```
Index of programming contest code library

Howard Cheng

Arithmetic:
  bigint:
    Big (signed) integer arithmetic
  bignumber.java:
    Java template for using large integer arithmetic (BigInteger)
  binomial:
    Computes binomial coefficients
  cra:
    Chinese remainder theorem
  diophantine_sys:
    Linear system of diophantine equations (works for single equation too!)
  euclid:
    Euclidean algorithm
  eulerphi:
    Computes the Euler phi (totient) function: given a positive n, return
    the number of integers between 1 and n relatively prime to n.
  exteuclid:
    Extended Euclidean algorithm
  exp:
    Fast exponentiation
  expmod:
    Fast exponentiation mod m
  factor:
    Integer prime factorization
  factor_large:
    Integer prime factorization for larger integers (>= 2^40)
  fflinsolve:
    Fraction-free solution of linear systems of equations (for systems
    with integer coefficients)
  fib:
    Computes n-th Fibonacci number with O(log n) complexity
  frac2dec:
    Obtain the decimal representation of a fraction.
  fraction:
    A rational number class.
  infix:
    Parses and evaluates infix arithmetic expressions.
  int_mult:
    Multiply integer factors on the numerator, divide by the integer
    factors in the denominator without overflow.
  linsolve:
    Solves linear systems of equations with LU decomposition.
  mult:
    Multiply factors on the numerator, divide by the factors in the
    denominator without overflow.
  ratlinsolve:
    Rational solution of linear systems of equations (can be solved by
    fflinsolve as well).
  roman_numerals
    Converts between Arabic and Roman numerals.

Geometric (mostly 2-D):
  areapoly:
    Computes the signed area of a simple (no self-intersection) polygon.
  ccw:
    Determines the orientation of 3 points (counterclockwise, clockwise,
    undefined).
  circle_3pts:
    Computes the center and radius of a circle given 3 points.
  convex_hull:
    Computes the convex hull of a list of points.
  dist3D:
    Computes the distance between two points, a point and a line segment,
    two line segments, or a point and a triangle in 3D.  There are also
    corresponding versions for infinite lines and infinite planes.
```



```
  dist_line:
    Computes the distance of a point to a line.
  greatcircle:
    Computes the distance between two points on a sphere along the surface.
    Also has routines to convert between Cartesian coordinates and
    spherical coordinates.
  heron:
    Computes the area of a triangle given the lengths of 3 sides.
  intersect_circle_circle:
    Computes the intersection of two circles.
  intersectTF:
    Given two line segments, return whether they intersect or not (but
    doesn't return the point of intersection)
  intersect_line:
    Given two 2-D line segments, return whether they intersect or not, and
    return the point of intersection if there is a unique one.
  intersect_iline:
    Given two 2-D lines (infinite), return whether they intersect or not,
    and return the point of intersection if there is a unique one.
  intersect_iline_circle:
    Given an infinite 2-D line and a circle, return whether they intersect
    and also the point(s) of intersection.
  pointpoly:
    Given a polygon and a point, determines whether the point is in the
    polygon.  The behaviour when the point is on the boundary is left to
    the user.
  polygon_inter:
    Given two convex polygons, compute their intersection as another
    polygon.

Graph:
  bellmanford:
    Computes the shortest distance from one vertex to all other
    vertices.  Also computes the paths.  It is slow (O(n^3)) but
    handles negative weights.  Can also be used to detect negative
    cycles.
  bfs_path:
    Computes the shortest distance from one vertex to all other
    vertices.  Also computes the paths.  The edges in the graph must
    have equal cost.
  bicomp:
    Finds the biconnected components and articulation points of a graph.
  dijkstra:
    Computes the shortest distance from one vertex to all other
    vertices.  Also computes the paths.
  dijkstra_sparse:
    Same as dijkstra but for sparse graphs.  Complexity O((n+m) log(n+m)).
  eulertour:
    Determines if there is an Eulerian tour in the graph.  If so,
    find one.
  floyd:
    Computes the shortest distance between any two vertices.
  floyd_path:
    Like floyd, but also stores the paths.
  hungarian:
    Maximum/minimum weight bipartite matching.  O(N^3).
  matching:
    Compute unweighted matching of bipartite graphs.  (Matthew)
  mst:
    Compute the minimum spanning tree.
