Hack Pack

Team 10 - Granola

Table of contents

1. Bruteforce
2. Campout
3. CountSeq
4. DFS\_BFS
5. Dijkstras
6. Dinic
7. DJset
8. EEA
9. Election
10. Floyd
11. FordFulk
12. Framing
13. Knapsack
14. Krus
15. LCS
16. LIS
17. Math
18. MCMDP
19. MCM
20. Peg
21. Poly
22. Prims
23. TestArray
24. Avl
25. Bellmanford

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200 | // Arup Guha  // 9/18/2015  // Examples for Brute Force Lecture for Programming Team  // http://www.cs.ucf.edu/~dmarino/progcontests/devteam-15/bruteforce.c    #include <stdio.h>    #define ODOMETER 0  #define PERMS 0  #define COMBOS 0  #define DERANGEMENTS 1  #define UPWARDS 0    // Recursive functions.  void printOdometer(int odometer[], int k, int n);  void printPerms(int perm[], int used[], int k, int n);  void printCombos(int subset[], int k, int n);  void printDerangements(int perm[], int used[], int k, int n);  void printUpWards(char word[], int skip, int k, int n);    // Prints our results.  void print(int array[], int n);  void printSubsets(int subset[], int n);    // Wrapper functions that run the recursive functions.  void runOdometer();  void runPerms();  void runCombos();  void runDerangements();  void runUpWards();    int main() {        // Run what you want...      if (ODOMETER) runOdometer();      if (PERMS) runPerms();      if (COMBOS) runCombos();      if (DERANGEMENTS) runDerangements();      if (UPWARDS) runUpWards();        return 0;  }    // Runs a 4 digit odometer.  void runOdometer() {      int meter[4];      printOdometer(meter, 0, 4);      printf("\n");  }    // Prints all possible seetings of odometer with n digits with the first k fixed.  void printOdometer(int odometer[], int k, int n) {        // Base case.      if (k == n) print(odometer, n);        // Fill in each possible digit in slot k and recurse.      else {          int i;          for (i=0; i<10; i++) {              odometer[k] = i;              printOdometer(odometer, k+1, n);          }      }  }    // Prints array[0..n-1].  void print(int array[], int n) {      int i;      for (i=0; i<n; i++)          printf("%d ", array[i]);      printf("\n");  }    // Prints out all permutations of 0,1,2,3.  void runPerms() {      int perm[4];      int i, used[4];      for (i=0; i<4; i++) used[i] = 0;      printPerms(perm, used, 0, 4);      printf("\n");  }    // Prints all permutations of 0,1,...,n-1 where the first k items of perm are fixed.  void printPerms(int perm[], int used[], int k, int n) {        // Base case.      if (k == n) print(perm, n);        // Recursive case - fill in slot k.      else {          int i;            // Only fill slot k with items that have yet to be used. If i hasn't been used,          // put it in slot k and recursively print all permutations with these k+1 starting items.          for (i=0; i<n; i++) {              if (!used[i]) {                  used[i] = 1;                  perm[k] = i;                  printPerms(perm, used, k+1, n);                  used[i] = 0;              }          }      }  }    // Prints all combinations of 0,1,2,3,4.  void runCombos() {      int i, items[5];      for (i=0; i<5; i++) items[i] = 0;      printCombos(items, 0, 5);      printf("\n");  }    void printCombos(int subset[], int k, int n) {        // Base case, subset filled in.      if (k == n) printSubsets(subset, n);        // Recursive case - fill slot k.      else {            // First do subset without item k.          printCombos(subset, k+1, n);            // Now do the subset with item k. Must return subset to original setting!!!          subset[k] = 1;          printCombos(subset, k+1, n);          subset[k] = 0;      }  }    // Prints out the subset of 0,1,2..,n-1 represented by subset. subset[i] is 1 iff i is in the subset.  void printSubsets(int subset[], int n) {      int i;      for (i=0; i<n; i++)          if (subset[i])              printf("%d ", i);      printf("\n");  }    // Prints out all derangements of 0,1,2,3,4.  void runDerangements() {      int perm[4];      int i, used[4];      for (i=0; i<4; i++) used[i] = 0;      printDerangements(perm, used, 0, 4);      printf("\n");  }    // Prints out all derangements in perm with the first k digits fixed.  void printDerangements(int perm[], int used[], int k, int n) {       // Base case.      if (k == n) print(perm, n);        // Recursive case - fill in slot k.      else {          int i;            // Same as permutation, but we don't allow slot k to be filled with k.          for (i=0; i<n; i++) {              if (!used[i] && i != k) {                  used[i] = 1;                  perm[k] = i;                  printDerangements(perm, used, k+1, n);                  used[i] = 0;              }          }      }  }    // Prints all 5 length upwards with skip 2.  void runUpWards() {      char word[6];      word[5] = '\0';      printUpWards(word, 2, 0, 5);      printf("\n");  }    // Prints all skip-level upwords of length n.  void printUpWards(char word[], int skip, int k, int n) {        // Base case - word is filled in.      if (k == n) printf("%s\n", word);        // Try each possible item in slot k.      else {            // If we haven't filled anything yet, we start with 'a', otherwise go to the last character and          // count forward skip+1 places.          char start = k == 0 ? 'a' : word[k-1] + skip + 1;            char next;          for (next=start; next<='z'; next++) {              word[k] = next;              printUpWards(word, skip, k+1, n);          }      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161 | // Arup Guha  // 8/31 - 9/2/2011  // Solution to 2011 UCF Locals Contest Problem: Campout  // Edited on 3/10/2017 to use FordFulkerson code shown in class.    import java.io.\*;  import java.util.\*;    public class campout {        final public static int NUMSTUD = 10;      final public static int NUMSHIFTS = 42;      final public static int N = NUMSTUD+NUMSHIFTS;        public static void main(String[] arg) throws Exception {            Scanner fin = new Scanner(System.in);          int numCases = fin.nextInt();            // Go through each case.          for (int caseNum=1; caseNum<=numCases; caseNum++) {                // Create flow object, adding in capacities based on the problem (20 shifts max per student,              // 3 students for each shift minimum.)              FordFulkerson net = new FordFulkerson(N);              for (int i=0; i<NUMSTUD; i++) net.add(N,i,20);              for (int i=0; i<NUMSHIFTS; i++) net.add(NUMSTUD+i,N+1,3);                // Read in all 10 students' info.              for (int stud=0; stud<10; stud++) {                    // Initialize each slot so that this student can cover it.                  boolean[] cancover = new boolean[NUMSHIFTS];                  for (int i=0; i<NUMSHIFTS; i++)                      cancover[i] = true;                    int numConflicts = fin.nextInt();                    // Go through each conflict.                  for (int con=0; con<numConflicts; con++) {                        int d = fin.nextInt();                      int s = fin.nextInt();                      int e = fin.nextInt();                        // Each time maps to one of the time ranges, so we just go through these and make them false.                      for (int i=(d-1)\*6+s/4; i<(d-1)\*6+(e+3)/4; i++)                          cancover[i] = false;                  }                    // Fill in shifts student can do.                  for (int i=0; i<NUMSHIFTS; i++)                      if (cancover[i])                          net.add(stud, NUMSTUD+i, 1);              }                // Get the flow of this network.              int maxflow = net.ff();                System.out.print("Case #"+caseNum+": ");                // This means all of 42 shifts were covered by at least 3 students each.              if (maxflow == 126)                  System.out.println("YES");              else                  System.out.println("NO");                System.out.println();            }      }    }    class FordFulkerson {        // Stores graph.      public int[][] cap;      public int n;      public int source;      public int sink;        // "Infinite" flow.      final public static int oo = (int)(1E9);        // Set up default flow network with size+2 vertices, size is source, size+1 is sink.      public FordFulkerson(int size) {          n = size + 2;          source = n - 2;          sink = n - 1;          cap = new int[n][n];      }        // Adds an edge from v1 -> v2 with capacity c.      public void add(int v1, int v2, int c) {          cap[v1][v2] = c;      }        // Wrapper function for Ford-Fulkerson Algorithm      public int ff() {            // Set visited array and flow.          boolean[] visited = new boolean[n];          int flow = 0;            // Loop until no augmenting paths found.          while (true) {                // Run one DFS.              