2017

Hack Pack

TEAM TEQUILA

NICHOLAS DEVIEW, CUONG NGUYEN, TYLER JONES

Contents

Permutation/Combination Generation

// Prints all **permutation** 0,...,n-1 where the first k items of perm are fixed.

// perms(perm[n], used[0,0,…,n], k, n);

```
void perms(int perm[], int used[], int k, int n) {

   if (k == n){
      for (i=0; i<n; i++){
         System.out.print(perm[i]);
      }
       System.out.println();
   }

   else {
      for (int i=0; i<n; i++) {
         if (!used[i]) {
            used[i] = 1;
            perm[k] = i;
            perms(perm, used, k+1, n);
            used[i] = 0;
         }
      }
   }
}
```

// Prints all **combinations** of 0,1,2,3,…,n.

// combos(items[0,1,…,n], k, n);

```
void combos(int subset[], int k, int n) {

   if (k == n){
      for (int i=0; i<n; i++){
         if (subset[i])
             System.out.print(i);
      }
      System.out.println();
   }
   else {
      combos(subset, k+1, n);

      subset[k] = 1;
      combos(subset, k+1, n);
      subset[k] = 0;
   }
}
```

// Prints out all **derangements** in perm with the first k digits fixed.

// derangements(perm[n], used[0,0,…,n], k, n);

```
void derangements(int perm[], int used[], int k, int n) {

    if (k == n){
        for (i=0; i<n; i++){
            System.out.print(perm[i]);
        }
         System.out.println();
    }

    else {

        for (int i=0; i<n; i++) {
            if (!used[i] && i != k) {
                used[i] = 1;
                perm[k] = i;
                derangements(perm, used, k+1, n);
                used[i] = 0;
            }
        }
    }
}
```

Permutation/Combination code source (Arup Guha)

```
// Arup Guha
// 9/18/2015
// Examples for Brute Force Lecture for Programming Team
// http://www.cs.ucf.edu/~dmarino/progcontests/devteam-15/bruteforce.c

// 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 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;
            }
        }
    }
}
```

GCD/LCM/Prime Factorization/Prime Sieve

```
// Returns the GCD of a and b.
public static int gcd(int a, int b) {
   if(b==0)
      return a;

   return gcd(b, a%b);
}


//Returns LCM of a and b.
public static int lcm(int a, int b){

   return a * (b / gcd(a, b));

}



//Prime Factorization
class pair {
   public int prime;
   public int exp;

   public pair(int p, int e) {
      prime = p;
      exp = e;
   }
}


public static ArrayList<pair> primeFactorize(int n) {
      ArrayList<pair> res = new ArrayList<pair>();

      int div = 2;

      while (div*div <= n) {

         int exp = 0;

         while (n%div == 0) {
            n /= div;
            exp++;
         }

         if (exp > 0)
```

```
            res.add(new pair(div, exp));
        div++;
    }

    if (n > 1)
        res.add(new pair(n, 1));

    return res;
}
```

//PrimeSieve

```
public static boolean[] primeSieve(int n) {

    boolean[] isPrime = new boolean[n+1];
    Arrays.fill(isPrime, true);
    isPrime[0]= false;
    isPrime[1] = false;

    for (int i=2; i*i<=n; i++)
        for (int j=2*i; j<=n; j+=i)
            isPrime[j] = false;

    return isPrime;
}
```

GCD/Prime Factorization/Prime Sieve source (Arup Guha)

```
// 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();
    }

    // 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;
    }


// Just for prime factorization.
```

```
class pair {
    public int prime;
    public int exp;

    public pair(int p, int e) {
        prime = p;
        exp = e;
    }
}
```

Kruskals Algorithm

```
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;

		for (int i=0; i<list.length; i++) {

			boolean merged = trees.union(list[i].v1, list[i].v2);
			if (!merged) continue;

			numEdges++;
			res += list[i].w;
			if (numEdges == n-1) break;
		}

		if(numEdges == n-1)
			return res;
		else
			return -1;
	}
}
```

Kruskals Algorithm Source (Arup Guha)

```
// 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;
	}
}
```

