**A1 – C++**

#include <bits/stdc++.h>

using namespace std;

int main() {

 int a, b;

 // stop when both inputs are 0

 while(scanf("%d%d", &a, &b), (a || b))

 // scanf returns the number  of items read

 while(scanf("%d%d", &a, &b) == 2)

 // scanf until EOF

 while(scanf("%d%d", &a, &b) != EOF)

}

**A2 – Bitmask (HARUS ADA)**

#define isOn(S, j) (S & (1 << j))

#define setBit(S, j) (S |= (1 << j))

#define clearBit(S, j) (S &= ~(1 << j))

#define toggleBit(S, j) (S ^= (1 << j))

#define lowBit(S) (S & (-S))

#define setAll(S, n) (S = (1 << n) - 1)

#define modulo(S, N) ((S) & (N - 1))   // returns S % N, where N is a power of 2

#define isPowerOfTwo(S) (!(S & (S - 1)))

#define nearestPowerOfTwo(S) ((int)pow(2.0, (int)((log((double)S) / log(2.0)) + 0.5)))

#define turnOffLastBit(S) ((S) & (S - 1))

#define turnOnLastZero(S) ((S) | (S + 1))

#define turnOffLastConsecutiveBits(S) ((S) & (S + 1))

#define turnOnLastConsecutiveZeroes(S) ((S) | (S - 1))

#define LSOne(S) (S & (-S))

class FenwickTree {

 private:

   vector<long long int> ft;

   int n;

 public:

   FenwickTree(int \_n) : n(\_n) {

     ft.assign(n+1, 0);

   }

   FenwickTree(const vector<int>& f) : n(f.size() - 1) {

     ft.assign(n+1, 0);

     for(int i = 1; i <= n; ++i) {

       ft[i] += f[i];

       if(i + LSOne(i) <= n) {

         ft[i + LSOne(i)] += ft[i];

       }

     }

   }

   long long int rsq(int b) {

     long long int sum = 0;

     for(; b; b -= LSOne(b)) {

       sum += ft[b];

     }

     return sum;

   }

   long long int rsq(int a, int b) {

     return rsq(b) - rsq(a);

   }

   void adjust(int k, long long int v) {

     for(; k <=n; k += LSOne(k)) {

       ft[k] += v;

     }

   }

};

**A3 – UFDS (HARUS ADA)**

class UnionFind {

 public:

   vector<int> p, rank, setSize;

   int numSets;

 public:

   UnionFind(int N) {

     numSets = N;

     setSize.assign(N, 1);

     rank.assign(N, 0);

     p.assign(N, 0);

     for(int i = 0; i < N; ++i) {

       p[i] = i;

     }

   }

   int findSet(int i) {

     return (p[i] == i) ? i : (p[i] = findSet(p[i]));

   }

   bool isSameSet(int i, int j) {

     bool x = findSet(i) == findSet(j);

     return x;

   }

   void unionSet(int i, int j) {

     if(!isSameSet(i, j)) {

       numSets--;

       int x = findSet(i);

       int y = findSet(j);

       if(rank[x] > rank[y]) {

         p[y] = x; setSize[x] += setSize[y];

       } else {

         p[x] = y; setSize[y] += setSize[x];

         if(rank[x] == rank[y]) {

           rank[y]++;

         }

       }

     }

   }

   int numDisjoinSets() {

     return numSets;

   }

   int sizeOfSet(int i) {

     return setSize[findSet(i)];

   }

};

**A4 – Segment Tree (HARUS ADA)**

class SegmentTree {

   private:

     vi st, A;

     int n;

     int left(int p) {

       return p<<1;

     }

     int right(int p) {

       return (p<<1) + 1;

     }

     void build(int p, int L, int R) {

       if (L==R) {

         st[p] = L;

       } else {

         build(left(p), L, (L+R)/2);

         build(right(p), (L+R)/2 + 1, R);

         int p1 = st[left(p)];

         int p2 = st[right(p)];

         st[p] = (A[p1] <= A[p2]) ? p1 : p2;

       }

     }

     int rmq(int p, int L, int R, int i, int j) {

       if (i > R || j < L) {

         return -1;

       }

       if (L >= i && R <= j) {

         return st[p];

       }

       int p1 = rmq(left(p), L, (L+R)/2, i, j);

       int p2 = rmq(right(p), (L+R)/2 + 1, R, i, j);

       if (p1 == -1) {

         return p2;

       }

       if (p2 == -1) {

         return p1;

       }

       return (A[p1] <= A[p2]) ? p1 : p2;

     }

     int update(int p, int L, int R, int idx, int new\_value) {

       int i = idx;

       int j = idx;

       if (i > R || j < L) {

         return st[p];

       }

       if (L == i && R == j) {

         A[i] = new\_value;

         return st[p] = L;

       }

       int p1, p2;

       p1 = update(left(p), L, (L+R)/2, idx, new\_value);

       p2 = update(right(p), (L+R)/2+1, R, idx, new\_value);

       return st[p] = (A[p1] <= A[p2]) ? p1 : p2;

     }

   public:

     SegmentTree(const vi &\_A) {

       A = \_A;

       n = (int)A.size();

       st.assign(4\*n, 0);

       build(1, 0, n-1);

     }

     int rmq(int i, int j) {

       return rmq(1, 0, n-1, i, j);

     }

     int update(int i, int v) {

       return update(1, 0, n-1, i, v);

     }

   };

**A5 - Backtracking**

/\* 8 Queens Chess Problem \*/

#include <cstdlib>                       // we use the int version of 'abs'

#include <cstdio>

#include <cstring>

using namespace std;

int row[8], TC, a, b, lineCounter;            // ok to use global variables

bool place(int r, int c) {

 for (int prev = 0; prev < c; prev++)    // check previously placed queens

   if (row[prev] == r || (abs(row[prev] - r) == abs(prev - c)))

     return false;        // share same row or same diagonal -> infeasible

 return true; }

void backtrack(int c) {

 if (c == 8 && row[b] == a) {

// candidate sol, (a, b) has 1 queen

   printf("%2d %d", ++lineCounter, row[0] + 1);

   for (int j = 1; j < 8; j++) printf(" %d", row[j] + 1);

   printf("\n"); }

 for (int r = 0; r < 8; r++)                       // try all possible row

   if (place(r, c)) {          // if can place a queen at this col and row

     row[c] = r; backtrack(c + 1);

     // put this queen here and recurse

}   }

int main() {

 scanf("%d", &TC);

 while (TC--) {

   scanf("%d %d", &a, &b); a--; b--;         // switch to 0-based indexing

   memset(row, 0, sizeof row); lineCounter = 0;

   printf("SOLN       COLUMN\n");

   printf(" #      1 2 3 4 5 6 7 8\n\n");

   backtrack(0);           // generate all possible 8! candidate solutions

   if (TC) printf("\n");

} } // return 0;

**A6 -  DP Top Down**

/\* UVa 11450 - Wedding Shopping - Top Down \*/

// this code is similar to recursive backtracking code

// parts of the code specific to top-down DP are commented with: `TOP-DOWN'

// if these lines are commented, this top-down DP will become backtracking!

