge)) {
61. // successful removal from priority queue: allEdges
63. if (nodesAreInDifferentSets(curEdge.from, curEdge.to)) {
64. // System.out.println("Nodes are in different sets ...");
65. HashSet src, dst;
66. int dstHashSetIndex;
68. if (nodes[curEdge.from].size() > nodes[curEdge.to].size()) {
69. // have to transfer all nodes including curEdge.to
70. src = nodes[curEdge.to];
71. dst = nodes[dstHashSetIndex = curEdge.from];
72. } else {
73. // have to transfer all nodes including curEdge.from
74. src = nodes[curEdge.from];
75. dst = nodes[dstHashSetIndex = curEdge.to];
76. }
78. Object srcArray[] = src.toArray();
79. int transferSize = srcArray.length;
80. for (int j = 0; j < transferSize; j++) {
81. // move each node from set: src into set: dst
82. // and update appropriate index in array: nodes
83. if (src.remove(srcArray[j])) {
84. dst.add(srcArray[j]);
85. nodes[((Integer) srcArray[j]).intValue()] = nodes[dstHashSetIndex];
86. } else {
87. // This is a serious problem
88. System.out.println("Something wrong: set union");
89. System.exit(1);
90. }
91. }
93. allNewEdges.add(curEdge);
94. // add new edge to MST edge vector
95. } else {
96. // System.out.println("Nodes are in the same set ... nothing to do here");
97. }
98. } else {
99. // This is a serious problem
100. System.out.println("TreeSet should have contained this element!!");
101. System.exit(1);
102. }
103. }
104. }
105. private boolean nodesAreInDifferentSets(int a, int b) {
106. // returns true if graph nodes (a,b) are in different
107. // connected components, ie the set for 'a' is different
108. // from that for 'b'
109. return (!nodes[a].equals(nodes[b]));
110. }
111. private void printFinalEdges() {
112. System.out.println("The minimal spanning tree generated by " +
113. "\nKruskal's algorithm is: ");
114. while (!allNewEdges.isEmpty()) {
115. // for each edge in Vector of MST edges
116. Edge e = (Edge) allNewEdges.firstElement();
117. System.out.println("Nodes: (" + e.from + ", " + e.to +
118. ") with cost: " + e.cost);
119. allNewEdges.remove(e);
120. }
121. }
122. class Edge implements Comparator {
123. // Inner class for representing edge+end-points
124. public int from, to, cost;
125. public Edge() {
126. // Default constructor for TreeSet creation
127. }
128. public Edge(int f, int t, int c) {
129. // Inner class constructor
130. from = f;
131. to = t;
132. cost = c;
133. }
134. public int compare(Object o1, Object o2) {
135. // Used for comparisions during add/remove operations
136. int cost1 = ((Edge) o1).cost;
137. int cost2 = ((Edge) o2).cost;
138. int from1 = ((Edge) o1).from;
139. int from2 = ((Edge) o2).from;
140. int to1 = ((Edge) o1).to;
141. int to2 = ((Edge) o2).to;
143. if (cost1 < cost2)
144. return (-1);
145. else if (cost1 == cost2 && from1 == from2 && to1 == to2)
146. return (0);
147. else if (cost1 == cost2)
148. return (-1);
149. else if (cost1 > cost2)
150. return (1);
151. else
152. return (0);
153. }
154. public boolean equals(Object obj) {// Used for comparisions during operations
155. Edge e = (Edge) obj;
156. return (cost == e.cost && from == e.from && to == e.to);   } } }

Dynamic Programming Algorithms

1. public class DynamicProgramming {
2. public static void main(String[] args) {
3. int[] matrix = new int[] {
4. 300, 21, 42, 67, 198, 10  // 3812760
5. };
6. System.out.println(multiplication((matrix.length - 2), matrix));
7. System.out.println(lcs("TEST1ING2", "TEST233333ING1"));
8. int sackSize = 45, numItems = 10;
9. int[] weights = {
10. 7, 19, 8, 5, 14, 23, 1, 4, 9, 11
11. };
12. int[] values = {
13. 10, 50, 12, 7, 25, 63, 2, 5, 15, 20
14. };
15. System.out.println("Test: " + knapsack(numItems, weights, values, sackSize));
16. }