  mincostmaxflowdense:
    Compute the minimum cost maximum flow in a network.  Good for
    dense graphs when maximum flow is small.  Complexity is O(n^2 * flow).
  mincostmaxflowsparse:
    Compute the minimum cost maximum flow in a network.  Good for
    sparse graphs when maximum flow is small.  Complexity is
    O(m log(m) * flow).
  networkflow:
    Compute the maximum flow in a network.  Uses Ford-Fulkerson
```

```
    with complexity O(fm) where f is the value of the maximum flow
    and m is the number of edges.  Good for sparse graphs where
    the maximum flow is small.
  networkflow2:
    Compute the maximum flow in a network.  Uses relabel-to-front
    with complexity O(n^3).  Good for dense (but small) graphs
    where the maximum flow is large.
  scc:
    Compute the strongly connected components (and possibly the
    compressed graph) of a directed graph.
  top_sort:
    Topological sort on directed acyclic graph (or detect if a
    cycle exists). O(n+m)

Data Structures:
  fenwicktree:
    A data structure that supports the maintainence of cumulative sums
    in an array dynamically.  Most operations can be done in O(log N)
    time where N is the number of elements.

  suffixarray:
    An O(n) algorithm to construct a suffix array (and longest
    common prefix information) from a string.

Miscellaneous:
  asc_subseq:
    Longest (strictly) ascending/decreasing subsequence.
  binsearch:
    Binary search that also returns the position to insert an
    element if it is not found.
  common_subseq:
    Find the longest common subsequence of the two sequences.
  date:
    A class for dealing with dates in the Gregorian calendar.
  dow:
    Computing the day of the week.
  josephus:
    Finding the last survivor and killing order of the Josephus problem.
  kmp:
    Linear time string searching routines.
  int_prog:
    Integer programming.
  simplex:
    Linear programming by simplex algorithm.
  str_rotation_period:
    Computes the lexicographically least rotation of a string, as well
    as its period.
  unionfind:
    Union-find implementation to compute equivalence classes.
  vecsum:
    Find the contiguous subvector that gives the largest sum.
  zero_one:
    Zero-one programming.
```

```
// 2SAT solver: returns T/F whether it is satisfiable -- O(n+m)
//   - use NOT() to negate a variable (works on negated ones too!)
//   - ALWAYS use VAR() to talk about the non-negated version of the var i
//   - use add_clause to add a clause
//   - one possible satisfying assignment is returned in val[], if
//     it exists
//   - To FORCE i to be true:  add_clause(G,VAR(i),VAR(i));
//   - To implement XOR -- say (i XOR j) :
//      add_clause(G,VAR(i),VAR(j)); add_clause(G,NOT(VAR(i)),NOT(VAR(j)));
//     NOTE: val[] is indexed by i for var i, not by VAR(i)!!!

#include <iostream>
#include <algorithm>
#include <stack>
#include <cassert>
#include <vector>
using namespace std;

const int MAX_VARS = 100;          // maximum number of variables
const int MAX_NODES = 2*MAX_VARS;

struct Graph{
  int numNodes;
  vector<int> adj[MAX_NODES];
  void clear(){
    numNodes = 0;
    for(int i=0;i<MAX_NODES;i++)
      adj[i].clear();
  }
  void add_edge(int u,int v){
    if(find(adj[u].begin(),adj[u].end(),v) == adj[u].end())
      adj[u].push_back(v);
  }
};

int po[MAX_NODES],comp[MAX_NODES];
int num_scc;

void DFS(int v, const Graph& G, int& C, stack<int>& P,stack<int>& S){
  po[v] = C++;

  S.push(v);  P.push(v);
  for(unsigned int i=0;i<G.adj[v].size();i++){
    int w = G.adj[v][i];
    if(po[w] == -1){
      DFS(w,G,C,P,S);
    } else if(comp[w] == -1){
      while(!P.empty() && (po[P.top()] > po[w]))
        P.pop();
    }
  }
  if(!P.empty() && P.top() == v){
    while(!S.empty()){
      int t = S.top();
      S.pop();
      comp[t] = num_scc;
      if(t == v)
        break;
    }
    P.pop();
    num_scc++;
  }
}

int SCC(const Graph& G){
  num_scc = 0;
  int C = 1;
  stack<int> P,S;
  fill(po,po+G.numNodes,-1);
```



```
  fill(comp,comp+G.numNodes,-1);
  for(int i=0;i<G.numNodes;i++)
    if(po[i] == -1)
      DFS(i,G,C,P,S);

  return num_scc;
}


int VAR(int i) { return 2*i; }
int NOT(int i) { return i ^ 1; }

void add_clause(Graph &G, int v, int w) { // adds (v || w)
  if (v == NOT(w)) return;
  G.add_edge(NOT(v), w);
  G.add_edge(NOT(w), v);
}

bool twoSAT(const Graph &G, bool val[]) {   // assumes graph is built
  SCC(G);
  for (int i = 0; i < G.numNodes; i += 2) {
    if (comp[i] == comp[i+1]) return false;
    val[i/2] = (comp[i] < comp[i+1]);
  }
  return true;
}


// Declare this as a global variable if MAX_NODES is large to
//   avoid Runtime Error.