#include <algorithm>

#include <cstdio>

#include <cstring>

using namespace std;

int M, C, price[25][25];                 // price[g (<= 20)][model (<= 20)]

int memo[210][25];    // TOP-DOWN: dp table memo[money (<= 200)][g (<= 20)]

int shop(int money, int g) {

 if (money < 0) return -1000000000;     // fail, return a large -ve number

 if (g == C) return M - money;        // we have bought last garment, done

 if (memo[money][g] != -1) return memo[money][g]; // TOP-DOWN: memoization

 int ans = -1;   // start with a -ve number as all prices are non negative

 for (int model = 1; model <= price[g][0]; model++)      // try all models

   ans = max(ans, shop(money - price[g][model], g + 1));

 return memo[money][g] = ans; // TOP-DOWN: assign ans to table + return it

}

int main() {            // easy to code if you are already familiar with it

 int i, j, TC, score;

 scanf("%d", &TC);

 while (TC--) {

   scanf("%d %d", &M, &C);

   for (i = 0; i < C; i++) {

     scanf("%d", &price[i][0]);                  // store K in price[i][0]

     for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);

   }

   memset(memo, -1, sizeof memo);    // TOP-DOWN: initialize DP memo table

   score = shop(M, 0);                            // start the top-down DP

   if (score < 0) printf("no solution\n");

   else           printf("%d\n", score);

} } // return 0;

**A7 – DP Bottom Up**

/\* UVa 11450 - Wedding Shopping - Bottom Up \*/

#include <cstdio>

#include <cstring>

using namespace std;

int main() {

 int i, j, k, TC, M, C;

 int price[25][25];                     // price[g (<= 20)][model (<= 20)]

 bool reachable[25][210];    // reachable table[g (<= 20)][money (<= 200)]

 scanf("%d", &TC);

 while (TC--) {

   scanf("%d %d", &M, &C);

   for (i = 0; i < C; i++) {

     scanf("%d", &price[i][0]);               // we store K in price[i][0]

     for (j = 1; j <= price[i][0]; j++) scanf("%d", &price[i][j]);

   }

   memset(reachable, false, sizeof reachable);         // clear everything

   for (i = 1; i <= price[0][0]; i++)

      // initial values (base cases)

     if (M - price[0][i] >= 0)

     // to prevent array index out of bound

       reachable[0][M - price[0][i]] = true;

 // using first garment g = 0

   for (i = 1; i < C; i++)

// for each remaining garment

     for (j = 0; j < M; j++) if (reachable[i - 1][j]) // a reachable state

       for (k = 1; k <= price[i][0]; k++) if (j - price[i][k] >= 0)

         reachable[i][j - price[i][k]] = true;   // also a reachable state

   for (j = 0; j <= M && !reachable[C - 1][j]; j++); // the answer in here

   if (j == M + 1) printf("no solution\n");         // last row has on bit

   else            printf("%d\n", M - j);

} } // return 0;

**G1 - DFS**

typedef pair<int, int> ii;      // In this chapter, we will frequently use these

typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic

typedef vector<int> vi;   // but shortcuts are useful in competitive programming

#define DFS\_WHITE -1 // normal DFS, do not change this with other values (other than 0), because we usually use memset with conjunction with DFS\_WHITE

#define DFS\_BLACK 1

vector<vii> AdjList;

vi dfs\_num;     // this variable has to be global, we cannot put it in recursion

int numCC;

void dfs(int u) {          // DFS for normal usage: as graph traversal algorithm

 printf(" %d", u);                                    // this vertex is visited

 dfs\_num[u] = DFS\_BLACK;      // important step: we mark this vertex as visited

 for (int j = 0; j < (int)AdjList[u].size(); j++) {

   ii v = AdjList[u][j];                      // v is a (neighbor, weight) pair

   if (dfs\_num[v.first] == DFS\_WHITE)         // important check to avoid cycle

     dfs(v.first);      // recursively visits unvisited neighbors v of vertex u

} }

**G2 - BFS**

// inside int main() ---no recursion

 vi d(V, INF); d[s] = 0;

 queue<int> q; q.push(s);

 while(!q.empty()) {

   int u = q.front(); q.pop();

   for (auto v : AdjList[u]) {

     if (d[v.first] == INF) {

       d[v.first] = d[u]+1;

       q.push(v.first);

 } } }

**G3 – Kruskal and Prim**

#include <algorithm>

#include <cstdio>

#include <vector>

#include <queue>

using namespace std;

typedef pair<int, int> ii;

typedef vector<int> vi;

typedef vector<ii> vii;

// Union-Find Disjoint Sets Library written in OOP manner, using both path compression and union by rank heuristics

class UnionFind {                                              // OOP style

private:

 vi p, rank, setSize;                       // remember: vi is vector<int>

 int numSets;

public:

 UnionFind(int N) {

   setSize.assign(N, 1); numSets = N; rank.assign(N, 0);

   p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }

 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }

 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }

 void unionSet(int i, int j) {

   if (!isSameSet(i, j)) { numSets--;

   int x = findSet(i), y = findSet(j);

   // rank is used to keep the tree short

   if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }

   else                   { p[x] = y; setSize[y] += setSize[x];