DFS/BFS

```
void DFS()
{
	boolean[] V=new boolean[N];
	int numComponets=0;

		for (int i=0; i<N; ++i)
		if (!V[i])
		{
			numComponets++;
			DFS(i,V);
		}

	System.out.println("found “ + numComponets + “ components.");
}

//starts at node at
void DFS(int at, boolean[] V)
{

	V[at]=true;

	for (int i=0; i<N; ++i)
		if (G[at][i] && !V[i])
		{
			DFS(i,V);
		}
}

void BFS()
{
	boolean[] V=new boolean[N];

	int numComponets=0;

	for (int i=0; i<N; ++i)
		if (!V[i])
		{
			numComponets++;
			BFS(i,V);
		}
	System.out.println("found “ + numComponets + “ components.");
}
```

```
//starts at node start
void BFS(int start, boolean[] V)
{
	Queue<Integer> Q=new LinkedList<Integer>();

	Q.offer(start);
	V[start]=true;

	while (!Q.isEmpty())
	{
		int at=Q.poll();

		for (int i=0; i<N; ++i)
			if (G[at][i] && !V[i])
			{
				Q.offer(i);
				V[i]=true;
			}
	}
}
```

DFS/BFS source (Stephen Fulwider)

```
// Stephen Fulwider
//	A sample program to show working examples of Depth/Breadth First Search
(DFS,BFS)

	// 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);
        }
```

Topologicalsort

```
public class topologicalsort {

	int vertices;
	LinkedList<Integer> adj[];

	// Constructor
	topologicalsort(int v)
	{
		vertices = v;
		adj = new LinkedList[v];
		for (int i = 0; i < v; i++)
			adj[i] = new LinkedList();
	}

	void addEdge(int v, int weight) {
		adj[v].add(weight);
	}

	void sortHelper(int v, boolean visited[], Stack stack)
	{
		visited[v] = true;
		Integer i;

		Iterator<Integer> it = adj[v].iterator();
		while (it.hasNext())
		{
			i = it.next();
			if (!visited[i])
				sortHelper(i, visited, stack);
		}

		stack.push(new Integer(v));
	}

	void topologicalSort()
	{
		Stack stack = new Stack();

		boolean visited[] = new boolean[vertices];
		for (int i = 0; i < vertices; i++)
			visited[i] = false;

		for (int i = 0; i < vertices; i++)
			if (!visited[i])
```

```
                    sortHelper(i, visited, stack);

            while (!stack.empty())
                System.out.print(stack.pop() + " ");
        }
}
```

Floyd-Warshall's Algorithm (Stephen Fulwider)

```
//Stephen Fulwider
//Floyd's all pairs shortest path algorithm with path reconstruction
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;
	}

}
```

Dijkstra’s Algorithm

```
public class dijkstra {

	int BIG = (int) 1e9;
	int num; // num of nodes
	ArrayList<Edge>[] graph;

	public int dijkstras(int v1, int v2) {

	  boolean[] visited = new boolean[num];
	  PriorityQueue<Edge> queue = new PriorityQueue<Edge>();
	  queue.add(new Edge(v1, 0));

	  while (!queue.isEmpty()) {
			Edge at = queue.poll();
			if (visited[at.edge])
				continue;

			visited[at.edge] = true;

			if (at.edge == v2)
				return at.weight;

			for (Edge adj : graph[at.edge]) {
				if (!visited[adj.edge])
					queue.add(new Edge(adj.edge, adj.weight + at.weight));
			}
	  }

			return BIG;
	}
}

class Edge implements Comparable<Edge> {
	int edge, weight;

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

	public int compareTo(Edge o) {
		return weight - o.weight;
	}
}
```

Dijkstra's Algorithm (Danny Wasserman)

```
//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;
  }
 }
}
```

Bellman Ford’s Algorithm

```
public class bellmanford {

	static int BIG = (int) 1e9;
	static int num = 5; // num nodes, change accordingly
	static edge[] list = new edge[num];

	public static void main(String[] args) {

		int[] ans = new int[num];

		for (int i = 0; i < ans.length; i++) {
			ans[i] = BIG;
		}

		ans[0] = 0; // assumes the beginning is 0

		for (int i = 0; i < num-1; i++) {
			for (edge e : list) {
				if ((ans[e.v1] + e.weight) < ans[e.v2])
					ans[e.v2] = ans[e.v1] + e.weight;
			}
		}

		// return ans; - answer to be returned
	}
}

class edge {

	public int v1, v2, weight;

	public edge(int v1, int v2, int weight) {
		this.v1 = v1;
		this.v2 = v2;
		this.weight = weight;
	}

	public void negate() {
		weight = -weight;
	}
}
```