Graph G;

int main(){
  int m,n;
  while(cin >> n >> m && (n || m)){
    G.clear();
    G.numNodes = 2*n;

    for (int i = 0; i < m; i++) {
      cout << "Enter two variables for clause (1 - " << n
           << "), negative means negated: ";
      int x, y;
      cin >> x >> y;

      int var1 = VAR(abs(x)-1), var2 = VAR(abs(y)-1);
      if (x < 0) var1 = NOT(var1);
      if (y < 0) var2 = NOT(var2);
      add_clause(G, var1, var2);
    }

    bool val[MAX_VARS];
    if (twoSAT(G, val)) {
      for (int i = 0; i < n; i++) {
        cout << val[i] << ' ';
      }
      cout << endl;
    } else {
      cout << "Impossible" << endl;
    }
  }
  return 0;
}
```

```
/*
 * Area of a polygon
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/cga2.html
 *
 * This routine returns the SIGNED area of a polygon represented as an
 * array of n points (n >= 1).  The result is positive if the orientation is
 * counterclockwise, and negative otherwise.
 *
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <cassert>

using namespace std;

struct Point {
  double x, y;
};

double area_polygon(Point polygon[], int n)
{
  double sum = 0.0;

  for (int i = 0; i < n-1; i++) {
    sum += polygon[i].x * polygon[i+1].y - polygon[i].y * polygon[i+1].x;
  }
  sum += polygon[n-1].x * polygon[0].y - polygon[n-1].y * polygon[0].x;
  return sum/2.0;
}

int main(void)
{
  Point *polygon;
  int n;

  while (cin >> n && n > 0) {
    polygon = new Point[n];
    assert(polygon);
    for (int i = 0; i < n; i++) {
      cin >> polygon[i].x >> polygon[i].y;
    }
    cout << "Area = " << fixed << setprecision(2)
         << area_polygon(polygon, n) << endl;
    delete[] polygon;
  }
  return 0;
}
```

```
/*
 * Longest Ascending Subsequence
 *
 * Author: Howard Cheng
 * Reference:
 *   Gries, D.  The Science of Programming
 *
 * Given an array of size n, asc_seq returns the length of the longest
 * ascending subsequence, as well as one of the subsequences in S.
 * sasc_seq returns the length of the longest strictly ascending
 * subsequence.  It runs in O(n log n) time.
 *
 * Also included are simplified versions when only the length is needed.
 *
 * Note: If we want to find do the same things with descending
 * subsequences, just reverse the array before calling the routines.