Arrays.fill(visited, false);              int res = dfs(source, visited, oo);                // Nothing found, get out.              if (res == 0) break;                // Add this flow.              flow += res;          }            // Return it.          return flow;      }        // DFS to find augmenting math from v with maxflow at most min.      public int dfs(int v, boolean[] visited, int min) {            // got to the sink, this is our flow.          if (v == sink)  return min;            // We've been here before - no flow.          if (visited[v])  return 0;            // Mark this node and recurse.          visited[v] = true;          int flow = 0;            // Just loop through all possible next nodes.          for (int i = 0; i < n; i++) {                // We can augment in this direction.              if (cap[v][i] > 0)                  flow = dfs(i, visited, Math.min(cap[v][i], min));                // We got positive flow on this recursive route, return it.              if (flow > 0) {                    // Subtract it going forward.                  cap[v][i] -= flow;                    // Add it going backwards, so that later, we can flow back through this edge as a backedge.                  cap[i][v] += flow;                    // Return this flow.                  return flow;              }          }            // If we get here there was no flow.          return 0;      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | // Arup Guha  // 3/14/2013  // Solution to COP 3503 Program #5A: Counting Subsequences    import java.util.\*;    public class countseq {        public static void main(String[] args) {            Scanner stdin = new Scanner(System.in);            int numCases = stdin.nextInt();            for (int loop=0; loop<numCases; loop++) {              String text = stdin.next();              String pattern = stdin.next();              System.out.println(solve(text, pattern));          }      }        public static long solve(String text, String pattern) {            long[][] dp = new long[text.length()+1][pattern.length()+1];            // Initialize table - be CAREFUL here (Note 0s and 1s!!!)          for (int i=0; i<dp[0].length; i++)              dp[0][i] = 0;          for (int i=0; i<dp.length; i++)              dp[i][0] = 1;            // Solve each subcase in order.          for (int i=1; i<dp.length; i++) {              for (int j=1; j<dp[i].length; j++) {                    // These last characters match.                  if (text.charAt(i-1) == pattern.charAt(j-1))                      dp[i][j] = dp[i-1][j] + dp[i-1][j-1];                    // Just copy over old matches, since this character doesn't help.                  else                      dp[i][j] = dp[i-1][j];              }          }            /\*\*\* JUST TO VERIFY INPUT \*\*\*/          if (hasNeg(dp))              System.out.println("There is overflow in this case.");            // Answer is stored here.          return dp[dp.length-1][dp[0].length-1];      }        /\*\*\* Used to verify input. \*\*\*/      public static boolean hasNeg(long[][] array) {            for (int i=0; i<array.length; i++)              for (int j=0; j<array[0].length; j++)                  if (array[i][j] < 0)                      return true;          return false;      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | // Arup Guha  // 9/11/2014  // Code to illustrate a cumulative frequency array.    import java.util.\*;    public class cumfreq {        final public static int SIZE = 10;      final public static int MAX = 20;        public static void main(String[] args) {            Random r = new Random();          int[] orig = mkArray(r, SIZE, MAX+1);          printArray(orig);            // Forming cumulative frequency array.          int[] cumfreq = new int[SIZE];          cumfreq[0] = orig[0];          for (int i=1; i<SIZE; i++)              cumfreq[i] = cumfreq[i-1] + orig[i];            // An example - sum index 2 to 7, inclusive.          System.out.println("Sum from index 2 to index 7: "+(cumfreq[7]-cumfreq[1]));            int sum = 0;          for (int i=2; i<=7; i++)              sum += orig[i];          System.out.println("Long sum: "+sum);            // Sorted List Matching.          int[] arr1 = mkArray(r, SIZE, MAX+1);          int[] arr2 = mkArray(r, SIZE, MAX+1);          Arrays.sort(arr1);          Arrays.sort(arr2);          System.out.println("Arrays for sorted list matching.");          printArray(arr1);          printArray(arr2);            // Sorted List Matching - prints out common names.          System.out.print("Numbers on both lists: ");          int i = 0, j = 0;          while (i < arr1.length && j < arr2.length) {                // Advance appropriate pointer with non-match.              if (arr1[i] < arr2[j]) i++;              else if (arr2[j] < arr1[i]) j++;                // Process a match.              else {                  System.out.print(arr1[i]+" ");                  i++;                  j++;              }          }          System.out.println();        }        // Fills an array with size random values in between 0 and maxval-1,      // inclusive.      public static int[] mkArray(Random r, int size, int maxval) {          int[] arr = new int[size];          for (int i=0; i<size; i++)              arr[i] = r.nextInt(maxval);          return arr;      }        // Prints Array.      public static void printArray(int[] arr) {          for (int i=0; i<arr.length; i++)              System.out.print(arr[i]+" ");          System.out.println();      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145 | // Stephen Fulwider  //  A sample program to show working examples of Depth/Breadth First Search (DFS,BFS)    import java.util.LinkedList;  import java.util.Queue;    public class DFS\_BFS  {        public static void main(String[] args)      {          new DFS\_BFS(); // I like doing things this way...      }        int N; // number of vertices in the graph      boolean[][] G; // the graph as an adjacency matrix                     // G[i][j] is true if there is an edge from i to j        DFS\_BFS()      {          setupGraph();          System.out.println("------------------------------");          System.out.println();            // perform a DFS on the graph          DFS();          System.out.println();          System.out.println("------------------------------");          System.out.println();            // perform a BFS on the graph          BFS();          System.out.println();          System.out.println("------------------------------");          System.out.println();          System.out.println("All done - have a good day!");      }        // initial setup of the graph      void setupGraph()      {          // set up a graph with 8 vertices + 3 components that looks like:          /\*              0 --- 1        5---6              | \    \       |  /              |  \    \      | /              2   3----4     7          \*/          N=8;          G=new boolean[N][N];          // notice that for each edge G[i][j] == G[j][i]          // this means that the graph is undirected          G[0][1]=G[1][0]=true; G[0][2]=G[2][0]=true; G[0][3]=G[3][0]=true;          G[1][4]=G[4][1]=true; G[3][4]=G[4][3]=true; G[5][6]=G[6][5]=true;          G[5][7]=G[7][5]=true; G[6][7]=G[7][6]=true;      }        // perform a DFS on the graph G      void DFS()      {          // a visited array to mark visited vertices in DFS          boolean[] V=new boolean[N];          int numComponets=0; // the number of components in the graph            // do the DFS from each node not already visited          for (int i=0; i<N; ++i)              if (!V[i])              {                  ++numComponets;                  System.out.printf("DFS for component %d starting at node %d%n",numComponets,i);                  DFS(i,V);              }            System.out.println();          System.out.printf("Finished DFS - found %d components.%n", numComponets);      }        // perform a DFS starting at node at (works recursively)      void DFS(int at, boolean[] V)      {          System.out.printf("At node %d in the DFS%n",at);            // mark that we are visiting this node          V[at]=true;            // recursively visit every node connected yet to be visited          for (int i=0; i<N; ++i)              if (G[at][i] && !V[i])              {                  System.out.printf("Going to node %d...",i);                  DFS(i,V);              }          System.out.printf("Done processing node %d%n", at);      }        // perform a BFS on the graph G      void BFS()      {          // a visited array to mark which vertices have been visited in BFS          boolean[] V=new boolean[N];            int numComponets=0; // the number of components in the graph            // do the BFS from each node not already visited          for (int i=0; i<N; ++i)              if (!V[i])              {                  ++numComponets;                  System.out.printf("BFS for component %d starting at node %d%n",numComponets,i);                    BFS(i,V);              }          System.out.println();          System.out.printf("Finished BFS - found %d components.