                            if (rank[x] == rank[y]) rank[y]++; } } }

 int numDisjointSets() { return numSets; }

 int sizeOfSet(int i) { return setSize[findSet(i)]; }

};

vector<vii> AdjList;

vi taken;                                  // global boolean flag to avoid cycle

priority\_queue<ii> pq;            // priority queue to help choose shorter edges

void process(int vtx) {    // so, we use -ve sign to reverse the sort order

 taken[vtx] = 1;

 for (int j = 0; j < (int)AdjList[vtx].size(); j++) {

   ii v = AdjList[vtx][j];

   if (!taken[v.first]) pq.push(ii(-v.second, -v.first));

} }                                // sort by (inc) weight then by (inc) id

int main() {

 int V, E, u, v, w;

 /\*

 // Graph in Figure 4.10 left, format: list of weighted edges

 // This example shows another form of reading graph input

 5 7

 0 1 4

 0 2 4

 0 3 6

 0 4 6

 1 2 2

 2 3 8

 3 4 9

 \*/

 freopen("in\_03.txt", "r", stdin);

 scanf("%d %d", &V, &E);

 // Kruskal's algorithm merged with Prim's algorithm

 AdjList.assign(V, vii());

 vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the edge

 for (int i = 0; i < E; i++) {

   scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)

   EdgeList.push\_back(make\_pair(w, ii(u, v)));                // (w, u, v)

   AdjList[u].push\_back(ii(v, w));

   AdjList[v].push\_back(ii(u, w));

 }

 sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)

                     // note: pair object has built-in comparison function

 int mst\_cost = 0;

 UnionFind UF(V);                     // all V are disjoint sets initially

 for (int i = 0; i < E; i++) {                      // for each edge, O(E)

   pair<int, ii> front = EdgeList[i];

   if (!UF.isSameSet(front.second.first, front.second.second)) {  // check

     mst\_cost += front.first;                // add the weight of e to MST

     UF.unionSet(front.second.first, front.second.second);    // link them

 } }                       // note: the runtime cost of UFDS is very light

 // note: the number of disjoint sets must eventually be 1 for a valid MST

 printf("MST cost = %d (Kruskal's)\n", mst\_cost);

// inside int main() --- assume the graph is stored in AdjList, pq is empty

 taken.assign(V, 0);                // no vertex is taken at the beginning

 process(0);   // take vertex 0 and process all edges incident to vertex 0

 mst\_cost = 0;

 while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken

   ii front = pq.top(); pq.pop();

   u = -front.second, w = -front.first;  // negate the id and weight again

   if (!taken[u])                 // we have not connected this vertex yet

     mst\_cost += w, process(u); // take u, process all edges incident to u

 }                                       // each edge is in pq only once!

 printf("MST cost = %d (Prim's)\n", mst\_cost);

 return 0;

}

**G4 - Dijkstra**

int main() {

 int V, E, s, u, v, w;

 vector<vii> AdjList;

 AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList

 for (int i = 0; i < E; i++) {

   scanf("%d %d %d", &u, &v, &w);

   AdjList[u].push\_back(ii(v, w));                              // directed graph

 }

 // Dijkstra routine

 vi dist(V, INF); dist[s] = 0;                    // INF = 1B to avoid overflow

 priority\_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));

                            // ^to sort the pairs by increasing distance from s

 while (!pq.empty()) {                                             // main loop

   ii front = pq.top(); pq.pop();     // greedy: pick shortest unvisited vertex

   int d = front.first, u = front.second;

   if (d > dist[u]) continue;   // this check is important, see the explanation

   for (int j = 0; j < (int)AdjList[u].size(); j++) {

     ii v = AdjList[u][j];                       // all outgoing edges from u

     if (dist[u] + v.second < dist[v.first]) {

       dist[v.first] = dist[u] + v.second;                 // relax operation

       pq.push(ii(dist[v.first], v.first));

 } } }  // note: this variant can cause duplicate items in the priority queue

 for (int i = 0; i < V; i++) // index + 1 for final answer

   printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

 return 0;

}

**G5 – Bellman Ford**

int main() {

 int V, E, s, a, b, w;

 vector<vii> AdjList;

 AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList

 for (int i = 0; i < E; i++) {

   scanf("%d %d %d", &a, &b, &w);

   AdjList[a].push\_back(ii(b, w));

 }

 // Bellman Ford routine

 vi dist(V, INF); dist[s] = 0;

 for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)

   for (int u = 0; u < V; u++)                        // these two loops = O(E)

     for (int j = 0; j < (int)AdjList[u].size(); j++) {

       ii v = AdjList[u][j];        // we can record SP spanning here if needed

       dist[v.first] = min(dist[v.first], dist[u] + v.second);         // relax

     }

 bool hasNegativeCycle = false;

 for (int u = 0; u < V; u++)                          // one more pass to check

   for (int j = 0; j < (int)AdjList[u].size(); j++) {

     ii v = AdjList[u][j];

     if (dist[v.first] > dist[u] + v.second)                 // should be false

       hasNegativeCycle = true;     // but if true, then negative cycle exists!

   }

 printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

 if (!hasNegativeCycle)

   for (int i = 0; i < V; i++)

     printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

 return 0;

}

**G6 – Dinic (HARUS ADA)**

#include <iostream>

#include <stdio.h>

#include <algorithm>

#include <math.h>

#include <string>

#include <stack>

#include <queue>

#include <tuple>

using namespace std;

typedef long long LL;

struct Edge {

 int u, v;

 LL cap, flow;

 Edge() {}

 Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}

};

struct Dinic {

 int N;

 vector<int> sourceSet;

 vector<Edge> E;

 vector<vector<int>> g;

 vector<int> d, pt;

 Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

 void AddEdge(int u, int v, LL cap) {

   if (u != v) {

     E.emplace\_back(Edge(u, v, cap));

     g[u].emplace\_back(E.size() - 1);

     E.emplace\_back(Edge(v, u, 0));

     g[v].emplace\_back(E.size() - 1);

   }

 }

 bool BFS(int S, int T) {

   queue<int> q({S});

   fill(d.begin(), d.end(), N + 1);

   d[S] = 0;

   while(!q.empty()) {

     int u = q.front(); q.pop();

     if (u == T) break;

     for (int k: g[u]) {

       Edge &e = E[k];

       if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {

         d[e.v] = d[e.u] + 1;

         q.emplace(e.v);

       }

     }

   }

   return d[T] != N + 1;