 *
 */


#include <iostream>
#include <algorithm>
#include <vector>
#include <cassert>

using namespace std;

int asc_seq(int A[], int n, int S[])
{
  vector<int> last(n+1), pos(n+1), pred(n);
  if (n == 0) {
    return 0;
  }

  int len = 1;
  last[1] = A[pos[1] = 0];

  for (int i = 1; i < n; i++) {
    int j = upper_bound(last.begin()+1, last.begin()+len+1, A[i]) -
      last.begin();
    pred[i] = (j-1 > 0) ? pos[j-1] : -1;
    last[j] = A[pos[j] = i];
    len = max(len, j);
  }

  int start = pos[len];
  for (int i = len-1; i >= 0; i--) {
    S[i] = A[start];
    assert(i == 0 || pred[start] < start);
    start = pred[start];
  }

  return len;
}

int asc_seq(int A[], int n)
{
  vector<int> last(n+1);
  if (n == 0) {
    return 0;
  }

  int len = 1;
  last[1] = A[0];

  for (int i = 1; i < n; i++) {
    int j = upper_bound(last.begin()+1, last.begin()+len+1, A[i]) -
      last.begin();
    last[j] = A[i];
```



```
    len = max(len, j);
  }

  return len;
}

int sasc_seq(int A[], int n, int S[])
{
  vector<int> last(n+1), pos(n+1), pred(n);
  if (n == 0) {
    return 0;
  }

  int len = 1;
  last[1] = A[pos[1] = 0];

  for (int i = 1; i < n; i++) {
    int j = lower_bound(last.begin()+1, last.begin()+len+1, A[i]) -
      last.begin();
    pred[i] = (j-1 > 0) ? pos[j-1] : -1;
    last[j] = A[pos[j] = i];
    len = max(len, j);
  }

  int start = pos[len];
  for (int i = len-1; i >= 0; i--) {
    S[i] = A[start];
    start = pred[start];
  }

  return len;
}

int sasc_seq(int A[], int n)
{
  vector<int> last(n+1);
  if (n == 0) {
    return 0;
  }

  int len = 1;
  last[1] = A[0];

  for (int i = 1; i < n; i++) {
    int j = lower_bound(last.begin()+1, last.begin()+len+1, A[i]) -
      last.begin();
    last[j] = A[i];
    len = max(len, j);
  }

  return len;
}

int main(void)
{
  int *A, *S, n, i, k;

  while (cin >> n && n > 0) {
    A = new int[n];
    S = new int[n];
    for (i = 0; i < n; i++) {
      cin >> A[i];
    }
    k = asc_seq(A, n, S);
    cout << "length = " << k << endl;
    for (i = 0; i < k; i++) {
      cout << S[i] << " ";
    }
    cout << endl;
```

```
    k = sasc_seq(A, n, S);
    cout << "length = " << k << endl;
    for (i = 0; i < k; i++) {
      cout << S[i] << " ";
    }
    cout << endl;
    delete[] A;
    delete[] S;
  }
  return 0;
}
```

```
/*
 * Bellman-Ford Shortest Path Algorithm
 *
 * Author: Howard Cheng
 *
 * Given a weight matrix representing a graph and a source vertex, this
 * algorithm computes the shortest distance, as well as path, to each
 * of the other vertices.  The paths are represented by an inverted list,
 * such that if v preceeds immediately before w in a path from the
 * source to vertex w, then the path P[w] is v.  The distances from
 * the source to v is given in D[v] (DISCONNECT if not connected).
 *
 * Call get_path to recover the path.
 *
 * Note: the Bellman-Ford algorithm has complexity O(n^3), but it works even
 *       when edges have negative weights.  As long as there are no negative
 *       cycles the computed results are correct.
 *
 *       We can make this O(n*m) if we use an adjacency list representation.
 *
 *       This works for directed graphs too.
 *
 *       You can use this to detect negative cycles too.  See code.