Bellman-Ford (Arup Guha)

```
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;
	}
}
```

```
// Team Tequila
public class networkflow {
	// different network flow algorithms below
}

class FordFulkerson {

	int[][] limit;
	boolean[] visited;
	int BIG = (int) (1e9);

	public FordFulkerson(int size) {
		int n = size + 2;
		int s = n - 2;
		int t = n - 1;
		limit = new int[n][n];
	}

	void add(int v1, int v2, int c) {
		limit[v1][v2] = c;
	}

	int ff(int s, int n, int t) {
		visited = new boolean[n];
		int f = 0;
		while (true) {
			Arrays.fill(visited, false);  int res = dfs(s, BIG, t, n);
			if (res == 0) {
				break;
			}
			f += res;
		}
		return f;
	}

	int dfs(int pos, int min, int t, int n) {

		if (pos == t)
			return min;
		if (visited[pos])
			return 0;
		visited[pos] = true;  int f = 0;

		for (int i = 0; i < n; i++) {
			if (limit[pos][i] > 0)
```

```
                    f = dfs(i, Math.min(limit[pos][i], min), t, n);
                if (f > 0) {
                    limit[pos][i] -= f;
                    limit[i][pos] += f;
                    return f;
                }
            }
            return 0;
        }
}

class EdmundsKarp {

        public int[][] limit;
        public int num;
        public int source;
        public int sink;
        public int BIG = (int)(1E9);

        public EdmundsKarp(int size) {
            num = size + 2;
            source = num - 2;
            sink = num - 1;
            limit = new int[num][num];
        }

        public void add(int v1, int v2, int c) {
            limit[v1][v2] = c;
        }

        public int flow() {
            int flow = 0;

            while (true) {
                int result = bfs();
                if (result == 0)
                    break;

                flow += result;
            }

            return flow;
        }

        public int bfs() {
```

```
		int[] reach = new int[num+2];
		int[] prev = new int[num+2];
		LinkedList<Integer> queue = new LinkedList<Integer>();

		reach[source] = BIG;
		Arrays.fill(prev, -1);
		prev[source] = source;

		queue.offer(source);

		while (queue.size() > 0) {
			int v = queue.poll();
			if (v == sink) break;

			for (int i=0; i<num; i++) {
				if (prev[i] == -1 && limit[v][i] > 0) {
					prev[i] = v;
					reach[i] = Math.min(limit[v][i], reach[v]);
					queue.offer(i);
				}
			}
		}

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

		int v1 = prev[sink];
		int v2 = sink;
		int flow = reach[sink];

		while (v2 != source) {
			limit[v1][v2] -= flow;
			limit[v2][v1] += flow;
			v2 = v1;
			v1 = prev[v1];
		}

		return flow;
	}
}

class Dinic {

	ArrayDeque<Integer> queue;
	ArrayList<Edge>[] adj;
	int n, s, t, oo = (int)1E9;
```

```
boolean[] blocked;
int[] dist;

public Dinic (int N) {
	n = N; s = n++; t = n++;
	blocked = new boolean[n];
	dist = new int[n];
	queue = new ArrayDeque<Integer>();
	adj = new ArrayList[n];
	for(int i = 0; i < n; ++i)
		adj[i] = new ArrayList<Edge>();
}

void add(int v1, int v2, int limit, int flow) {
	Edge e = new Edge(v1, v2, limit, flow);
	Edge rev = new Edge(v2, v1, 0, 0);
	adj[v1].add(rev.rev = e);
	adj[v2].add(e.rev = rev);
}

boolean bfs() {
	queue.clear();
	Arrays.fill(dist, -1);
	dist[t] = 0;
	queue.add(t);

	while(!queue.isEmpty()) {
		int node = queue.poll();

		if(node == s)
			return true;

		for(Edge e : adj[node]) {
			if(e.rev.limit > e.rev.flow && dist[e.v2] == -1) {
				dist[e.v2] = dist[node] + 1;
				queue.add(e.v2);
			}
		}
	}
	return dist[s] != -1;
}

int dfs(int pos, int min) {
	if(pos == t)
		return min;
```

```
		int flow = 0;

		for(Edge e : adj[pos]) {
			int cur = 0;
			if(!blocked[e.v2] && dist[e.v2] == dist[pos]-1 && e.limit - e.flow >
0) {
				cur = dfs(e.v2, Math.min(min-flow, e.limit - e.flow));
				e.flow += cur;
				e.rev.flow = -e.flow;
				flow += cur;
			}

			if(flow == min)
				return flow;

		}
		blocked[pos] = flow != min;
		return flow;
	}

	int flow() {
		clear();
		int ret = 0;

		while(bfs()) {
			Arrays.fill(blocked, false);
			ret += dfs(s, oo);
		}

		return ret;
	}

	void clear() {
		for(ArrayList<Edge> list : adj)
			for(Edge e : list)
				e.flow = 0;
	}
}

class Edge { // for dinic

	int v1, v2, limit, flow;
	Edge rev;

	Edge(int v1, int v2, int limit, int flow) {
		this.v1 = v1;
```

```
		this.v2 = v2;
		this.limit = limit;
		this.flow = flow;
	}
}
```

```
//"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;
		}
	}
```