 }

 LL DFS(int u, int T, LL flow = -1) {

   if (u == T || flow == 0) return flow;

   for (int &i = pt[u]; i < g[u].size(); ++i) {

     Edge &e = E[g[u][i]];

     Edge &oe = E[g[u][i]^1];

     if (d[e.v] == d[e.u] + 1) {

       LL amt = e.cap - e.flow;

       if (flow != -1 && amt > flow) amt = flow;

       if (LL pushed = DFS(e.v, T, amt)) {

         e.flow += pushed;

         oe.flow -= pushed;

         return pushed;

       }

     }

   }

   return 0;

 }

 LL MaxFlow(int S, int T) {

   LL total = 0;

   while (BFS(S, T)) {

     fill(pt.begin(), pt.end(), 0);

     while (LL flow = DFS(S, T))

       total += flow;

   }

   return total;

 }

 int numUsedEdge() {

   int ans = 0;

   for(vector<Edge>::iterator iter = E.begin(); iter != E.end(); ++iter) {

     if(iter->flow > 0) {

       ans++;

     }

   }

   return ans;

 }

 void printFlow() {

   vector< tuple<int, int, long long int> > temp;

   for(vector<Edge>::iterator iter = E.begin(); iter != E.end(); ++iter) {

     if(iter->flow > 0) {

       temp.push\_back(tuple<int, int, long long int>(iter->u, iter->v, iter->flow));

     }

   }

   sort(temp.begin(), temp.end());

   for(vector< tuple<int, int, long long int> >::iterator iter = temp.begin(); iter != temp.end(); ++iter) {

     printf("%d %d %lld\n", get<0>(\*iter), get<1>(\*iter), get<2>(\*iter));

   }

 }

 void sourceSetProcess(int S) {

   int sourceCheck[N];

   for(int i = 0; i < N; ++i) {

     sourceCheck[i] = 0;

   }

   queue<int> q;

   q.push(S);

   sourceCheck[S] = 1;

   sourceSet.push\_back(S);

   while(!q.empty()) {

     int u = q.front();

     q.pop();

     for (int k: g[u]) {

       Edge &e = E[k];

       if(sourceCheck[e.v] == 0) {

         if (e.cap - e.flow > 0) {

           sourceCheck[e.v] = 1;

           sourceSet.push\_back(e.v);

           q.push(e.v);

         }

       }

     }

   }

 }

 int sourceSetSize() {

   return sourceSet.size();

 }

 void printSourceSet() {

   for(int i = 0; i < sourceSetSize(); ++i) {

     printf("%d\n", sourceSet[i]);

   }

 }

};

int main() {

 int n, m, s, t;

 scanf("%d %d %d %d", &n, &m, &s, &t);

 Dinic dinic(n);

 int u, v;

 LL c;

 while(m--) {

   scanf("%d %d %lld", &u, &v, &c);

   dinic.AddEdge(u, v, c);

 }

 long long int answer = dinic.MaxFlow(s, t);

 printf("%d %lld %d\n", n, answer, dinic.numUsedEdge());

 dinic.printFlow();

 // to get mincut

 // do sourceSetProcess (after MaxFlow)

 // the answer is the sourceSetSize

}

**G7 – MCBM (HARUS ADA)**

vector<vi> AdjList;

vi match, vis;                                          // global variables

int Aug(int l) {                 // return 1 if an augmenting path is found

 if (vis[l]) return 0;                               // return 0 otherwise

 vis[l] = 1;

 for (int j = 0; j < (int)AdjList[l].size(); j++) {

   int r = AdjList[l][j];

   if (match[r] == -1 || Aug(match[r])) {

     match[r] = l; return 1;                           // found 1 matching

 } }

 return 0;                                                  // no matching

}

int main() {

 int V = 5, Vleft = 3;                               // we ignore vertex 0

 AdjList.assign(V, vi());

 AdjList[1].push\_back(3); AdjList[1].push\_back(4);

 AdjList[2].push\_back(3);

 int MCBM = 0;

 match.assign(V, -1);    // V is the number of vertices in bipartite graph

 for (int l = 0; l < Vleft; l++) {         // Vleft = size of the left set

   vis.assign(Vleft, 0);                    // reset before each recursion

   MCBM += Aug(l);

 }

 printf("Found %d matchings\n", MCBM);  // the answer is 2 for Figure 4.42

 return 0;

}

**M1 - BigInteger**

import java.io.\*;

import java.util.\*;

import java.util.Scanner;

import java.math.BigInteger;

class BigInteger2 {

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

   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

   PrintWriter pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)));

   while(true) {

     String N = br.readLine();

     if(N == null) break;

     BigInteger BN = new BigInteger(N);

     BigInteger BRN = new BigInteger(

       new StringBuffer(BN.toString()).reverse().toString());

     pw.printf("%s is ", N);

     if(!BN.isProbablePrime(10)) // 10 is enough

       pw.printf("not prime.\n";

     else if (!BN.equals(BRN) && BRN.isProbablePrime(10))

       pw.printf("emirp.\n")

     else

       pw.printf("prime.\n")

   }

   pw.close();

 }

}

import java.util.Scanner; // Scanner class is inside package java.util

import java.math.BigInteger; // BigInteger class is inside package java.math

class Main { /\* UVa 10925 - Krakovia, 0.732s in Java \*/

 public static void main(String[] args) {

   Scanner sc = new Scanner(System.in);

   int caseNo = 1;

   while (true) {

     int N = sc.nextInt(), F = sc.nextInt(); // N bills, F friends

     if (N == 0 && F == 0) break;

     BigInteger sum = BigInteger.ZERO; // BigInteger has this constant ZERO

     for (int i = 0; i < N; i++) { // sum the N large bills

       BigInteger V = sc.nextBigInteger(); // for reading next BigInteger!

       sum = sum.add(V); // this is BigInteger addition

     }

     System.out.println("Bill #" + (caseNo++) + " costs " +

       sum + ": each friend should pay " + sum.divide(BigInteger.valueOf(F)));

     System.out.println(); // the line above is BigInteger division

} } }                       // divide the large sum to F friends

**M2 – Prime**

ll \_sieve\_size;

bitset<10000010> bs;   // 10^7 should be enough for most cases

vi primes;   // compact list of primes in form of vector<int>

// first part

void sieve(ll upperbound) {          // create list of primes in [0..upperbound]

 \_sieve\_size = upperbound + 1;                   // add 1 to include upperbound

 bs.set();                                                 // set all bits to 1

 bs[0] = bs[1] = 0;                                     // except index 0 and 1

 for (ll i = 2; i <= \_sieve\_size; i++) if (bs[i]) {

   // cross out multiples of i starting from i \* i!

   for (ll j = i \* i; j <= \_sieve\_size; j += i) bs[j] = 0;

   primes.push\_back(i);  // also add this vector containing list of primes

} }                                           // call this method in main method

bool isPrime(ll N) {                 // a good enough deterministic prime tester

 if (N < \_sieve\_size) return bs[N];                   // O(1) for small primes

 for (int i = 0; i < (int)primes.size(); i++)

   if (N % primes[i] == 0) return false;

 return true;                    // it takes longer time if N is a large prime!