%n", numComponets);      }        // perform a BFS starting at node start      void BFS(int start, boolean[] V)      {          Queue<Integer> Q=new LinkedList<Integer>(); // A queue to process nodes            // start with only the start node in the queue and mark it as visited          Q.offer(start);          V[start]=true;            // continue searching the graph while still nodes in the queue          while (!Q.isEmpty())          {              int at=Q.poll(); // get the head of the queue              System.out.printf("At node %d in the BFS%n",at);                // add any unseen nodes to the queue to process, then mark them as              // visited so they don't get re-added              for (int i=0; i<N; ++i)                  if (G[at][i] && !V[i])                  {                      Q.offer(i);                      V[i]=true;                      System.out.printf("Adding node %d to the queue in the BFS%n", i);                  }              System.out.printf("Done processing node %d%n", at);          }          System.out.printf("Finished with the BFS from start node %d%n", start);      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | // Danny Wasserman  // 7/14/2014  // Implementation of Dijkstra's Algorithm with a Priority Queue.    import java.util.ArrayList;  import java.util.PriorityQueue;    public class dijkstras {      // Everyone has access to these.    static int oo = (int) 1e9;    static int n;    static ArrayList<Edge>[] g;      // Returns the shortest distance from vertex s to d.    public static int dijkstras(int s, int d) {        // Set up the priority queue.      boolean[] visited = new boolean[n];      PriorityQueue<Edge> pq = new PriorityQueue<Edge>();      pq.add(new Edge(s, 0));        // Go till empty.      while (!pq.isEmpty()) {          // Get the next edge.        Edge at = pq.poll();        if (visited[at.e]) continue;        visited[at.e] = true;          // We made it, return the distance.        if (at.e == d) return at.w;          // Enqueue all the neighboring edges.        for (Edge adj : g[at.e])          if (!visited[adj.e]) pq.add(new Edge(adj.e, adj.w + at.w));      }      return oo;    }      // Stores where an edge is going to and its weight.    static class Edge implements Comparable<Edge> {      int e, w;        public Edge(int e, int w) {        this.e = e;        this.w = w;      }        public int compareTo(Edge o) {        return w - o.w;      }    }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152 | // "UCF Programming Team" Hackpack Code  // Original Author(s) - Unknown  // Taken from Team Badlands Hackpack  // Commented and Edited by Arup Guha on 3/6/2017 for COP 4516  // Code for Dinic's Network Flow Algorithm    import java.util.\*;    // An edge connects v1 to v2 with a capacity of cap, flow of flow.  class Edge {      int v1, v2, cap, flow;      Edge rev;      Edge(int V1, int V2, int Cap, int Flow) {          v1 = V1;          v2 = V2;          cap = Cap;          flow = Flow;      }  }    public class Dinic {        // Queue for the top level BFS.      public ArrayDeque<Integer> q;        // Stores the graph.      public ArrayList<Edge>[] adj;      public int n;        // s = source, t = sink      public int s;      public int t;          // For BFS.      public boolean[] blocked;      public int[] dist;        final public static int oo = (int)1E9;        // Constructor.      public Dinic (int N) {            // s is the source, t is the sink, add these as last two nodes.          n = N; s = n++; t = n++;            // Everything else is empty.          blocked = new boolean[n];          dist = new int[n];          q = new ArrayDeque<Integer>();          adj = new ArrayList[n];          for(int i = 0; i < n; ++i)              adj[i] = new ArrayList<Edge>();      }        // Just adds an edge and ALSO adds it going backwards.      public void add(int v1, int v2, int cap, int flow) {          Edge e = new Edge(v1, v2, cap, flow);          Edge rev = new Edge(v2, v1, 0, 0);          adj[v1].add(rev.rev = e);          adj[v2].add(e.rev = rev);      }        // Runs other level BFS.      public boolean bfs() {            // Set up BFS          q.clear();          Arrays.fill(dist, -1);          dist[t] = 0;          q.add(t);            // Go backwards from sink looking for source.          // We just care to mark distances left to the sink.          while(!q.isEmpty()) {              int node = q.poll();              if(node == s)                  return true;              for(Edge e : adj[node]) {                  if(e.rev.cap > e.rev.flow && dist[e.v2] == -1) {                      dist[e.v2] = dist[node] + 1;                      q.add(e.v2);                  }              }          }            // Augmenting paths exist iff we made it back to the source.          return dist[s] != -1;      }        // Runs inner DFS in Dinic's, from node pos with a flow of min.      public int dfs(int pos, int min) {            // Made it to the sink, we're good, return this as our max flow for the augmenting path.          if(pos == t)              return min;          int flow = 0;            // Try each edge from here.          for(Edge e : adj[pos]) {              int cur = 0;                // If our destination isn't blocked and it's 1 closer to the sink and there's flow, we              // can go this way.              if(!blocked[e.v2] && dist[e.v2] == dist[pos]-1 && e.cap - e.flow > 0) {                    // Recursively run dfs from here - limiting flow based on current and what's left on this edge.                  cur = dfs(e.v2, Math.min(min-flow, e.cap - e.flow));                    // Add the flow through this edge and subtract it from the reverse flow.                  e.flow += cur;                  e.rev.flow = -e.flow;                    // Add to the total flow.                  flow += cur;              }                // No more can go through, we're good.              if(flow == min)                  return flow;          }            // mark if this node is now blocked.          blocked[pos] = flow != min;            // This is the flow          return flow;      }        public int flow() {          clear();          int ret = 0;            // Run a top level BFS.          while(bfs()) {                // Reset this.              Arrays.fill(blocked, false);                // Run multiple DFS's until there is no flow left to push through.              ret += dfs(s, oo);          }          return ret;      }        // Just resets flow through all edges to be 0.      public void clear() {          for(ArrayList<Edge> edges : adj)              for(Edge e : edges)                  e.flow = 0;      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | // Arup Guha  // 1/23/2018  // Disjoint Set Implementation with Path Compression  // Written in COP 3503 class    import java.util.\*;    public class djset {        final public static int N = 10;        public int[] parent;        // Creates a disjoint set of size n (0, 1, ..., n-1)      public djset(int n) {          parent = new int[n];          for (int i=0; i<n; i++)              parent[i] = i;      }        public int find(int v) {            // I am the club president!!! (root of the tree)          if (parent[v] == v) return v;            // Find my parent's root.          int res = find(parent[v]);            // Attach me directly to the root of my tree.          parent[v] = res;          return res;      }        public boolean union(int v1, int v2) {            // Find respective roots.          int rootv1 = find(v1);          int rootv2 = find(v2);            // No union done, v1, v2 already together.          if (rootv1 == rootv2) return false;            // Attach tree of v2 to tree of v1.          parent[rootv2] = rootv1;          return true;      }        // For testing.      public void print() {          for (int i=0; i<parent.length; i++)              System.out.print(parent[i]+" ");          System.out.println();      }        // Run a little test.      public static void main(String[] args) {            djset set = new djset(N);          Random r = new Random();          int numUnion = 0;            // Union N-1 times.          while (numUnion < N-1) {                // Make v1 and v2 two distinct values.              int v1 = r.nextInt(N);              int v2 = v1;              while (v2 == v1) v2 = r.nextInt(N);                System.out.println("Union of "+v1+" and "+v2);              boolean res = set.union(v1, v2);                // Already in the same tree/set.              if (!res)                  System.out.println("Already together");                // Merge them.              else {                  System.out.println("Merged the two trees.");                  numUnion++;              }                // Let's look at the set.              set.print();          }        }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | /\*\*   \* Returns an int array containing:   \* 0+1) x and y in ax + by = gcd(a,b)   \* 2) gcd(a,b)   \* Note: values at indexes 0 and 1 may be negative   \*   \* O(log(min(a,b)))   \*\*/     import java.util.