 *
 */

#include <iostream>
#include <climits>
#include <cassert>

using namespace std;

const int MAX_NODES = 20;
const int DISCONNECT = INT_MAX;

/* assume that D and P have been allocated */
void bellmanford(int graph[MAX_NODES][MAX_NODES], int n, int src,
                 int D[], int P[])
{
  int v, w, k;

  for (v = 0; v < n; v++) {
    D[v] = INT_MAX;
    P[v] = -1;
  }
  D[src] = 0;

  for (k = 0; k < n-1; k++) {
    for (v = 0; v < n; v++) {
      for (w = 0; w < n; w++) {
        if (graph[v][w] != DISCONNECT && D[v] != INT_MAX) {
          if (D[w] == INT_MAX || D[w] > D[v] + graph[v][w]) {
            D[w] = D[v] + graph[v][w];
            P[w] = v;
          } else if (D[w] == D[v] + graph[v][w]) {
            /* do some tie-breaking here */
          }
        }
      }
    }
  }

  /* the following loop is used only to detect negative cycles, not */
  /* needed if you don't care about this                            */
  for (v = 0; v < n; v++) {
    for (w = 0; w < n; w++) {
      if (graph[v][w] != DISCONNECT && D[v] != INT_MAX) {
        if (D[w] == INT_MAX || D[w] > D[v] + graph[v][w]) {
          /* if we get here then there is a negative cycle somewhere */
```



```
          /* on the path from src to                                 */
        }
      }
    }
  }

}

int get_path(int v, int P[], int path[])
{
  int A[MAX_NODES];
  int i, k;

  k = 0;
  A[k++] = v;
  while (P[v] != -1) {
    v = P[v];
    A[k++] = v;
  }
  for (i = k-1; i >= 0; i--) {
    path[k-1-i] = A[i];
  }
  return k;
}

int main(void)
{
  int m, w, num;
  int i, j;
  int graph[MAX_NODES][MAX_NODES];
  int P[MAX_NODES][MAX_NODES], D[MAX_NODES][MAX_NODES];
  int path[MAX_NODES];

  /* clear graph */
  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      graph[i][j] = DISCONNECT;
    }
  }

  /* read graph */
  cin >> i >> j >> w;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    graph[i][j] = graph[j][i] = w;
    cin >> i >> j >> w;
  }

  for (i = 0; i < MAX_NODES; i++) {
    bellmanford(graph, MAX_NODES, i, D[i], P[i]);
  }

  /* do queries */
  cin >> i >> j;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    cout << i << " " << j << ":" << D[i][j] << endl;
    for (m = j; m != -1; m = P[i][m]) {
      cout << " " << m;
    }
    cout << endl;
    num = get_path(j, P[i], path);
    for (m = 0; m < num; m++) {
      cout << " " << path[m];
    }
    cout << endl;
    cin >> i >> j;
  }
```

```
  return 0;
}
```

```
/*
 * Shortest Path with BFS
 *
 * Author: Howard Cheng
 *
 * Given a graph represented by an adjacency list, this algorithm uses
 * BFS to find the shortest path from a source vertex to each of the
 * other vertices.  The distances from the source to v is given in D[v], and
 * D[v] is set to -1 if the source vertex is not connected to w.  Also,
 * the shortest path tree is stored in the array P.
 *
 * Call get_path to recover the path.
 *
 * Note: All edges must have the same cost for this to work.
 *       This algorithm has complexity O(n+m).
 *
 */

#include <iostream>
#include <cassert>
#include <algorithm>
#include <queue>

using namespace std;

const int MAX_NODES = 100;

struct Node {
  int deg;           /* number of outgoing edges */
  int adj[MAX_NODES];

  /* the following is not necessary, but useful in many situations */
  int cost[MAX_NODES];
};

void BFS_shortest_path(Node graph[], int n, int src, int D[], int P[])
{
  char used[MAX_NODES];
  queue<int> q;
  int i, v, w;

  fill(used, used+MAX_NODES, 0);
  q.push(src);
  used[src] = 1;

  for (i = 0; i < MAX_NODES; i++) {
    D[i] = -1;
    P[i] = -1;
  }
  D[src] = 0;

  while (!q.empty()) {
    v = q.front();
    q.pop();
    for (i = 0; i < graph[v].deg; i++) {
      w = graph[v].adj[i];
      if (!used[w]) {