Arup Guha
Edit of FordFulkerson Code (from UCF Hackpack)
3/6/2017
Code for Edmunds Karp Algorithm

```
import java.util.*;

public class EdmundsKarp {

	// 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 EdmundsKarp(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 flow() {

		// Set visited array and flow.
		int flow = 0;

		// Loop until no augmenting paths found.
		while (true) {

			// Run one BFS.
			int res = bfs();

			// 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 bfs() {

		// Set up BFS.
		int[] reach = new int[n+2];
		int[] prev = new int[n+2];
		LinkedList<Integer> q = new LinkedList<Integer>();
		reach[source] = oo;
		Arrays.fill(prev, -1);
		prev[source] = source;
		q.offer(source);

		// Run BFS loop.
		while (q.size() > 0) {

			// Get next node - if it's sink, we're done.
			int v = q.poll();
			if (v == sink) break;

			// Try each neighbor.
			for (int i=0; i<n; i++) {

				// If we can go here, mark, previous, flow to i, and put i in
queue.

				if (prev[i] == -1 && cap[v][i] > 0) {
					prev[i] = v;
					reach[i] = Math.min(cap[v][i], reach[v]);
					q.offer(i);
				}
			}
		}

		// Didn't work.
		if (reach[sink] == 0) return 0;

		// Mark last two vertices.
		int v1 = prev[sink];
```

```
			int v2 = sink;
			int flow = reach[sink];

			// Actually put flow through.
			while (v2 != source) {

				// Puts flow through.
				cap[v1][v2] -= flow;
				cap[v2][v1] += flow;

				// Moves to previous edge.
				v2 = v1;
				v1 = prev[v1];
			}

			// This was our flow.
			return flow;
		}
}
```

"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;
	}
```

Dynamic programming

Binomial Coefficient

```
int binomialCoefficent(int n, int k){
		int mem[][]=new int[n+1][k+1];
		for(int i=0;i<n+1;i++){
				mem[i][0]=1;
				mem[i][i]=1;
		}
		for(int i=2;i<n+1;i++){
				for(int j=1;j<i;j++){
				mem[i][j] = mem[i-1][j-1]
						+mem[i-1][j];
				}
		}
		return mem[n][k];
}
```

Subset Sum

```
boolean isSS(int set[], int n, int sum){
		boolean subset[][] = new boolean[sum+1][n+1];
		for(int i=0;i<=sum;i++)
				subset[0][i] = true;
		for(int i=1;i<=sum;i++)
				subset[i][0] = false;
		for(int i=1;i<=sum;i++)
				for(int j=1;j<=n;j++){
						subset[i][j]=subset[i][j-1];
						if(i>=set[j-1])
								subset[i][j]=subset[i][j]||
								subset[i-set[j-1]][j-1];
				}
		return subset[sum][n];
}
```

Subset Sum (Arup Guha)