\*;    public class EEA {          // Returns b inverse mod a in res[1]. Note - value returned could be negative.      // Only works if a and b are ints due to the multiplication in the code.      public static long[] extendedEuclideanAlgorithm(long a, long b) {          if (b==0)              return new long[]{1,0,a};          else {              long q = a/b;              long r = a%b;              long[] rec = extendedEuclideanAlgorithm(b,r);              return new long[]{rec[1], rec[0] - q\*rec[1], rec[2]};          }      }        public static long gcd(long a, long b) {          return b == 0 ? a : gcd(b, a%b);      }        public static void main(String[] args) {            Random r = new Random();            // We will test the method 1000 times.          for (int i=0; i<1000; i++) {                // Generate random ints.              long a = r.nextInt(2000000000);              long b = r.nextInt(2000000000);                // Until we get a gcd of 1.              while (gcd(a, b) != 1) {                  a = r.nextInt(2000000000);                  b = r.nextInt(2000000000);              }                // Make sure a > b.              if (a < b) {                  long tmp = a;                  a = b;                  b = tmp;              }                // Call it.              long[] res = extendedEuclideanAlgorithm(a, b);              long inverse = (res[1]+a)%a;                // Print out an error message if the inverse doesn't work.              if ((inverse\*b)%a != 1) {                  System.out.println("Failed on "+a+" and "+b+" got "+inverse);              }            }      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | // Arup Guha  // 1/16/2018  // Example that shows how to custom sort arrays and lists in java.    /\*\*\* Input format: Line 1: n, number of candidates   \*                 Lines 2-(n+1): one name followed by number of votes that person got.   \*   Output format: Sorted list of vote getters from most to least. If two   \*                  people get the same number of votes, break ties by alpha order.   \*\*\*/    import java.util.\*;    public class election1 {        public static void main(String[] args) {            Scanner stdin = new Scanner(System.in);          int numCandidates = stdin.nextInt();            // Read in all the candidates.          candidate[] array = new candidate[numCandidates];          ArrayList<candidate> list = new ArrayList<candidate>();          for (int i=0; i<numCandidates; i++) {              String name = stdin.next();              int numVotes = stdin.nextInt();              array[i] = new candidate(name, numVotes);              list.add(array[i]);          }            // How to sort an array.          Arrays.sort(array);            for (int i=0; i<array.length; i++)              System.out.println(array[i]);            System.out.println();            // Unsorted list.          for (int i=0; i<list.size(); i++)              System.out.println(list.get(i));            // How to sort a collection.          Collections.sort(list);            System.out.println();            // Unsorted list.          for (int i=0; i<list.size(); i++)              System.out.println(list.get(i));      }  }    class candidate implements Comparable<candidate> {        private String name;      private int numVotes;        public candidate(String n, int v) {          name = n;          numVotes = v;      }        public String toString() {          return name+" "+numVotes;      }        public int compareTo(candidate other) {            // We can break the tie by looking at votes, go most to least.          if (this.numVotes != other.numVotes)              return other.numVotes - this.numVotes;            // Otherwise, for same number of votes, we go in alpha order.          return this.name.compareTo(other.name);      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125 | // Stephen Fulwider  //  Floyd's all pairs shortest path algorithm  //  with path reconstruction  // Email knightry@gmail.com with questions    import java.util.LinkedList;    public class Floyd  {        public static void main(String[] args)      {          new Floyd(); // I do this so I don't have to use static variables everywhere      }        final int oo = (int) 1e9; // infinity!        int N; // number of nodes      int[][] G; // original graph in adj. matrix form        int[][] D; // computed distance between each pair of vertices      int[][] P; // predecessor matrix        Floyd()      {          // set up a graph - draw this out so you can do some testing!          //  notice this is a directed graph. this means an edge from a->b          //  doesn't imply an edge of the same weight from b->a          // oo denotes no edge, otherwise the number denotes the length          //  of the edge (negative edge weights are possible)          N = 5;          G = new int[][] {                  {0,3,8,oo,-4},                  {oo,0,oo,1,7},                  {oo,4,0,oo,oo},                  {oo,oo,-5,0,oo},                  {oo,oo,oo,6,0}          };              // run floyds - we only need to run this ONCE to get the shortest path          //  between ALL pairs of points!          floyds();              // Print out all the paths          for (int source=0; source<N; ++source)              for (int target=0; target<N; ++target)              {                  LinkedList<Integer> path = getPath(source,target);                  System.out.printf("Length of shortest path from %d to %d: %d%n",source,target,D[source][target]);                  if (path != null)                      System.out.printf("   Path: %s%n%n",path);                  else                      System.out.printf("   Path does not exist!%n%n");              }      }        // run floyds all pairs shortest path algorithm      //  this algorithm runs in O(N^3) time      void floyds()      {          // first set up the predecessor matrix          //  we will define P[i][j] to be the predecessor of node j          //  when traveling on the path from i->j.          // Example 1: If the path from 1 to 2 is the single edge 1->2,          //  then P[1][2] = 1          // Example 2: If the path from 3 to 4 is the path 3->4->5,          //  then P[3][5] = 4 and P[3][4] = 3          // We use -1 to denote no path from i to j            P = new int[N][N];          for (int i=0; i<N; ++i)              for (int j=0; j<N; ++j)              {                  if (G[i][j] < oo) // only consider edges which exist originally                      P[i][j] = i;                  else                      P[i][j] = -1;              }              // next make a copy of the original graph to do work on          //  notice that you only need to do this if you need to maintain the          //  original graph for some reason. Otherwise you can just overwrite G            D = new int[N][N];          for (int i=0; i<N; ++i)              for (int j=0; j<N; ++j)                  D[i][j] = G[i][j];              // now run the algorithm itself            for (int k=0; k<N; ++k)              for (int i=0; i<N; ++i)                  for (int j=0; j<N; ++j)                      if (D[i][j] > D[i][k] + D[k][j] && D[i][k] < oo && D[k][j] < oo)                      {                          // using node k helps, update the weight and the predecessor of i->j                          D[i][j] = D[i][k] + D[k][j];                          P[i][j] = P[k][j];                      }      }        // reconstruct the path in reverse (since we store the predecessor, it makes      //  sense that we would need to start and the end and work our way backwards)      LinkedList<Integer> getPath(int source, int target)      {          // first check if the path exists - if it doesn't return null          if (D[source][target] == oo)              return null;            // now we know the path exists, so reconstruct it          LinkedList<Integer> path = new LinkedList<Integer>();          path.addFirst(target);          while (target != source)          {              target = P[source][target];              path.addFirst(target);          }          return path;      }    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | // "UCF Programming Team" Hackpack Code  // Original Author(s) - Unknown  // Taken from Team Badlands Hackpack  // Commented and Edited by Arup Guha on 3/6/2017 for COP 4516  // Code for Ford Fulkerson Algorithm    import java.util.\*;    public class FordFulkerson {        // Stores graph.      