        D[w] = D[v] + 1;
        P[w] = v;
        q.push(w);
        used[w] = 1;
      } else if (D[v] + 1 == D[w]) {
        /* put tie-breaker here */
        /* eg. find largest path in lexicographic order, when the path */
        /*     is considered in REVERSE!                               */
        P[w] = max(P[w], v);
      }
    }
  }
```



```
}

void clear(Node graph[], int n)
{
  int i;
  for (i = 0; i < n; i++) {
    graph[i].deg = 0;
  }
}

void add_edge(Node graph[], int v, int w, int cost)
{
  int i;

  /* make sure that we have no duplicate edges */
  for (i = 0; i < graph[v].deg; i++) {
    if (graph[v].adj[i] == w) {
      assert(0);
    }
  }

  graph[v].cost[graph[v].deg] = cost;
  graph[v].adj[graph[v].deg] = w;
  graph[v].deg++;
}

int get_path(int v, int P[], int path[])
{
  int A[MAX_NODES];
  int i, k;

  k = 0;
  A[k++] = v;
  while (P[v] != -1) {
    v = P[v];
    A[k++] = v;
  }
  for (i = k-1; i >= 0; i--) {
    path[k-1-i] = A[i];
  }
  return k;
}

int main(void)
{
  int v, w, num;
  int i;
  Node graph[MAX_NODES];
  int P[MAX_NODES][MAX_NODES], D[MAX_NODES][MAX_NODES];
  int path[MAX_NODES];

  clear(graph, MAX_NODES);
  while (cin >> v >> w && v >= 0 && w >= 0) {
    add_edge(graph, v, w, 1);
  }

  for (i = 0; i < MAX_NODES; i++) {
    BFS_shortest_path(graph, MAX_NODES, i, D[i], P[i]);
  }

  while (cin >> v >> w && v >= 0 && w >= 0) {
    cout << v << " " << w << ":" << D[v][w] << endl;
    num = get_path(w, P[v], path);
    assert(D[v][w] == -1 || num == D[v][w]+1);
    for (i = 0; i < num; i++) {
      cout << " " << path[i];
    }
    cout << endl;
  }
```

```
  return 0;
}
```

```
/*
 * Biconnected Components
 *
 * Author: Howard Cheng
 * Date: Oct 15, 2004
 *
 * The routine bicomp() uses DFS to find the biconnected components in
 * a graph.  The graph is stored as an adjacency list.  Use clear_graph()
 * and add_edge() to build the graph.
 *
 * Note: This works only on connected graphs.  See comment below in code.
 *
 * The code simply prints the biconnected components and the articulation
 * points.  Replace the printing code to do whatever is appropriate.
 *
 * NOTE: some articulation points may be printed multiple times.
 *
 *
 */

#include <iostream>
#include <stack>
#include <algorithm>
#include <cassert>

using namespace std;

/* maximum number of nodes, maximum degree, and maximum number of edges */
const int MAX_N = 1000;
const int MAX_DEG = 4;

struct Node {
  int deg;
  int nbrs[MAX_DEG];
  int dfs, back;
};

int dfn;

void clear_graph(Node G[], int n)
{
  int i;
  for (i = 0; i < n; i++) {
    G[i].deg = 0;
  }
}

void add_edge(Node G[], int u, int v)
{
  G[u].nbrs[G[u].deg++] = v;
  G[v].nbrs[G[v].deg++] = u;
}

void do_dfs(Node G[], int v, int pred, stack<int> &v_stack,
            stack<int> &w_stack)
{
  int i, w, child = 0;

  G[v].dfs = G[v].back = ++dfn;
  for (i = 0; i < G[v].deg; i++) {
    w = G[v].nbrs[i];
    if (G[w].dfs < G[v].dfs && w != pred) {
      /* back edge or unexamined forward edge */
      v_stack.push(v);
      w_stack.push(w);
    }
    if (!G[w].dfs) {
      do_dfs(G, w, v, v_stack, w_stack);
      child++;
```



```
      /* back up from recursion */
      if (G[w].back >= G[v].dfs) {
        /* new bicomponent */
        cout << "edges in new biconnected component:" << endl;
        while (v_stack.top() != v || w_stack.top() != w) {
          cout << v_stack.top() << " " << w_stack.top() << endl;
          v_stack.pop();
          w_stack.pop();
        }
        cout << v_stack.top() << " " << w_stack.top() << endl;
        v_stack.pop();
        w_stack.pop();

        if (pred != -1) {
          cout << "articulation point: " << v << endl;
        }
      } else {
        G[v].back = min(G[v].back, G[w].back);