Matrix Chain Multiplication DP

1. public static int multiplication(int size, int matrix[]) {
2. int memo[][] = new int[matrix.length - 1][matrix.length - 1];
3. int tempMin;
4. for (int i = 0; i <= size; i++) memo[i][i] = 0; // Init all diagonals to 0
5. for (int i = (size - 1); i >= 0; i--)
6. for (int j = (i + 1); j <= size; j++) {
7. tempMin = Integer.MAX\_VALUE;
8. for (int k = i; k < j; k++){//Add our last memo'd values to the mult. Perm. of our matrix
9. int temp = ((memo[k + 1][j] + memo[i][k]) + matrix[i] \* matrix[k] \* matrix[j]);
10. if(temp<tempMin) tempMin = temp;//Check to see if that is smaller than the previous min
11. }
12. memo[i][j] = tempMin;
13. }
14. return memo[0][size];
15. }

Longest Common Subsequence DP

1. public static int lcs(String a, String b) {
2. int x = a.length();
3. int y = b.length();
4. int memo[][] = new int[x + 1][y + 1];
5. for (int i = 0; i <= x; i++)
6. for (int j = 0; j <= y; j++) {
7. memo[i][0] = 0;
8. memo[0][j] = 0;
9. }
10. for (int i = 1; i <= x; i++) {
11. for (int j = 1; j <= y; j++) {
12. if(a.charAt(i-1) != b.charAt(j-1)) memo[i][j] = Integer.max(memo[i][j-1], memo[i-1][j]);
13. else memo[i][j] = memo[i - 1][j - 1] + 1;
14. }
15. }
16. return memo[x][y];
17. }

Knapsack DP

1. public static int knapsack(int n, int[] w, int[] v, int size) {
2. int[][] memo = new int[n + 1][size + 1];
3. for (int i = 0; i <= size; i++) memo[0][i] = 0;
4. for (int i = 0; i <= n; i++) memo[i][0] = 0;
5. for (int i = 1; i <= n; i++)
6. for (int j = 1; j <= size; j++) {
7. if (j >= w[i-1]) memo[i][j] = Integer.max(memo[i-1][j],(memo[i-1][j - w[i-1]] + v[i-1]));
8. else memo[i][j] = memo[i - 1][j];
9. }
10. return memo[n][size];
11. }
12. }

Geometry Algorithms

1. import java.lang.Math;
2. import java.util.\*;
3. import java.awt.geom.Line2D;
4. import java.awt.geom.Path2D;
5. import java.awt.geom.Point2D;
6. class Vect {
7. public double x, y, z;
8. public Vect(double \_x, double \_y, double \_z) {
9. x = \_x;
10. y = \_y;
11. z = \_z;
12. }
13. public Vect add(Vect v) {
14. return new Vect(x + v.x, y + v.y, z + v.z);
15. }
16. public Vect subtract(Vect v) {
17. return new Vect(x - v.x, y - v.y, z - v.z);
18. }
19. public Vect scale(double mult) {
20. return new Vect(mult \* x, mult \* y, mult \* z);
21. }
22. public Vect crossProd(Vect v) {
23. return new Vect((y \* v.z) - (z \* v.y), (z - v.x) - (x \* v.z), (x - v.y) - (y \* v.x));
24. }
25. public double dotProd(Vect v) {
26. return (x \* v.x) + (y \* v.y) + (z \* v.z);
27. }
28. }
29. class pt {
30. public int x;
31. public int y;
32. public pt(int \_x, int \_y) {
33. x = \_x;
34. y = \_y;
35. }
36. }
37. public class Geometry {
38. public static final int CW = 1;
39. public static final int CCW = 2;
40. public static void main(String[] args) {
41. double p1x = 0 D, p1y = 0 D, p1z = 0 D, p2x = 0 D, p2y = 0 D, p2z = 0;
42. double p3x = 0 D, p3y = 0 D, p4x = 0 D, p4y = 0 D;
43. double circX = 0 D, circY = 0 D, circR = 0 D;
44. Line2D line1 = new Line2D.Double(p1x,p1y,p2x,p2y), line2 = new Line2D.Double(p3x,p3y,p4x,p4y);
45. Point2D circle = new Point2D.Double(circX, circY);
46. Vect vect1 = new Vect(p1x, p1y, p1z), vect2 = new Vect(p2x, p2y, p2z);
47. Vect plane1=new Vect(p1x,p1y,p1z), plane2=new Vect(p1x,p1y,p1z), plane3=new Vect(p1x,p1y,p1z);
48. double[] coordsX = {p1x, p2x, p3x}, coordsY = {p1y, p2y, p3y};
49. boolean res1 = intersectLineSegs(line1, line2);
50. boolean res2 = intersectLines(p1x, p1y, p2x, p2y, p3x, p3y, p4x, p4y);
51. boolean res3 = intersectLineSegCircle(line1, circle, circR);
52. boolean res4 = intersectLineCircle(line1, circle, circR);
53. boolean res5 = intersectLinePlane3(vect1, vect2, plane1, plane2, plane3);
54. double res6 = polygonArea(coordsX, coordsY);
55. boolean res7 = pointInPolygon(p1x, p1y, coordsX, coordsY);
56. pt[] pts = {new pt(1, 1), new pt(3, 1), new pt(2, 2), new pt(3, 3), new pt(1, 3)};
57. Set < pt > hull = grahamScan(pts);
58. for (pt p: hull) System.out.format("(%d, %d) ", p.x, p.y);
59. System.out.println();
60. }