}                      // note: only work for N <= (last prime in vi "primes")^2

// second part

vi primeFactors(ll N) {   // remember: vi is vector of integers, ll is long long

 vi factors;                    // vi `primes' (generated by sieve) is optional

 ll PF\_idx = 0, PF = primes[PF\_idx];     // using PF = 2, 3, 4, ..., is also ok

 while (N != 1 && (PF \* PF <= N)) {   // stop at sqrt(N), but N can get smaller

   while (N % PF == 0) { N /= PF; factors.push\_back(PF); }    // remove this PF

   PF = primes[++PF\_idx];                              // only consider primes!

 }

 if (N != 1) factors.push\_back(N);     // special case if N is actually a prime

 return factors;         // if pf exceeds 32-bit integer, you have to change vi

}

// third part

ll numPF(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

 while (N != 1 && (PF \* PF <= N)) {

   while (N % PF == 0) { N /= PF; ans++; }

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans++;

 return ans;

}

ll numDiffPF(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

 while (N != 1 && (PF \* PF <= N)) {

   if (N % PF == 0) ans++;                           // count this pf only once

   while (N % PF == 0) N /= PF;

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans++;

 return ans;

}

ll sumPF(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

 while (N != 1 && (PF \* PF <= N)) {

   while (N % PF == 0) { N /= PF; ans += PF; }

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans += N;

 return ans;

}

ll numDiv(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1;             // start from ans = 1

 while (N != 1 && (PF \* PF <= N)) {

   ll power = 0;                                             // count the power

   while (N % PF == 0) { N /= PF; power++; }

   ans \*= (power + 1);                              // according to the formula

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans \*= 2;             // (last factor has pow = 1, we add 1 to it)

 return ans;

}

ll sumDiv(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1;             // start from ans = 1

 while (N != 1 && (PF \* PF <= N)) {

   ll power = 0;

   while (N % PF == 0) { N /= PF; power++; }

   ans \*= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);         // formula

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans \*= ((ll)pow((double)N, 2.0) - 1) / (N - 1);        // last one

 return ans;

}

ll EulerPhi(ll N) {

 ll PF\_idx = 0, PF = primes[PF\_idx], ans = N;             // start from ans = N

 while (N != 1 && (PF \* PF <= N)) {

   if (N % PF == 0) ans -= ans / PF;                // only count unique factor

   while (N % PF == 0) N /= PF;

   PF = primes[++PF\_idx];

 }

 if (N != 1) ans -= ans / N;                                     // last factor

 return ans;

}

**M3 – Pollard Rho**  
  
#include <cstdio>

using namespace std;

#define abs\_val(a) (((a)>=0)?(a):-(a))

typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // returns (a \* b) % c, and minimize overflow

 ll x = 0, y = a % c;

 while (b > 0) {

   if (b % 2 == 1) x = (x + y) % c;

   y = (y \* 2) % c;

   b /= 2;

 }

 return x % c;

}

ll gcd(ll a, ll b) { return !b ? a : gcd(b, a % b); }       // standard gcd

ll pollard\_rho(ll n) {

 int i = 0, k = 2;

 ll x = 3, y = 3;                // random seed = 3, other values possible

 while (1) {

   i++;

   x = (mulmod(x, x, n) + n - 1) % n;               // generating function

   ll d = gcd(abs\_val(y - x), n);                       // the key insight

   if (d != 1 && d != n) return d;         // found one non-trivial factor

   if (i == k) y = x, k \*= 2;

} }

int main() {

 ll n = 2063512844981574047LL;    // we assume that n is not a large prime

 ll ans = pollard\_rho(n);          // break n into two non trivial factors

 if (ans > n / ans) ans = n / ans;          // make ans the smaller factor

 printf("%lld %lld\n", ans, n / ans);  // should be: 1112041493 1855607779

} // return 0;

**M4 – Cycle Finding**

// Pseudo-Random Numbers

#include <cstdio>

#include <iostream>

using namespace std;

typedef pair<int, int> ii;

int caseNo = 1, Z, I, M, L;

int f(int x) { return (Z \* x + I) % M; }

ii floydCycleFinding(int x0) {  // function int f(int x) is defined earlier

 // 1st part: finding k\*mu, hare's speed is 2x tortoise's

 int tortoise = f(x0), hare = f(f(x0));    // f(x0) is the node next to x0

 while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }

 // 2nd part: finding mu, hare and tortoise move at the same speed

 int mu = 0; hare = x0;

 while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++; }

 // 3rd part: finding lambda, hare moves, tortoise stays

 int lambda = 1; hare = f(tortoise);

 while (tortoise != hare) { hare = f(hare); lambda++; }

 return ii(mu, lambda);

}

int main() {

 while (scanf("%d %d %d %d", &Z, &I, &M, &L), (Z || I || M || L)) {

   ii result = floydCycleFinding(L);

   printf("Case %d: %d\n", caseNo++, result.second);

 }

 return 0;

}

**O1 – Points and Lines**

#include <algorithm>

#include <cstdio>

#include <cmath>

#include <vector>

using namespace std;

#define INF 1e9

#define EPS 1e-9

#define PI acos(-1.0) // important constant; alternative #define PI (2.0 \* acos(0.0))`