```
boolean isSS(int set[],int sum){
		boolean[] mem = new boolean[sum+1];
		Arrays.fill(mem, false);
		for(int i=0;i<set.length;i++)
				for(int j=sum;j>=set[i];j--)
						if(mem[j-set[i]])
								mem[j]=true;

		//Choose 1 below
		//subset specific number
		int[] subset = new int[sum+1];
		Arrays.fill(subset,0);
```

```
        for(int i=0;i<S.length;i++)
                for(int j=sum;j>=set[i];j--)
                        if(subset[j-set[i]]!=0
                        || j==set[i])
                        subset[j] = set[i];

        //subset with multiple copies
        for(int i=0;i<set.length;i++)
                for(int j=set[i];j<=sum;j++)
                        if(subset[j-set[i]]!=0
                        || j == set[i])
                        subset[j] = set[i];
}
```

Knapsack

```
int knapsack(int[] weights, int[] values,capacity,int n){
        int[] dp = new int[capacity+1];
        for (int i=0; i<n; i++)
                for (int w=weights[i]; w<=capacity; w++)
                        dp[w] = Math.max(dp[w], dp[w-weights[i]] + values[i] );
                return dp[n];
}
```

Longest Common Subsequence

```
int lcs(String x,String y){
        int i,j;
        int lenx = x.length();
        int leny = y.length();
        int[][] table = new int[lenx+1][leny+1];
                for (i = 1; i<=lenx; i++) {
                        for (j = 1; j<=leny; j++) {
                                if (x.charAt(i-1) == y.charAt(j-1))
                                        table[i][j] = 1+table[i-1][j-1];
                                else
                                        table[i][j] = Math.max(table[i][j-1], table[i-1][j]);
                        }
                }
                return table[lenx][leny];
}
```

Coint Change

```
static long countChangeWays(int S[], int m, int n)
{
        long[] table = new long[n+1];
        Arrays.fill(table, 0);
```

```
    table[0] = 1;
    for (int i=0; i<m; i++)
        for (int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];
    return table[n];
}
```

Geometry

Line-line

```
class PT{
   double x, y;
   PT(){}
   PT(double x, double y){this.x =x; this.y=y;}
   PT add(PT p){return new PT(x+p.x,y+p.y);}
   PT minus(PT p){return new PT(x-p.x,y-p.y);}
   PT mul(double c){return new PT(x*c, y*c);}
   PT div(double c){return new PT(x/c, y/c);}
}
public class geometry {
   static final double INF = 1e100;
   static final double EPS = 1e-12;

   static double dot(PT p,PT q){return p.x*q.x+p.y*q.y;}
   static double dist2(PT p, PT q){return dot(p.minus(q),p.minus(q));}
   static double cross(PT p, PT q){return p.x*q.y-p.y*q.x;}

            /**LINE**/
   //determine if lines-segment ab and cd are // or colinear
   static boolean LinesParallel(PT a, PT b, PT c, PT d){
      return (Math.abs(cross(b.minus(a), c.minus(d)))< EPS);
   }
   static boolean LinesCollinear(PT a, PT b, PT c, PT d){
      return (LinesParallel(a, b, c, d) &&
            Math.abs(cross(a.minus(b),a.minus(c)))<EPS
            && Math.abs(cross(c.minus(d), c.minus(a)))<EPS
            );
   }

   //determine if line segment ab intersects with line-segment cd
   static boolean SegmentsIntersect(PT a, PT b, PT c, PT d){
      if(LinesCollinear(a, b, c, d)){
         if (dist2(a,c) < EPS || dist2(a,d) < EPS ||
                                    dist2(b,c)< EPS || dist2(b,d)<EPS)
                                    return true;
                              if(dot(c.minus(a), c.minus(b)) >0 && dot(d.minus(a),d.minus(b))>0
                                    && dot(c.minus(b),d.minus(b))>0)
                                    return false;
                              return true;
      }
      if(cross(d.minus(a),b.minus(a))*cross(c.minus(a), b.minus(a) )>0) return false;
      if(cross(a.minus(c),d.minus(c))*cross(b.minus(c), d.minus(c) )>0) return false;

      return true;
```

```
    }

        //compute intersection of line through a and b with line and line through c and d
        //assume that point exist
        PT ComputeLineIntersection(PT a, PT b, PT c ,PT d){
                b= b.minus(a);
                d = c.minus(d);
                c = c.minus(a);
                assert dot(b,b)> EPS && dot(d,d)>EPS;
                return a.add(b.mul(cross(c,d)/cross(b,d)));
        }