public int[][] cap;      public int n;      public int source;      public int sink;        // "Infinite" flow.      final public static int oo = (int)(1E9);        // Set up default flow network with size+2 vertices, size is source, size+1 is sink.      public FordFulkerson(int size) {          n = size + 2;          source = n - 2;          sink = n - 1;          cap = new int[n][n];      }        // Adds an edge from v1 -> v2 with capacity c.      public void add(int v1, int v2, int c) {          cap[v1][v2] = c;      }        // Wrapper function for Ford-Fulkerson Algorithm      public int ff() {            // Set visited array and flow.          boolean[] visited = new boolean[n];          int flow = 0;            // Loop until no augmenting paths found.          while (true) {                // Run one DFS.              Arrays.fill(visited, false);              int res = dfs(source, visited, oo);                // Nothing found, get out.              if (res == 0) break;                // Add this flow.              flow += res;          }            // Return it.          return flow;      }        // DFS to find augmenting math from v with maxflow at most min.      public int dfs(int v, boolean[] visited, int min) {            // got to the sink, this is our flow.          if (v == sink)  return min;            // We've been here before - no flow.          if (visited[v])  return 0;            // Mark this node and recurse.          visited[v] = true;          int flow = 0;            // Just loop through all possible next nodes.          for (int i = 0; i < n; i++) {                // We can augment in this direction.              if (cap[v][i] > 0)                  flow = dfs(i, visited, Math.min(cap[v][i], min));                // We got positive flow on this recursive route, return it.              if (flow > 0) {                    // Subtract it going forward.                  cap[v][i] -= flow;                    // Add it going backwards, so that later, we can flow back through this edge as a backedge.                  cap[i][v] += flow;                    // Return this flow.                  return flow;              }          }            // If we get here there was no flow.          return 0;      }  } |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | import java.util.\*;    public class framing {        public static void main (String [] args)      {          Scanner in = new Scanner(System.in);          int numGroups = in.nextInt();          double v = 0;          double totalV = 0;              for (int loop = 0; loop < numGroups; loop++) {                int people = in.nextInt();                  for (int groupLoop = 0; groupLoop < people; groupLoop++) {                  int frames = in.nextInt();                      for(int i = 0; i < frames; i++) {                      double areaW = 0, areaL = 0;                      int l = in.nextInt();                      int w = in.nextInt();                      int d = in.nextInt();                      double width, length, height;                        width = w + d \* Math.sqrt(2);                      length = l + d \* Math.sqrt(2);                      height = d / Math.sqrt(2);                        v = ((width + w) \* height) + ((length + l) \* height);                                  //v \*= 100;                      v = Math.round(v);                      //v /= 100;                      totalV += v;                  }                  }            System.out.printf("Group #%d: %.2f cubic inches", (loop + 1), totalV );          System.out.println();              totalV = 0;            }          }   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | // Arup Guha  // 3/21/2017  // Shows DP approach to knapsack problem.    import java.util.\*;    public class knapsack {        public static void main(String[] args) {            // Read in a single case from input.          Scanner stdin = new Scanner(System.in);          int n = stdin.nextInt();          int[] weights = new int[n];          int[] values = new int[n];          for (int i=0; i<n; i++) {              weights[i] = stdin.nextInt();              values[i] = stdin.nextInt();          }          int capacity = stdin.nextInt();            // dp[i] will store max value of knapsack with capacity i.          int[] dp = new int[capacity+1];            // go through each item.          for (int i=0; i<n; i++) {                // If we do the loop backwards, we only get 1 copy of each item.              // Forwards, we can as many copies as we want.              //for (int w=capacity; w>=weights[i]; w--)              for (int w=weights[i]; w<=capacity; w++)                  dp[w] = Math.max(dp[w], dp[w-weights[i]] + values[i] );                // Just to understand the algorithm, we print at the end of each iteration.              for (int j=0; j<=capacity; j++)                  System.out.printf("%5d", dp[j]);              System.out.println();          }      }  } |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84 | // Arup Guha  // 10/8/2015  // Kruskal's Algorithm - written as an example for the programming team.    import java.util.\*;    class dset {        public int[] parent;      public int[] height;      public int n;        public dset(int size) {          parent = new int[size];          height = new int[size];          for (int i=0; i<size; i++)              parent[i] = i;      }        public int find(int v) {          if (parent[v] == v) return v;          parent[v] = find(parent[v]);          height[v] = 1;          return parent[v];      }        public boolean union(int v1, int v2) {            int p1 = find(v1);          int p2 = find(v2);          if (p1 == p2) return false;            if (height[p2] < height[p1]) parent[p2] = p1;          else if (height[p1] < height[p2]) parent[p1] = p2;          else {              parent[p2] = p1;              height[p1]++;          }          return true;      }  }    class edge implements Comparable<edge> {        public int v1;      public int v2;      public int w;        public edge(int a, int b, int weight) {          v1 = a;          v2 = b;          w = weight;      }        public int compareTo(edge other) {          return this.w - other.w;      }  }    class kruskals {        public static int mst(edge[] list, int n) {          Arrays.sort(list);            dset trees = new dset(n);          int numEdges = 0, res = 0;            // Consider edges in order.          for (int i=0; i<list.length; i++) {                // Try to put together these two trees.              boolean merged = trees.union(list[i].v1, list[i].v2);              if (!merged) continue;                // Bookkeepping.              numEdges++;              res += list[i].w;              if (numEdges == n-1) break;          }            // -1 indicates no MST, so not connected.          return numEdges == n-1 ? res : -1;      }  } |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87 | // Arup Guha  // 9/26/03  // The methods below solve the longest common subsequence problem  // recursively and using dynamic programming  import java.io.\*;    public class LCS {        public static void main(String [] args) throws IOException {          System.out.println("LCS = "+lcsrec("LYENCNERSKA","YECNQCGKAF"));          System.out.println("LCS = "+lcsrec2("LYENCNERSKA","YECNQCGKAF"));          System.out.println("LCS = "+lcsdyn("YECNQCGKAF","LYECNERSKA"));      }        // Precondition: Both x and y are non-empty strings.      public static int lcsrec(String x, String y) {            // Simple base case...          if (x.length() == 0 || y.length() == 0) return 0;            // Corresponding beginning characters match.          if (x.charAt(0) == y.charAt(0))              return 1+lcsrec(x.substring(1), y.substring(1));            // Corresponding characters do not match.          else              return Math.max(lcsrec(x,y.substring(1)), lcsrec(x.substring(1),y));    }        // Note: This second recursive version, though more cumbersome in code      //       because of how the substring method works more accurately      //       mirrors the algorithms used in the dynamic programming method      //       below, since this uses LCS's of prefixes to compute LCS's      //       of longer strings.      // Precondition: Both x and y are non-empty strings.      public static int lcsrec2(String x, String y) {            // Simple base case...          if (x.length() == 0 || y.length() == 0) return 0;            // Corresponding last characters match.          if (x.charAt(x.length()-1) == y.charAt(y.length()-1))              return 1+lcsrec2(x.substring(0,x.length()-1), y.substring(0,y.length()-1));            // Corresponding last characters do not match.          else              return Math.max(lcsrec2(x,y.substring(0,y.length()-1)), lcsrec2(x.substring(0,x.length()-1),y));      }        // Precondition: Both x and y are non-empty strings.      