      }
    } else {
      /* w has been examined already */
      G[v].back = min(G[v].back, G[w].dfs);
    }
  }
  if (pred == -1 && child > 1) {
    cout << "articulation point: " << v << endl;
  }
}

void bicomp(Node G[], int n)
{
  int i;
  stack<int> v_stack, w_stack;

  dfn = 0;
  for (i = 0; i < n; i++) {
    G[i].dfs = 0;
  }
  do_dfs(G, 0, -1, v_stack, w_stack);

  // NOTE: if you wish to process all connected components, you can simply
  // run the following code instead of the line above:
  //
  // for (int i = 0; i < n; i++) {
  //   if (G[i].dfs == 0) {
  //     do_dfs(G, i, -1, v_stack, w_stack);
  //   }
  // }
}

int main(void)
{
  Node G[MAX_N];
  int n, m, i, u, v;

  cin >> n;
  clear_graph(G, n);
  cin >> m;
  for (i = 0; i < m; i++) {
    cin >> u >> v;
    add_edge(G, u-1, v-1);
  }
  bicomp(G, n);
  return 0;
}
```

```
/*
 * Big integer implementation
 *
 * Author: Howard Cheng
 *
 * Each digit in our representation represents LOG_BASE decimal digits
 *
 */

#include <vector>
#include <string>
#include <cstdio>
#include <cctype>
#include <iostream>
#include <algorithm>
#include <utility>
#include <cassert>

using namespace std;
using namespace std::rel_ops;

typedef long long Digit;
#define BASE 1000000000
#define LOG_BASE 9
#define FMT_STR "%lld"
#define FMT_STR0 "%09lld"

class BigInteger {
private:

  int sign;            // +1 = positive, 0 = zero, -1 = negative
  vector<Digit> mag;   // magnitude

  void normalize();

public:
  BigInteger(Digit n = 0);
  BigInteger(const string &s);    // no error checking

  long long toLongLong() const;   // convert to long long (assumes no overflow)
  string toString() const;        // convert to string

  void clear();  // set to zero

  // comparison
  bool operator<(const BigInteger &a) const;
  bool operator==(const BigInteger &a) const;
  bool isZero() const;

  // arithmetic
  BigInteger &operator+=(const BigInteger &a);
  BigInteger &operator-=(const BigInteger &a);
  BigInteger &operator*=(const BigInteger &a);
  BigInteger &operator*=(Digit a);
  BigInteger &operator<<=(Digit a);
  BigInteger &operator/=(const BigInteger &a);
  BigInteger &operator/=(Digit a);
  BigInteger &operator%=(const BigInteger &a);
  friend Digit operator%(const BigInteger &a, Digit b);

  // we have *this = b * q + r
  // r is such that 0 <= r < |b|
  void divide(const BigInteger &b, BigInteger &q, BigInteger &r) const;
  void divide(Digit b, BigInteger &q, Digit &r) const;

  // root = floor(sqrt(a)).  Returns 1 if a is a perfect square, 0 otherwise.
  // assume >= 0
  int sqrt(BigInteger &root) const;
};
```



```
BigInteger operator+(const BigInteger &a, const BigInteger &b);
BigInteger operator-(const BigInteger &a, const BigInteger &b);
BigInteger operator*(const BigInteger &a, const BigInteger &b);
BigInteger operator*(const BigInteger &a, Digit b);
BigInteger operator<<(const BigInteger &a, Digit b);
BigInteger operator/(const BigInteger &a, const BigInteger &b);
BigInteger operator/(const BigInteger &a, Digit b);
BigInteger operator%(const BigInteger &a, const BigInteger &b);
Digit operator%(const BigInteger &a, Digit b);

BigInteger power(BigInteger x, Digit y);
istream &operator>>(istream &is, BigInteger &a);
ostream &operator<<(ostream &os, const BigInteger &a);

void BigInteger::normalize()
{
  if (mag.size() == 0) {
    return;
  }
  vector<Digit>::iterator p = mag.end();
  do {
    if (*(--p) != 0) break;
  } while (p != mag.begin());
  if (p == mag.begin() && *p == 0) {
    clear();
    sign = 0;
  } else {
    mag.erase(++p, mag.end());
  }
}

BigInteger::BigInteger(Digit n)
{
  if (n == 0) {
    sign = 0;
    return;
  }
  if (n < 0) {
    sign = -1;
    n = -n;
  } else {
    sign = 1;
  }

  while (n > 0) {
    mag.push_back(n % BASE);