        PT RotateCCW90(PT p){return new PT(-p.y,p.x);}
        PT RotateCW90(PT p){return new PT(p.y,-p.x);}
        PT RotateCCW(PT p,double t){
                return new PT(p.x*Math.cos(t)-p.y*Math.sin(t),
p.x*Math.sin(t)+p.y*Math.cos(t));
        }

        //project point c onto line through a and b
        //assuming a!=  b
        PT ProjectPointLine(PT a, PT b, PT c){
                return
a.add((b.minus(a)).mul(dot(c.minus(a),c.minus(a))/dot(b.minus(a),b.minus(a))));
        }

        //project point c onto line through a and b
        PT ProjectPointSegment(PT a, PT b, PT c){
                double r =  dot(b.minus(a), b.minus(a));
                if(Math.abs(r)< EPS) return a;
                r = dot(c.minus(a), b.minus(a));
                if(r<0)return a;
                if(r>1)return b;
                return a.add((b.minus(a)).mul(r));
        }

        //Compute Distance from c to segment between a and b
        double DistancePointSegment(PT a, PT b, PT c){
                return Math.sqrt(dist2(c, ProjectPointSegment(a,b,c)));
        }
```

Polygon

```
        // determine if point is in a possibly non-convex polygon (by William
        // Randolph Franklin); returns 1 for strictly interior points, 0 for
        // strictly exterior points, and 0 or 1 for the remaining points.
        // Note that it is possible to convert this into an *exact* test using
```

```
	// integer arithmetic by taking care of the division appropriately
	// (making sure to deal with signs properly) and then by writing exact
	// tests for checking point on polygon boundary
	boolean PointInPolygon(Vector<PT> p, PT q) {
		boolean c = false;
		for (int i = 0; i < p.size(); i++){
			int j = (i+1)%p.size();
			if ((p.get(i).y <= q.y && q.y < p.get(j).y ||
			p.get(j).y <= q.y && q.y < p.get(i).y) &&
			q.x < p.get(i).x + (p.get(j).x - p.get(i).x) * (q.y - p.get(i).y) /
(p.get(j).y - p.get(i).y))
				c = !c;
		}
		return c;
	}

	// determine if point is on the boundary of a polygon
	boolean PointOnPolygon(Vector<PT> p, PT q) {
		for (int i = 0; i < p.size(); i++)
			if (dist2(ProjectPointSegment(p.get(i), p.get((i+1)%p.size()), q), q)
< EPS)
				return true;
		return false;
	}
```

Point In Poly (Arup Guha)

```
	public static boolean ptInPoly(pt myPt, pt[] poly) {

		double sumAngles = 0;

		// Add up angles from myPt to successive pairs of vertices in the poly.
		for (int i=0; i<poly.length; i++) {
			vect v1 = myPt.getVect(poly[i]);
			vect v2 = myPt.getVect(poly[(i+1)%poly.length]);
			sumAngles += v1.angleBetween(v2);
		}

		// If this sum is close to 2pi, we're good.
		return Math.abs(sumAngles - Math.PI*2) < 1e-9;
	}
}
```

Convex Hull

```
class Point implements Comparable<Point> {
	int x, y;

	public int compareTo(Point p) {
		if (this.x == p.x) {
			return this.y - p.y;
		} else {
			return this.x - p.x;
		}
	}

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

}

public class ConvexHull {

	public static long cross(Point O, Point A, Point B) {
		return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
	}

	public static Point[] convex_hull(Point[] P) {

		if (P.length > 1) {
			int n = P.length, k = 0;
			Point[] H = new Point[2 * n];

			Arrays.sort(P);

			// Build lower hull
			for (int i = 0; i < n; ++i) {
				while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
					k--;
				H[k++] = P[i];
			}

			// Build upper hull
			for (int i = n - 2, t = k + 1; i >= 0; i--) {
				while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
					k--;
				H[k++] = P[i];
```