Line-Line Intersection

1. // Returns whether two line segments intersect
2. public static boolean intersectLineSegs(Line2D line1, Line2D line2) {
3. return line1.intersectsLine(line2);
4. } // Returns whether two lines intersect
5. public static boolean intersectLines(double p1x,doublep1y,double p2x,double p2y,double p3x,double p3y,double p4x,double p4y) {
6. double line1Slope = (p2y - p1y) / (p2x - p1x);
7. double line2Slope = (p4y - p3y) / (p4x - p3x);
8. return line1Slope != line2Slope; }

Line-Plane Intersection

1. // Returns whether a line segment intersects a circle
2. public static boolean intersectLineSegCircle(Line2D line, Point2D circle, double r) {
3. return !(line.getP1().distance(circle) > r) || !(line.getP2().distance(circle) > r);
4. } // Returns whether a line intersects a circle
5. public static boolean intersectLineCircle(Line2D line, Point2D circle, double r) {
6. return line.getP1().distance(circle) < r && line.getP2().distance(circle) < r;
7. } // Returns whether a Vector and a line with 3 coordinates intersects
8. public static boolean intersectLinePlane3(Vect v1, Vect v2, Vect p1, Vect p2, Vect p3) {
9. Vect testV = (p2.subtract(p1).crossProd(p3.subtract(p1)));
10. if (Math.abs(testV.dotProd(v1.subtract(v2))) < 0.0000001) return false;
11. double scale = -(testV.dotProd(v1.subtract(p1)) / (testV.dotProd(v1.subtract(v2))));
12. Vect resV = v1.add(v1.subtract(v2).scale(scale));
13. double x = resV.subtract(p1).dotProd(p2.subtract(p1));
14. double y = resV.subtract(p1).dotProd(p3.subtract(p1));
15. return (x>=0D && x<=p2.subtract(p1).dotProd(p2.subtract(p1)) && y>=0D && y<=p3.subtract(p1).dotProd(p3.subtract(p1)));
16. }

Polygon Area

1. // Returns the area of a polygon
2. public static double polygonArea(double[] coordsX, double[] coordsY) {
3. int len = coordsX.length;
4. double res = 0 D;
5. for (int i = 0; i < len; i++) {
6. int k = (i + 1) % len;
7. res += coordsX[i] \* coordsY[k];
8. res -= coordsY[i] \* coordsX[k];
9. }
10. res /= 2;
11. res = Math.abs(res);

        return res; }

Point in Polygon

1. // Returns whether a point is in a polygon or not
2. public static boolean pointInPolygon(double pX,double pY, double[] coordsX, double[] coordsY) {
3. Path2D polygon = new Path2D.Double();
4. polygon.moveTo(coordsX[0], coordsY[0]);
5. for (int i = 1; i < coordsX.length; i++) {
6. polygon.lineTo(coordsX[i], coordsY[i]);
7. }
8. return polygon.contains(pX, pY);  }

Convex Hull (Graham Scan)

1. public static int orientation(pt p, pt q, pt r) { // Check orientation of three points
2. int val = (q.y - p.y) \* (r.x - q.x) - (q.x - p.x) \* (r.y - q.y);
3. if (val == 0) return 0; // Collinear
4. return (val > 0) ? CW : CCW;
5. }
6. public static Set < pt > grahamScan(pt[] pts) {
7. Set < pt > hull = new HashSet < pt > (); // Set of points in hull
8. int l = 0; // Select leftmost point to start with
9. for (int i = 0; i < pts.length; i++) l = (pts[i].x < pts[l].x) ? i : l;
10. int p = l, q;
11. do {
12. hull.add(pts[p]);
13. q = (p + 1) % pts.length; // Advance to next point in CW order
14. for (int i = 0; i < pts.length; i++)
15. if (orientation(pts[p], pts[i], pts[q]) == CCW) q = i;
16. p = q;
17. } while (p != l);
18. return hull;
19. } }