public static int lcsdyn(String x, String y) {            int i,j;          int lenx = x.length();          int leny = y.length();          int[][] table = new int[lenx+1][leny+1];            // Initialize table that will store LCS's of all prefix strings.          // This initialization is for all empty string cases.          for (i=0; i<=lenx; i++)              table[i][0] = 0;          for (i=0; i<=leny; i++)              table[0][i] = 0;            // Fill in each LCS value in order from top row to bottom row,          // moving left to right.          for (i = 1; i<=lenx; i++) {              for (j = 1; j<=leny; j++) {                    // If last characters of prefixes match, add one to former value.                  if (x.charAt(i-1) == y.charAt(j-1))                      table[i][j] = 1+table[i-1][j-1];                    // Otherwise, take the maximum of the two adjacent cases.                  else                      table[i][j] = Math.max(table[i][j-1], table[i-1][j]);                    System.out.print(table[i][j]+" ");              }              System.out.println();          }            // This is our answer.          return table[lenx][leny];      }    } |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | // Arup Guha  // 3/21/2017  // Shows DP approach to LIS problem.    import java.util.\*;    public class lis {        public static void main(String[] args) {            // Read in one case from input.          Scanner stdin = new Scanner(System.in);          int n = stdin.nextInt();          int[] vals = new int[n];          for (int i=0; i<n; i++)              vals[i] = stdin.nextInt();            int[] dp = new int[n];          dp[0] = 1;            // Find the best sequence that ends at index i.          System.out.print(dp[0]+" ");          for (int i=1; i<n; i++) {                // Initially we could just take this item.              dp[i] = 1;                // We try to build off j as the previous item in the increasing sequence.              for (int j=0; j<i; j++)                  if (vals[j] < vals[i])                      dp[i] = Math.max(dp[i], dp[j]+1);                // Just for learning...              System.out.print(dp[i]+" ");          }      }  } |  |  |  |  |  | | --- | --- | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | // Arup Guha  // 2/16/2016  // Some Math Code for COP 4516    import java.util.\*;    public class MathStuff {        public static void main(String[] args) {            // Test GCD.          Scanner stdin = new Scanner(System.in);          System.out.println("Enter two non-negative integers of which to find the GCD.");          int a = stdin.nextInt();          int b = stdin.nextInt();          System.out.println("The GCD is "+gcd(a,b));            // Test Prime Factorization.          System.out.println("Enter an integer to prime factorize.");          int n = stdin.nextInt();          ArrayList<pair> list = primeFactorize(n);          System.out.print("Your prime factorization is ");          for (int i=0; i<list.size()-1; i++)              System.out.print(list.get(i).prime+"^"+list.get(i).exp+" \* ");          System.out.println(list.get(list.size()-1).prime+"^"+list.get(list.size()-1).exp);            // Test Prime Sieve.          System.out.println("Enter a maximum bound for your prime search.");          n = stdin.nextInt();          boolean[] sieve = primeSieve(n);          System.out.print("Here are the primes up to n:");          for (int i=0; i<sieve.length; i++)              if (sieve[i])                  System.out.print(i+" ");          System.out.println();            // Test how many times p divides into n!          System.out.println("Enter a positive integer n and a prime p.");          n = stdin.nextInt();          int p = stdin.nextInt();          System.out.println(p+" divides "+n+"! "+numTimesDivide(n, p)+" times.");        }        // Returns the GCD of a and b.      public static int gcd(int a, int b) {          return b == 0 ? a : gcd(b, a%b);      }        public static ArrayList<pair> primeFactorize(int n) {            // Store the result here.          ArrayList<pair> res = new ArrayList<pair>();            int div = 2;            // Go till we know we're left with a prime.          while (div\*div <= n) {                // See how many times div divides into n.              int exp = 0;              while (n%div == 0) {                  n /= div;                  exp++;              }                // Add it, if it's a divisor.              if (exp > 0) res.add(new pair(div, exp));              div++;          }            // See if we have one last term to add before returning.          if (n > 1) res.add(new pair(n, 1));          return res;      }        // Returns an array of size n+1 such that array[i] = true iff i is prime.      public static boolean[] primeSieve(int n) {            // Initialize.          boolean[] isPrime = new boolean[n+1];          Arrays.fill(isPrime, true);          isPrime[0]= false;          isPrime[1] = false;            // Run really basic sieve.          for (int i=2; i\*i<=n; i++)              for (int j=2\*i; j<=n; j+=i)                  isPrime[j] = false;            // Here is our array.          return isPrime;      }        // Pre-condition: p is prime, n > 0.      // Post-condition: returns the number of times p divides into n!      public static int numTimesDivide(int n, int p) {          int res = 0;          while (n > 0) {              res += (n/p);              n /= p;          }          return res;      }    }    // Just for prime factorization.  class pair {      public int prime;      public int exp;        public pair(int p, int e) {          prime = p;          exp = e;      }   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | // Dynamic Programming Python implementation of Matrix  // Chain Multiplication.  // See the Cormen book for details of the following algorithm  class MatrixChainMultiplication  {      // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n      static int MatrixChainOrder(int p[], int n)      {          /\* For simplicity of the program, one extra row and one          extra column are allocated in m[][].  0th row and 0th          column of m[][] are not used \*/          int m[][] = new int[n][n];            int i, j, k, L, q;            /\* m[i,j] = Minimum number of scalar multiplications needed          to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where          dimension of A[i] is p[i-1] x p[i] \*/            // cost is zero when multiplying one matrix.          for (i = 1; i < n; i++)              m[i][i] = 0;            // L is chain length.          for (L=2; L<n; L++)          {              for (i=1; i<n-L+1; i++)              {                  j = i+L-1;                  if(j == n) continue;                  m[i][j] = Integer.MAX\_VALUE;                  for (k=i; k<=j-1; k++)                  {                      // q = cost/scalar multiplications                      q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];                      if (q < m[i][j])                          m[i][j] = q;                  }              }          }            return m[1][n-1];      }        // Driver program to test above function      public static void main(String args[])      {          int arr[] = new int[] {1, 2, 3, 4};          int size = arr.length;            System.out.println("Minimum number of multiplications is "+                             MatrixChainOrder(arr, size));      }  }  /\* This code is contributed by Rajat Mishra\*/ | | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | // Arup Guha  // 11/6/2014  // Written for DP #2 Lecture on Matrix Chain Multiplication Problem    import java.util.\*;    public class mcm {        public static int[][] memo;        public static void main(String[] args) {            Scanner stdin = new Scanner(System.in);          int numCases = stdin.nextInt();            // Process all input cases.          for (int loop=0; loop<numCases; loop++) {                // Read in all matrix dimensions.              int n = stdin.nextInt();              int[][] dim = new int[n][2];              for (int i=0; i<n; i++)                  for (int j=0; j<2; j++)                      dim[i][j] = stdin.nextInt();                // Set up memo table.              memo = new int[n][n];              for (int i=0; i<n; i++)                  Arrays.fill(memo[i], -1);                // Solve and print it.              System.out.println(minMult(dim, 0, n-1));          }      }        public static int minMult(int[][] dim, int sIndex, int eIndex) {            // If we've solved it, return the answer!          if (memo[sIndex][eIndex] != -1) return memo[sIndex][eIndex];            // No work to be done since the matrix itself is the answer.          if (sIndex == eIndex) return 0;            // Try first split.          int best = minMult(dim, sIndex+1, eIndex) + dim[sIndex][0]\*dim[sIndex][1]\*dim[eIndex][1];            // Try all other splits.          