```
				}
				if (k > 1) {
					H = Arrays.copyOfRange(H, 0, k - 1); // remove non-hull vertices after k;
remove k - 1 which is a duplicate
				}
				return H;
			} else if (P.length <= 1) {
				return P;
			} else{
				return null;
			}
		}

		public static void main(String[] args) throws IOException {

			BufferedReader f = new BufferedReader(new FileReader("hull.in"));	// "hull.in"
Input Sample => size x y x y x y x y
			StringTokenizer st = new StringTokenizer(f.readLine());
			Point[] p = new Point[Integer.parseInt(st.nextToken())];
			for (int i = 0; i < p.length; i++) {
				p[i] = new Point();
				p[i].x = Integer.parseInt(st.nextToken()); // Read X coordinate
				p[i].y = Integer.parseInt(st.nextToken()); // Read y coordinate
			}

			Point[] hull = convex_hull(p).clone();

			for (int i = 0; i < hull.length; i++) {
				if (hull[i] != null)
					System.out.print(hull[i]);
			}
		}

}
```

ConvexHull (Arup Guha)

```
public class convexhull {

	public static void main(String[] args) throws Exception {

		// Read in the points.
		BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
		int n = Integer.parseInt(stdin.readLine().trim());
		pt[] pts = new pt[n];
		for (int i=0; i<n; i++) {
			StringTokenizer tok = new
StringTokenizer(stdin.readLine());
			int x = Integer.parseInt(tok.nextToken());
			int y = Integer.parseInt(tok.nextToken());
			pts[i] = new pt(x, y);
		}

		// Set the reference point.
		int refIndex = getIndexMin(pts, n);
		pt.refX = pts[refIndex].x;
		pt.refY = pts[refIndex].y;

		// Output solution.
		System.out.printf("%.1f\n", grahamScan(pts, n));
	}

	// Returns the point in pts with minimum y breaking tie by minimum x.
	public static int getIndexMin(pt[] pts, int n) {
		int res = 0;
		for (int i=1; i<n; i++)
			if (pts[i].y < pts[res].y || (pts[i].y == pts[res].y &&
pts[i].x < pts[res].x))
				res = i;
		return res;
	}

	public static double grahamScan(pt[] pts, int n) {

		// Sort the points by angle with reference point.
		Arrays.sort(pts);

		// Push first two points on.
		Stack<pt> myStack = new Stack<pt>();
		myStack.push(pts[0]);
		myStack.push(pts[1]);

		// Go through the rest of the points.
		for (int i=2; i<n; i++) {

			// Get last three pts.
			pt cur = pts[i];
			pt mid = myStack.pop();
			pt prev = myStack.pop();

			// Pop off the left turns.
```

```
			while (!prev.isRightTurn(mid, cur)) {
				mid = prev;
				prev = myStack.pop();
			}

			// Push back the last right turn.
			myStack.push(prev);
			myStack.push(mid);
			myStack.push(cur);
		}

		// Add up distances around the hull.
		double res = 0;
		pt cur = pts[0];
		while (myStack.size() > 0) {
			pt next = myStack.pop();
			res += cur.dist(next);
			cur = next;
		}

		// Return.
		return res;
	}
}

class pt implements Comparable<pt> {

	// Stores reference pt
	public static int refX;
	public static int refY;

	public int x;
	public int y;

	public pt(int myx, int myy) {
		x = myx;
		y = myy;
	}

	// Returns the vector from this to other.
	public pt getVect(pt other) {
		return new pt(other.x-x, other.y-y);
	}

	// Returns the distance between this and other.
	public double dist(pt other) {
		return Math.sqrt((other.x-x)*(other.x-x) + (other.y-
y)*(other.y-y));
	}

	// Returns the magnitude ot this cross product other.
	public int crossProductMag(pt other) {
		return this.x*other.y - other.x*this.y;
	}

	// returns true iff this to mid to next is a right turn (180 degree is
considered right turn).
```

```java
	public boolean isRightTurn(pt mid, pt next) {
		pt v1 = getVect(mid);
		pt v2 = mid.getVect(next);
		return v1.crossProductMag(v2) >= 0; /*** Change to > 0 to skip 
collinear points. ***/
	}

	// Returns true iff this pt is the origin.
	public boolean isZero() {
		return x == 0 && y == 0;
	}

	public int compareTo(pt other) {

		pt myRef = new pt(refX, refY);
		pt v1 = myRef.getVect(this);
		pt v2 = myRef.getVect(other);