Additional Helpful Code

Collections Notes

1. import java.util.\*; // Everything is in java.util package
2. public class CollectionsNotes {
3. public static person john = new person("John", "Doe", 21);
4. public static person jane = new person("Jane", "Doe", 21);
5. public static void main(String[] args) {
6. lists();
7. sets();
8. maps();
9. }
10. public static void lists() {
11. int[] scores = {
12. 1, 2, 3, 42, 69, 1337, 4516, 2018
13. }; // Define + initialize array
14. List < Integer > scoreList = new ArrayList < > (); // Construct ArrayList
15. List < person > ppl = new ArrayList < person > ();
16. for (int score: scores) scoreList.add(score); // Add elements
17. ppl.add(john);
18. ppl.add(jane);
19. ppl.add(new person("Matt", "Smith", 22));
20. ppl.add(new person("Josh", "Garrett", 21));
21. Collections.shuffle(ppl); // Shuffle collection
22. Collections.reverse(ppl); // Reverse collection
23. Collections.sort(ppl); // Sort collection (low to high)
24. scoreList.sort(new Comparator < Integer > () { // Custom inline sort
25. public int compare(Integer a, Integer b) {
26. return a % 2 - b % 2;
27. }
28. }); // Even before odd
29. System.out.println(Arrays.toString(scoreList.toArray()));
30. System.out.println(Arrays.toString(ppl.toArray())); // Convert List to array, print array
31. for (person dude: ppl) System.out.println(dude.fname); // For-each
32. Iterator < person > it = ppl.iterator(); // Get iterator object
33. while (it.hasNext()) { // Are there more objects to iterate?
34. String lname = it.next().lname; // Get one object
35. if (lname.charAt(0) == 'D') it.remove();
36. } // Remove last object from .next();
37. int idx = Arrays.binarySearch(scores, 42);
38. } // Get index of needle
39. public static void sets() {
40. Set < person > ppl = new HashSet < person > (); // Construct HashSet
41. Set < person > ppl2 = new TreeSet<person>();//Construct TreeSet (<T implements Comparable<T>)
42. ppl.add(john); // Add element
43. if (ppl.contains(john)) // Query element
44. ppl.remove(john);
45. } // Remove element
46. public static void maps() {
47. Map < person, Integer > scores = new HashMap < person, Integer > (); // Construct HashMap (O(1), uses hashing)
48. Map<person,Integer> scores2 = new TreeMap<person,Integer>();//Construct TreeMap (O(log n), <K implements Comparable<T>)
49. scores.put(john, 1337); // Set key -> value
50. if (scores.containsKey(john)) { // Query
51. scores.put(john, scores.get(john) + 10); // Increment (Retrieval, Set)
52. scores.remove(john);
53. } // Removal
54. scores.getOrDefault(jane, 0); // Retrieval with default
55. scores.putIfAbsent(jane, 0); // Set if null
56. for (person dude: scores.keySet()) // Iterate over keys
57. System.out.println(dude.fname + ": " + scores.get(dude));
58. scores.clear();
59. }
60. } // Remove all elements
61. class person implements Comparable < person > {
62. public String fname;
63. public String lname;
64. public int age;
65. public person(String \_fname, String \_lname, int \_age) {
66. fname = \_fname;
67. lname = \_lname;
68. age = \_age;
69. }
70. public String toString() { // Override toString to make debugging easier
71. return fname + " " + lname + " (" + age + ")";
72. }
73. public int compareTo(person other) { // x<0: this<other | 0: this=other | x>0: this>other
74. if (age != other.age) return Integer.compare(age, other.age); // Integer, Double, Long
75. if (!lname.equals(other.lname)) return lname.compareTo(other.lname); // For String, use .compareTo
76. return fname.compareTo(other.fname);
77. }
78. }

Data Input

1. import java.util.\*;
2. public class NCases {
3. public static void main(String[] args) {
4. Scanner sc = new Scanner(System.in);
5. int n = sc.nextInt();
6. for (int i = 0; i < n; i++) { // Solution code goes here
7. }
8. }
9. }

End Case

1. import java.util.\*;
2. public class EndCondition {
3. public static void main(String[] args) {
4. Scanner sc = new Scanner(System.in);
5. while (sc.hasNext()) {
6. int x = sc.nextInt(), y = sc.nextInt(), z = sc.nextInt();
7. if ((x + y + z) == 0) break;
8. }
9. }
10. }

Binary Search

1. public class BinarySearch {
2. public static void main(String[] args) {
3. double value = 0 D, min = 0 D, max = 0 D;
4. double res1 = binarySearch(value, min, max);
5. }
6. public static double binarySearch(double value, double min, double max) {
7. double checkValue = (min + max) / 2;
8. if (checkValue > value) return binarySearch(value, min, checkValue);
9. else if (checkValue < value) return binarySearch(value, checkValue, max);
10. return value;
11. }
12. }