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

// struct point\_i { int x, y; };

struct point\_i { int x, y;

 point\_i() { x = y = 0; }                      // default constructor

 point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} };         // user-defined

struct point { double x, y;   // more precision

 point() { x = y = 0.0; }                      // default constructor

 point(double \_x, double \_y) : x(\_x), y(\_y) {}        // user-defined

 bool operator < (point other) const { // override less than operator

   if (fabs(x - other.x) > EPS)                 // useful for sorting

     return x < other.x;          // first criteria , by x-coordinate

   return y < other.y; }          // second criteria, by y-coordinate

 // use EPS (1e-9) when testing equality of two floating points

 bool operator == (point other) const {

  return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                // Euclidean distance

return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)

point rotate(point p, double theta) {

 double rad = DEG\_to\_RAD(theta);    // multiply theta with PI / 180.0

 return point(p.x \* cos(rad) - p.y \* sin(rad),

              p.x \* sin(rad) + p.y \* cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)

void pointsToLine(point p1, point p2, line &l) {

 if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine

   l.a = 1.0;   l.b = 0.0; l.c = -p1.x;           // default values

 } else {

   l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);

   l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0

   l.c = -(double)(l.a \* p1.x) - p1.y;

} }

// not needed since we will use the more robust form: ax + by + c = 0 (see above)

struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {

if (abs(p1.x - p2.x) < EPS) {          // special case: vertical line

  l.m = INF;                    // l contains m = INF and c = x\_value

  l.c = p1.x;                  // to denote vertical line x = x\_value

  return 0;   // we need this return variable to differentiate result

}

else {

  l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);

  l.c = p1.y - l.m \* p1.x;

  return 1;     // l contains m and c of the line equation y = mx + c

} }

bool areParallel(line l1, line l2) {       // check coefficients a & b

 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c

 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect

bool areIntersect(line l1, line l2, point &p) {

 if (areParallel(l1, l2)) return false;            // no intersection

 // solve system of 2 linear algebraic equations with 2 unknowns

 p.x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

 // special case: test for vertical line to avoid division by zero

 if (fabs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

 else                  p.y = -(l2.a \* p.x + l2.c);

 return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector

 vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b

 return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]

 return vec(v.x \* s, v.y \* s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v

 return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line

void pointSlopeToLine(point p, double m, line &l) {

 l.a = -m;                                               // always -m

 l.b = 1;                                                 // always 1

 l.c = -((l.a \* p.x) + (l.b \* p.y)); }                // compute this

void closestPoint(line l, point p, point &ans) {

 line perpendicular;         // perpendicular to l and pass through p

 if (fabs(l.b) < EPS) {              // special case 1: vertical line

   ans.x = -(l.c);   ans.y = p.y; return; }

 if (fabs(l.a) < EPS) {            // special case 2: horizontal line

   ans.x = p.x;      ans.y = -(l.c); return; }

 pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line

 // intersect line l with this perpendicular line

 // the intersection point is the closest point

 areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line

void reflectionPoint(line l, point p, point &ans) {

 point b;

 closestPoint(l, p, b);                     // similar to distToLine

 vec v = toVec(p, b);                             // create a vector

 ans = translate(translate(p, v), v); }         // translate p twice

double dot(vec a, vec b) { return (a.x \* b.x + a.y \* b.y); }

double norm\_sq(vec v) { return v.x \* v.x + v.y \* v.y; }

// returns the distance from p to the line defined by

// two points a and b (a and b must be different)

// the closest point is stored in the 4th parameter (byref)

double distToLine(point p, point a, point b, point &c) {

 // formula: c = a + u \* ab

 vec ap = toVec(a, p), ab = toVec(a, b);

 double u = dot(ap, ab) / norm\_sq(ab);

 c = translate(a, scale(ab, u));                  // translate a to c

 return dist(p, c); }           // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by

// two points a and b (still OK if a == b)

// the closest point is stored in the 4th parameter (byref)

double distToLineSegment(point p, point a, point b, point &c) {

 vec ap = toVec(a, p), ab = toVec(a, b);

 double u = dot(ap, ab) / norm\_sq(ab);

 if (u < 0.0) { c = point(a.x, a.y);                   // closer to a

   return dist(p, a); }         // Euclidean distance between p and a

 if (u > 1.0) { c = point(b.x, b.y);                   // closer to b

   return dist(p, b); }         // Euclidean distance between p and b

 return distToLine(p, a, b, c); }          // run distToLine as above

double angle(point a, point o, point b) {  // returns angle aob in rad

 vec oa = toVec(o, a), ob = toVec(o, b);

 return acos(dot(oa, ob) / sqrt(norm\_sq(oa) \* norm\_sq(ob))); }

double cross(vec a, vec b) { return a.x \* b.y - a.y \* b.x; }

//// another variant

//int area2(point p, point q, point r) { // returns 'twice' the area of this triangle A-B-c

//  return p.x \* q.y - p.y \* q.x +

//         q.x \* r.y - q.y \* r.x +

//         r.x \* p.y - r.y \* p.x;

//}

// note: to accept collinear points, we have to change the `> 0'

// returns true if point r is on the left side of line pq

bool ccw(point p, point q, point r) {

 return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq

bool collinear(point p, point q, point r) {

 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

**O2 – Circles**

#include <cstdio>

#include <cmath>

using namespace std;

#define INF 1e9

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point\_i { int x, y;     // whenever possible, work with point\_i

 point\_i() { x = y = 0; }                      // default constructor

 point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} };          // constructor

struct point { double x, y;   // only used if more precision is needed

 point() { x = y = 0.0; }                      // default constructor

 point(double \_x, double \_y) : x(\_x), y(\_y) {} };      // constructor

int insideCircle(point\_i p, point\_i c, int r) { // all integer version

 int dx = p.x - c.x, dy = p.y - c.y;

 int Euc = dx \* dx + dy \* dy, rSq = r \* r;             // all integer

 return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

bool circle2PtsRad(point p1, point p2, double r, point &c) {

 double d2 = (p1.x - p2.x) \* (p1.x - p2.x) +

             (p1.y - p2.y) \* (p1.y - p2.y);

 double det = r \* r / d2 - 0.25;

 if (det < 0.0) return false;

 double h = sqrt(det);

 c.x = (p1.x + p2.x) \* 0.5 + (p1.y - p2.y) \* h;

 c.y = (p1.y + p2.y) \* 0.5 + (p2.x - p1.x) \* h;

 return true; }         // to get the other center, reverse p1 and p2

**O3 – Triangles**

#include <cstdio>

#include <cmath>

using namespace std;