		// To avoid 0 issues.
		if (v1.isZero()) return -1;
		if (v2.isZero()) return 1;

		// Angles are different, we are going counter-clockwise here.
		if (v1.crossProductMag(v2) != 0)
			return -v1.crossProductMag(v2);

		// This should work, smaller vectors come first.
		if (myRef.dist(v1) < myRef.dist(v2)) return -1;
		return 1;
	}
}
```

Geometry 3D

Point-plane

```
// distance from point (x, y, z) to plane aX + bY + cZ + d = 0
public static double ptPlaneDist(double x, double y, double z,
    double a, double b, double c, double d) {
  return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
public static double planePlaneDist(double a, double b, double c,
    double d1, double d2) {
  return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
}

// distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
// (or ray, or segment; in the case of the ray, the endpoint is the
// first point)
public static final int LINE = 0;
public static final int SEGMENT = 1;
public static final int RAY = 2;
public static double ptLineDistSq(double x1, double y1, double z1,
    double x2, double y2, double z2, double px, double py, double pz,
    int type) {
  double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

  double x, y, z;
  if (pd2 == 0) {
    x = x1;
```

```
    y = y1;
    z = z1;
  } else {
    double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
    x = x1 + u * (x2 - x1);
    y = y1 + u * (y2 - y1);
    z = z1 + u * (z2 - z1);
    if (type != LINE && u < 0) {
      x = x1;
      y = y1;
      z = z1;
    }
    if (type == SEGMENT && u > 1.0) {
      x = x2;
      y = y2;
      z = z2;
    }
  }

  return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz);
}

public static double ptLineDist(double x1, double y1, double z1,
    double x2, double y2, double z2, double px, double py, double pz,
    int type) {
  return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));
}
```

Line-plane (Arup Guha)

```
class pt {

	public double x;
	public double y;
	public double z;

	public pt(double myx, double myy, double myz) {
		x = myx;
		y = myy;
		z = myz;
	}

	// Returns the vector from this to to.
	public vect getVect(pt to) {
		return new vect(to.x-x, to.y-y, to.z-z);
	}
}

class vect {

	public double x;
	public double y;
	public double z;

	public vect(double myx, double myy, double myz) {
		x = myx;
		y = myy;
		z = myz;
	}

	// Returns the cross product this x other.
	public vect cross(vect other) {
		return new vect(y*other.z-other.y*z, z*other.x-other.z*x, x*other.y-other.x*y);
	}

	// Returns the magnitude of this vect.
	public double mag() {
		return Math.sqrt(x*x+y*y+z*z);
	}
}

class line {
```

```
	public pt start;
	public pt end;
	public vect dir;

	public line(pt s, pt e) {
		start = s;
		end = e;
		dir = start.getVect(end);
	}

	// Returns the point that corresponds to parameter t on this line.
	public pt getPt(double t) {
		return new pt(start.x+dir.x*t, start.y+dir.y*t, start.z+dir.z*t);
	}
}

class plane {

	public pt p1;
	public pt p2;
	public pt p3;
	public vect normal;
	public double d;

	public plane(pt a, pt b, pt c) {
		p1 = a;
		p2 = b;
		p3 = c;
		vect v1 = p1.getVect(p2);
		vect v2 = p1.getVect(p3);
		normal = v1.cross(v2);

		// We get D by plugging in one of the plane points into the equation for the plane.
		d = normal.x*p1.x + normal.y*p1.y + normal.z*p1.z;
	}

	public pt intersect(line myLine) {

		// Get coefficient of parameter t in solution - cull out intersections that aren't points.
		double tCoeff = normal.x*myLine.dir.x + normal.y*myLine.dir.y + normal.z*myLine.dir.z;
		if (Math.abs(tCoeff) < 1e-9) return null;

		// Solve for the parameter.
```

```
			double rhs = d - normal.x*myLine.start.x - normal.y*myLine.start.y -
normal.z*myLine.start.z;

			// Return the corresponding point.
			return myLine.getPt(rhs/tCoeff);
		}

		// Returns true iff p is on this plane.
		public boolean onPlane(pt p) {
			return normal.x*p.x + normal.y*p.y + normal.z*p.z == d;
		}
}
```