for (int split = sIndex+1; split<eIndex; split++) {              int costLeft = minMult(dim, sIndex, split);              int costRight = minMult(dim, split+1, eIndex);              int cost = costLeft + costRight + dim[sIndex][0]\*dim[split][1]\*dim[eIndex][1];              best = Math.min(best, cost);          }            // Store the best answer and return.          memo[sIndex][eIndex] = best;          return best;      }   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149 | // Arup Guha  // 2/2/2016  // Program that does a BFS through the possible board positions in the peg game.    // Note: In our bit mask, if a peg is in board location (r, c), then the bit SIZE\*r + c is 1,  //       otherwise, that bit is set to 0.    import java.util.\*;    public class peg {        final public static int SIZE = 5;      final public static boolean DEBUG = false;        // Bottom middle.      final public static int HOLE\_R = 4;      final public static int HOLE\_C = 2;        // Stores pairs of the form (board position, distances).      public static HashMap<Integer,Integer> distances;        // Possible jump directions.      final public static int[] DX = {-1,-1,0,0,1,1};      final public static int[] DY = {-1,0,-1,1,0,1};        public static void main(String[] args) {            // Change this to be whereever you want the hole.          int start = initBoard(HOLE\_R, HOLE\_C);            // Starting position.          distances = new HashMap<Integer,Integer>();          distances.put(start, 0);            // Queue for BFS.          LinkedList<Integer> q = new LinkedList<Integer>();          q.offer(start);            // Run BFS.          while (q.size() > 0) {                // Get next move.              int cur = q.poll();              int curdist = distances.get(cur);                // Get all possible next moves.              ArrayList<Integer> nextList = getNextPos(cur);                // Apply them - only add if overall board position is unique.              for (int i=0; i<nextList.size(); i++) {                  if (!distances.containsKey(nextList.get(i))) {                      distances.put(nextList.get(i), curdist+1);                      q.offer(nextList.get(i));                  }              }                // Initial version will only print possible ending boards.              if (nextList.size() == 0) print(cur);          }            // If you want to see everything...          if (DEBUG) {              for (Integer board: distances.keySet())                  print(board);          }      }        // Returns the mask of a full board with only an empty hole at (holeR, holeC).      public static int initBoard(int holeR, int holeC) {            // This is the full board.          int mask = 0;          for (int i=0; i<SIZE; i++)              for (int j=0; j<=i; j++)                  mask = mask | (1<<(SIZE\*i+j));            // Just subtract out the current hole.          return mask - (1<<(SIZE\*holeR+holeC));      }        // Returns all possible future positions from the board position represented by mask.      public static ArrayList<Integer> getNextPos(int mask) {            ArrayList<Integer> pos = new ArrayList<Integer>();            // Try each potential starting position.          for (int r =0; r<SIZE; r++) {              for (int c=0; c<=SIZE; c++) {                    // Now try each move.                  for (int dir=0; dir<DX.length; dir++) {                        // Ending square is out of bounds.                      if (!inbounds(r+2\*DX[dir], c+2\*DY[dir])) continue;                        // A move is valid only if the first two holes have pegs and the destination doesn't.                      if (on(mask, SIZE\*r+c) && on(mask, SIZE\*(r+DX[dir]) + c + DY[dir]) && !on(mask, SIZE\*(r+2\*DX[dir]) + c + 2\*DY[dir])) {                          int newpos = apply(mask, dir, r, c);                          pos.add(newpos);                      }                  }              }          }            // Here is a list of all the possible moves.          return pos;      }        // Prints the board position corresponding to mask with an extra blank line at the end.      public static void print(int mask) {            // Go through each row.          for (int i=0; i<SIZE; i++) {                // To make it look like a triangle.              for (int j=0; j<SIZE-1-i; j++) System.out.print(" ");                // Here are the actual holes.              for (int j=0; j<=i; j++) {                  if (on(mask, SIZE\*i+j)) System.out.print("X ");                  else                    System.out.print("\_ ");              }              System.out.println();          }          System.out.println();      }        // Returns the result of appplying the move from (r,c) in direction dir on the board stored in mask.      public static int apply(int mask, int dir, int r, int c) {            // Get the bit location of the starting hole, middle hole and ending hole.          int start = SIZE\*r + c;          int mid = SIZE\*(r+DX[dir]) + c + DY[dir];          int end = SIZE\*(r+2\*DX[dir]) + c + 2\*DY[dir];            // We get rid of the start and midding and add in the end to apply a jump.          return mask - (1<<start) - (1<<mid) + (1<<end);      }        // Returns true iff bit is on in mask.      public static boolean on(int mask, int bit) {          return (mask & (1<<bit)) != 0;      }        // Returns true iff (myr, myc) is inbounds.      public static boolean inbounds(int myr, int myc) {          return myr >= 0 && myr < SIZE && myc >= 0 && myc <= myr;      }  } |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   \*  Compilation:  javac Polygon.java   \*  Execution:    java Polygon   \*  Dependencies: Point.java   \*   \*  Implementation of 2D polygon, possibly intersecting.   \*   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    public class Polygon {      private int N;        // number of points in the polygon      private Point[] a;    // the points, setting points[0] = points[N]        // default buffer = 4      public Polygon() {          N = 0;          a = new Point[4];      }          // double size of array      private void resize() {          Point[] temp = new Point[2\*N+1];          for (int i = 0; i <= N; i++) temp[i] = a[i];          a = temp;      }        // return size      public int size() { return N; }        // draw polygon      public void draw() {          for (int i = 0; i < N; i++)              a[i].drawTo(a[i+1]);      }          // add point p to end of polygon      public void add(Point p) {          if (N >= a.length - 1) resize();   // resize array if needed          a[N++] = p;                        // add point          a[N] = a[0];                       // close polygon      }        // return the perimeter      public double perimeter() {          double sum = 0.0;          for (int i = 0; i < N; i++)              sum = sum + a[i].distanceTo(a[i+1]);          return sum;      }        // return signed area of polygon      public double area() {          double sum = 0.0;          for (int i = 0; i < N; i++) {              sum = sum + (a[i].x \* a[i+1].y) - (a[i].y \* a[i+1].x);          }          return 0.5 \* sum;      }        // does this Polygon contain the point p?      // if p is on boundary then 0 or 1 is returned, and p is in exactly one point of every partition of plane      // Reference: http://exaflop.org/docs/cgafaq/cga2.html      public boolean contains2(Point p) {          int crossings = 0;          for (int i = 0; i < N; i++) {              int j = i + 1;              boolean cond1 = (a[i].y <= p.y) && (p.y < a[j].y);              boolean cond2 = (a[j].y <= p.y) && (p.y < a[i].y);              if (cond1 || cond2) {                  // need to cast to double                  if (p.x < (a[j].x - a[i].x) \* (p.y - a[i].y) / (a[j].y - a[i].y) + a[i].x)                      crossings++;              }          }          if (crossings % 2 == 1) return true;          else                    return false;      }        // does this Polygon contain the point p?      // Reference: http://softsurfer.com/Archive/algorithm\_0103/algorithm\_0103.htm      public boolean contains(Point p) {          int winding = 0;          for (int i = 0; i < N; i++) {              int ccw = Point.ccw(a[i], a[i+1], p);              if (a[i+1].y >  p.y && p.y >= a[i].y)  // upward crossing                  if (ccw == +1) winding++;              if (a[i+1].y <= p.y && p.y <  a[i].y)  // downward crossing                  if (ccw == -1) winding--;          }          return winding != 0;      }          // return string representation of this point      public String toString() {          if (N == 0) return "[ ]";          String s = "";          s = s + "[ ";          for (int i = 0; i <= N; i++)              s = s + a[i] + " ";          s = s + "]";          return s;      }          // test client      public static void main(String[] args) {          int N = Integer.parseInt(args[0]);            // a square          Polygon poly = new Polygon();          poly.add(new Point(5, 5));          poly.add(new Point(9, 5));          poly.add(new Point(9, 9));          poly.add(new Point(5, 9));            StdOut.println("polygon    = " + poly);          StdOut.println("perimeter  = " + poly.perimeter());          StdOut.println("area       = " + poly.area());            StdOut.println("contains(5, 5) = " + poly.contains(new Point(5, 5)));          StdOut.println("contains(9, 5) = " + poly.contains(new Point(9, 5)));          StdOut.println("contains(9, 9) = " + poly.contains(new Point(9, 9)));          StdOut.println("contains(5, 9) = " + poly.contains(new Point(5, 9)));          StdOut.println("contains(7, 5) = " + poly.contains(new Point(7, 5)));          StdOut.println("contains(5, 7) = " + poly.contains(new Point(5, 7)));          StdOut.println("contains(7, 9) = " + poly.contains(new Point(7, 9)));          StdOut.println("contains(9, 7) = " + poly.contains(new Point(9, 7)));            // generate N random points in the unit square and check what fraction are in the polygon          int yes = 0;          for (int i = 0; i < N; i++) {              int x = (int) (10 \* Math.random());              int y = (int) (10 \* Math.random());              Point p = new Point(x, y);              if (poly.contains(p)) yes++;              if (poly.contains(p) != poly.contains2(p)) StdOut.println("different " + p);          }            // true answer is = 0.16 (depends on how boundary points are handled)          StdOut.println("Fraction in polygon = " + 1.0 \* yes / N);          }  }      Copyright © 2000–2017, Robert Sedgewick and Kevin Wayne.  Last updated: Fri Oct 20 12:50:46 EDT 2017. |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | // Arup Guha  // 10/8/2015  // Prim's Algorithm - written as an example for the programming team.    import java.util.\*;    class edge implements Comparable<edge> {        public int v1;      public int v2;      public int w;        public edge(int a, int b, int weight) {          v1 = a;          v2 = b;          w = weight;      }        public int compareTo(edge other) {          return this.w - other.w;      }  }    class prims {        public static int mst(ArrayList[] graph, int v) {            // Mark vertex v as being in mst.          int n = graph.length;          boolean[] used = new boolean[n];          used[v] = true;            // Add all of v's edges into the priority queue.          PriorityQueue<edge> pq = new PriorityQueue<edge>();          for (int i=0; i<graph[v].size(); i++)              pq.offer( ((ArrayList<edge>)graph[v]).get(i));            int numEdges = 0, res = 0;            while (pq.size() > 0) {                // Get next edge.              edge next = pq.poll();              if (used[next.v1] && used[next.v2]) continue;                // Add new items to priority queue - need to check which vertex is new.              if (!used[next.v1]) {                  for (int i=0; i<graph[next.v1].size(); i++)                      pq.offer( ((ArrayList<edge>)graph[next.v1]).get(i));                  used[next.v1] = true;              }              else {                  for (int i=0; i<graph[next.v2].size(); i++)                      pq.offer( ((ArrayList<edge>)graph[next.v2]).get(i));                  used[next.v2] = true;              }                // Bookkeeping              numEdges++;              res += next.w;              if (numEdges == n-1) break;          }            // -1 indicates no MST, so not connected.          return numEdges == n-1 ? res : -1;      }  } |  |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // Arup Guha  // 1/16/2018  // Use of ArrayDeque    import java.util.\*;    public class testarraydeque {        public static void main(String[] args) {            ArrayDeque<Integer> myad = new ArrayDeque<Integer>();            // Add stuff. 12, 7, 3, 5, 11          myad.addFirst(3);          myad.addFirst(7);          myad.addLast(5);          myad.addFirst(12);          myad.addLast(11);            // Remove in order from front to back.          while (myad.size() > 0)              System.out.println(myad.pollFirst());          }  } | |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | import java.util.\*;    class Node{        int data, height;      Node left, right;        Node(int data) {          this.data = data;          left = right = null;          height = 1;      }  }    public class avl {        Node root;      boolean balance;         int height(Node node)      {          if (node == null)              return 0;            return node.height;      }          boolean isBalanced(Node node)      {              int leftHeight , rightHeight;                if (node == null)                  return true;                leftHeight = height(node.left);              rightHeight = height(node.right);                if(Math.abs(leftHeight - rightHeight) <= 1 &&                 isBalanced(node.left) &&                 isBalanced(node.right))                      return true;                 return false;      }        void insert (int x) {          root = insertRecursive(root, x);      }        Node insertRecursive(Node root, int x) {            if (root == null)  {              root = new Node(x);              return root;          }              if (x < root.data)              root.left = insertRecursive(root.left, x);          else if (x > root.data)              root.right = insertRecursive(root.right, x);            root.height = 1 + Math.max( height(root.left), height(root.right));                return root;      }        public static void main(String args[]) {            Scanner in = new Scanner(System.in);          int trees = in.nextInt();            for(int loop = 0; loop < trees; loop++) {                int inserts = in.nextInt();              avl tree = new avl();              int treeNum = loop + 1;                for(int i = 0; i < inserts; i++)              {                  tree.insert(in.nextInt());                  if(!tree.isBalanced(tree.root) )                  {                      System.out.printf("Tree #%d: REMOVE", treeNum);                      break;                  }              }                if(tree.isBalanced(tree.root) )                  System.out.printf("Tree #%d: KEEP", treeNum);          }        }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84 | // Arup Guha  // November 5, 2006  // Written for COP 3503 Exam Solution  // Edited on 2/20/2014 forCOP 4516 to illustrate Bellman-Ford on a small graph.    import java.io.\*;  import java.util.\*;    public class bellmanford {        final public static int MAX = 1000000000;        // Short driver program to test the Bellman Ford's method.      public static void main(String[] args) {            // Read in graph.          int[][] adj = new int[5][5];          Scanner fin = new Scanner(System.in);          int numEdges = 0;          for (int i = 0; i<25; i++) {              adj[i/5][i%5] = fin.nextInt();              if (adj[i/5][i%5] != 0) numEdges++;          }            // Form edge list.          edge[] eList = new edge[numEdges];          int eCnt = 0;          for (int i = 0; i<25; i++)              if (adj[i/5][i%5] != 0)                  eList[eCnt++] = new edge(i/5, i%5, adj[i/5][i%5]);            // Run algorithm and print out shortest distances.          int[] answers = bellmanford(eList, 5, 0);          for (int i=0; i<5; i++)              System.out.print(answers[i]+" ");          System.out.println();      }        // Returns the shortest paths from vertex source to the rest of the      // vertices in the graph via Bellman Ford's algorithm.      public static int[] bellmanford(edge[] eList, int numVertices, int source) {            // This array will store our estimates of shortest distances.          int[] estimates = new int[numVertices];            // Set these to a very large number, larger than any path in our          // graph could possibly be.          for (int i=0; i<estimates.length; i++)              estimates[i] = MAX;            // We are already at our source vertex.          estimates[source] = 0;            // Runs v-1 times since the max number of edges on any shortest path is v-1, if          // there are no negative weight cycles.          for (int i=0; i<numVertices-1; i++) {                // Update all estimates based on this particular edge only.              for (edge e: eList) {                  if (estimates[e.v1]+e.w < estimates[e.v2])                      estimates[e.v2] = estimates[e.v1] + e.w;              }            }          return estimates;      }  }    class edge {        public int v1;      public int v2;      public int w;        public edge(int a, int b, int c) {          v1 = a;          v2 = b;          w = c;      }        public void negate() {          w = -w;      }  } |