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point\_i { int x, y;     // whenever possible, work with point\_i

 point\_i() { x = y = 0; }                      // default constructor

 point\_i(int \_x, int \_y) : x(\_x), y(\_y) {} };          // constructor

struct point { double x, y;   // only used if more precision is needed

 point() { x = y = 0.0; }                      // default constructor

 point(double \_x, double \_y) : x(\_x), y(\_y) {} };      // constructor

double dist(point p1, point p2) {

 return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double ca) {

 return ab + bc + ca; }

double perimeter(point a, point b, point c) {

 return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {

 // Heron's formula, split sqrt(a \* b) into sqrt(a) \* sqrt(b); in implementation

 double s = 0.5 \* perimeter(ab, bc, ca);

 return sqrt(s) \* sqrt(s - ab) \* sqrt(s - bc) \* sqrt(s - ca); }

double area(point a, point b, point c) {

 return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================

// from ch7\_01\_points\_lines

struct line { double a, b, c; }; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)

void pointsToLine(point p1, point p2, line &l) {

 if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine

   l.a = 1.0;   l.b = 0.0; l.c = -p1.x;           // default values

 } else {

   l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);

   l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0

   l.c = -(double)(l.a \* p1.x) - p1.y;

} }

bool areParallel(line l1, line l2) {        // check coefficient a + b

 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

// returns true (+ intersection point) if two lines are intersect

bool areIntersect(line l1, line l2, point &p) {

 if (areParallel(l1, l2)) return false;            // no intersection

 // solve system of 2 linear algebraic equations with 2 unknowns

 p.x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

 // special case: test for vertical line to avoid division by zero

 if (fabs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

 else                  p.y = -(l2.a \* p.x + l2.c);

 return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector

 vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b

 return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]

 return vec(v.x \* s, v.y \* s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v

 return point(p.x + v.x , p.y + v.y); }

//====================================================================

double rInCircle(double ab, double bc, double ca) {

 return area(ab, bc, ca) / (0.5 \* perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {

 return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is an inCircle center, returns 0 otherwise

// if this function returns 1, ctr will be the inCircle center

// and r is the same as rInCircle

int inCircle(point p1, point p2, point p3, point &ctr, double &r) {

 r = rInCircle(p1, p2, p3);

 if (fabs(r) < EPS) return 0;                   // no inCircle center

 line l1, l2;                    // compute these two angle bisectors

 double ratio = dist(p1, p2) / dist(p1, p3);

 point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));

 pointsToLine(p1, p, l1);

 ratio = dist(p2, p1) / dist(p2, p3);

 p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));

 pointsToLine(p2, p, l2);

 areIntersect(l1, l2, ctr);           // get their intersection point

 return 1; }

double rCircumCircle(double ab, double bc, double ca) {

 return ab \* bc \* ca / (4.0 \* area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {

 return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written

// returns 1 if there is a circumCenter center, returns 0 otherwise

// if this function returns 1, ctr will be the circumCircle center

// and r is the same as rCircumCircle

int circumCircle(point p1, point p2, point p3, point &ctr, double &r){

 double a = p2.x - p1.x, b = p2.y - p1.y;

 double c = p3.x - p1.x, d = p3.y - p1.y;

 double e = a \* (p1.x + p2.x) + b \* (p1.y + p2.y);

 double f = c \* (p1.x + p3.x) + d \* (p1.y + p3.y);

 double g = 2.0 \* (a \* (p3.y - p2.y) - b \* (p3.x - p2.x));

 if (fabs(g) < EPS) return 0;

 ctr.x = (d\*e - b\*f) / g;

 ctr.y = (a\*f - c\*e) / g;

 r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points

 return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c

int inCircumCircle(point a, point b, point c, point d) {

 return (a.x - d.x) \* (b.y - d.y) \* ((c.x - d.x) \* (c.x - d.x) + (c.y - d.y) \* (c.y - d.y)) +

        (a.y - d.y) \* ((b.x - d.x) \* (b.x - d.x) + (b.y - d.y) \* (b.y - d.y)) \* (c.x - d.x) +

        ((a.x - d.x) \* (a.x - d.x) + (a.y - d.y) \* (a.y - d.y)) \* (b.x - d.x) \* (c.y - d.y) -

        ((a.x - d.x) \* (a.x - d.x) + (a.y - d.y) \* (a.y - d.y)) \* (b.y - d.y) \* (c.x - d.x) -

        (a.y - d.y) \* (b.x - d.x) \* ((c.x - d.x) \* (c.x - d.x) + (c.y - d.y) \* (c.y - d.y)) -

        (a.x - d.x) \* ((b.x - d.x) \* (b.x - d.x) + (b.y - d.y) \* (b.y - d.y)) \* (c.y - d.y) > 0 ? 1 : 0;

}

bool canFormTriangle(double a, double b, double c) {

 return (a + b > c) && (a + c > b) && (b + c > a); }

**O4 – Polygon**

#include <algorithm>

#include <cstdio>

#include <cmath>

#include <stack>

#include <vector>

using namespace std;

#define EPS 1e-9

#define PI acos(-1.0)

double DEG\_to\_RAD(double d) { return d \* PI / 180.0; }

double RAD\_to\_DEG(double r) { return r \* 180.0 / PI; }

struct point { double x, y;   // only used if more precision is needed

 point() { x = y = 0.0; }                      // default constructor

 point(double \_x, double \_y) : x(\_x), y(\_y) {}        // user-defined

 bool operator == (point other) const {

  return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y;  // name: `vec' is different from STL vector

 vec(double \_x, double \_y) : x(\_x), y(\_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b

 return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) {                // Euclidean distance

 return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// returns the perimeter, which is the sum of Euclidian distances

// of consecutive line segments (polygon edges)

double perimeter(const vector<point> &P) {

 double result = 0.0;

 for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]

   result += dist(P[i], P[i+1]);

 return result; }

// returns the area, which is half the determinant

double area(const vector<point> &P) {

 double result = 0.0, x1, y1, x2, y2;

 for (int i = 0; i < (int)P.size()-1; i++) {

   x1 = P[i].x; x2 = P[i+1].x;

   y1 = P[i].y; y2 = P[i+1].y;

   result += (x1 \* y2 - x2 \* y1);

 }

 return fabs(result) / 2.0; }

double dot(vec a, vec b) { return (a.x \* b.x + a.y \* b.y); }

double norm\_sq(vec v) { return v.x \* v.x + v.y \* v.y; }

double angle(point a, point o, point b) {  // returns angle aob in rad

 vec oa = toVec(o, a), ob = toVec(o, b);

 return acos(dot(oa, ob) / sqrt(norm\_sq(oa) \* norm\_sq(ob))); }

double cross(vec a, vec b) { return a.x \* b.y - a.y \* b.x; }

// note: to accept collinear points, we have to change the `> 0'

// returns true if point r is on the left side of line pq

bool ccw(point p, point q, point r) {

 return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq

bool collinear(point p, point q, point r) {

 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining

// all the edges of the polygon one by one

bool isConvex(const vector<point> &P) {

 int sz = (int)P.size();

 if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex

 bool isLeft = ccw(P[0], P[1], P[2]);               // remember one result

 for (int i = 1; i < sz-1; i++)            // then compare with the others

   if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)

     return false;            // different sign -> this polygon is concave

 return true; }                                  // this polygon is convex

// returns true if point p is in either convex/concave polygon P

bool inPolygon(point pt, const vector<point> &P) {

 if ((int)P.size() == 0) return false;

 double sum = 0;    // assume the first vertex is equal to the last vertex

 for (int i = 0; i < (int)P.size()-1; i++) {

   if (ccw(pt, P[i], P[i+1]))

        sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw

   else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw

 return fabs(fabs(sum) - 2\*PI) < EPS; }

// line segment p-q intersect with line A-B.

point lineIntersectSeg(point p, point q, point A, point B) {

 double a = B.y - A.y;

 double b = A.x - B.x;

 double c = B.x \* A.y - A.x \* B.y;

 double u = fabs(a \* p.x + b \* p.y + c);

 double v = fabs(a \* q.x + b \* q.y + c);

 return point((p.x \* v + q.x \* u) / (u+v), (p.y \* v + q.y \* u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b

// (note: the last point must be the same as the first point)

vector<point> cutPolygon(point a, point b, const vector<point> &Q) {

 vector<point> P;

 for (int i = 0; i < (int)Q.size(); i++) {

   double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;

   if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));

   if (left1 > -EPS) P.push\_back(Q[i]);       // Q[i] is on the left of ab

   if (left1 \* left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab

     P.push\_back(lineIntersectSeg(Q[i], Q[i+1], a, b));

 }

 if (!P.empty() && !(P.back() == P.front()))

   P.push\_back(P.front());        // make P's first point = P's last point

 return P; }

point pivot;

bool angleCmp(point a, point b) {                 // angle-sorting function

 if (collinear(pivot, a, b))                               // special case

   return dist(pivot, a) < dist(pivot, b);    // check which one is closer

 double d1x = a.x - pivot.x, d1y = a.y - pivot.y;

 double d2x = b.x - pivot.x, d2y = b.y - pivot.y;

 return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles

vector<point> CH(vector<point> P) {   // the content of P may be reshuffled

 int i, j, n = (int)P.size();

 if (n <= 3) {

   if (!(P[0] == P[n-1])) P.push\_back(P[0]); // safeguard from corner case

   return P;                           // special case, the CH is P itself

 }

 // first, find P0 = point with lowest Y and if tie: rightmost X

 int P0 = 0;

 for (i = 1; i < n; i++)

   if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))

     P0 = i;

 point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

 // second, sort points by angle w.r.t. pivot P0

 pivot = P[0];                    // use this global variable as reference

 sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]

 // third, the ccw tests

 vector<point> S;

 S.push\_back(P[n-1]); S.push\_back(P[0]); S.push\_back(P[1]);   // initial S

 i = 2;                                         // then, we check the rest

 while (i < n) {           // note: N must be >= 3 for this method to work

   j = (int)S.size()-1;

   if (ccw(S[j-1], S[j], P[i])) S.push\_back(P[i++]);  // left turn, accept

   else S.pop\_back(); }   // or pop the top of S until we have a left turn

 return S; }                                          // return the result

int main() {

 // 6 points, entered in counter clockwise order, 0-based indexing

 vector<point> P;

}  
--------------------------------

**SHK C++ Template**

#include <bits/stdc++.h>

using namespace std;

 const double PI = acos(-1);

 #define INF 1E9

 #define endl '\n'

 #define pb push\_back

 // A lot of typedefs

   typedef long long ll;

 // struct

   struct mystruct {

     int counter;

   };

 //custom hashing

   struct custom\_hash {

     inline std::size\_t operator()(const std::pair<int,int> & v) const {

         return v.first\*31+v.second\*7;

     }

   };

 // pq/set custom comparator, will get reversed (at least in pq)

   class mycomp {

     public:

     bool operator() (mystruct a, mystruct b) {

       return a.counter > b.counter;

     }

   };

 // sort custom comparator

   bool customcompare(mystruct a, mystruct b) {

     return a.counter > b.counter;

   }

int main ()

{

 ios::sync\_with\_stdio(false);

 cin.tie(nullptr);

 return 0;

}

**Python input (easy)**

import sys

class SamInput(object):

 def \_\_init\_\_(self):

   self.inp = []

   for i in sys.stdin:

     i = i.replace("\n", "")

     j = list(i.split())

     self.inp.append(j)

 def readln(self):

   if (len(self.inp) == 0) :

     return False

   else:

     return str.join(" ", self.inp.pop(0))

 def read(self):

   if (len(self.inp) == 0):

     return False

   while (len(self.inp[0]) == 0):

     self.inp.pop(0)

     if (len(self.inp) == 0):

       return False

   return self.inp[0].pop(0)

|  |  |
| --- | --- |
| Comment highlighted code | Ctrl + Shift + C |
| Uncomment highlighted code | Ctrl + Shift + X |

|  |  |
| --- | --- |
| Indent block. | Tab |
| Dedent block. | Shift + Tab |

|  |  |
| --- | --- |
| Line cut. | Ctrl + L |
| Line copy. | Ctrl + Shift + T |

g++ x.cpp -std=c++11 -o x

x.exe < test.txt