    n /= BASE;
  }
}

BigInteger::BigInteger(const string &s)
{
  int l = 0;
  bool zero = true;
  bool neg = false;

  clear();

  sign = 1;
  if (s[l] == '-') {
    neg = true;
    l++;
  }

  for (; l < s.length(); l++) {
    *this *= 10;
    *this += s[l] - '0';
    zero &= s[l] == '0';
```

```
  }

  if (zero) {
    clear();
  }
  if (neg) {
    sign = -1;
  }
}

long long BigInteger::toLongLong() const
{
  long long a = 0;
  for (int i = mag.size()-1; i >= 0; i--) {
    a *= BASE;
    a += mag[i];
  }
  return sign * a;
}

string BigInteger::toString() const
{
  char buffer[LOG_BASE+1];
  string s;

  if (isZero()) {
    return "0";
  } else {
    if (sign < 0) {
      s += "-";
    }
    for (int i = mag.size()-1; i >= 0; i--) {
      if (i == (int)(mag.size()-1)) {
        sprintf(buffer, FMT_STR, mag[i]);
      } else {
        sprintf(buffer, FMT_STR0, mag[i]);
      }
      s += buffer;
    }
    return s;
  }
}

void BigInteger::clear()
{
  sign = 0;
  mag.clear();
}

bool BigInteger::operator<(const BigInteger &a) const
{
  if (sign != a.sign) {
    return sign < a.sign;
  } else if (sign == 0) {
    return false;
  } else if (mag.size() < a.mag.size()) {
    return sign > 0;
  } else if (mag.size() > a.mag.size()) {
    return sign < 0;
  } else {
    for (int i = mag.size()-1; i >= 0; i--) {
      if (mag[i] < a.mag[i]) {
        return sign > 0;
      } else if (mag[i] > a.mag[i]) {
        return sign < 0;
      }
    }
    return false;
  }
```



```
}

bool BigInteger::operator==(const BigInteger &a) const
{
  return sign == a.sign && mag == a.mag;
}

bool BigInteger::isZero() const
{
  return sign == 0;
}

BigInteger &BigInteger::operator+=(const BigInteger &a)
{
  if (a.sign == 0) {
    return *this;
  } else if (sign == 0) {
    sign = a.sign;
    mag = a.mag;
    return *this;
  } else if (sign < 0 && a.sign > 0) {
    BigInteger b(a);
    sign = 1;
    b -= *this;
    return *this = b;
  } else if (sign > 0 && a.sign < 0) {
    BigInteger b(a);
    b.sign = 1;
    return (*this) -= b;
  } else {
    Digit carry = 0;
    unsigned int limit = max(mag.size(), a.mag.size());
    for (unsigned int i = 0; i < limit; i++) {
      Digit s1 = (i < mag.size()) ? mag[i] : 0;
      Digit s2 = (i < a.mag.size()) ? a.mag[i] : 0;
      Digit sum = s1 + s2 + carry;
      Digit result = (sum < BASE) ? sum : sum - BASE;
      carry = (sum >= BASE);
      if (i < mag.size()) {
        mag[i] = result;
      } else {
        mag.push_back(result);
      }
    }
    if (carry) {
      mag.push_back(carry);
    }
    return *this;
  }
}

BigInteger &BigInteger::operator-=(const BigInteger &a)
{
  if (a.sign == 0) {
    return *this;
  } else if (sign == 0) {
    sign = -a.sign;
    mag = a.mag;
    return *this;
  } else if (sign != a.sign) {
    BigInteger b(a);
    b.sign *= -1;
    return *this += b;
  } else if (sign < 0) {
    BigInteger b(a);
    b.sign *= -1;
    sign *= -1;
    b -= *this;
    return *this = b;
```

```
  } else {
    if (*this == a) {
      clear();
      return *this;
    } else if (*this < a) {
      BigInteger b(a);
      b -= *this;
      b.sign *= -1;
      return *this = b;
    } else {
      // we know that *this > a
      unsigned int limit = mag.size();
      Digit borrow = 0;
      for (unsigned int i = 0; i < limit; i++) {
        Digit s1 = mag[i];
        Digit s2 = (i < a.mag.size()) ? a.mag[i] : 0;
        Digit diff = s1 - s2 - borrow;
        mag[i] = (diff >= 0) ? diff : diff + BASE;
        borrow = (diff < 0);
      }
      normalize();
      return *this;
    }
  }
}

BigInteger &BigInteger::operator*=(const BigInteger &a)
{
  BigInteger temp(*this);
  BigInteger c;

  if (this == &a) {
    c = a;          // make a copy to prevent clobbering it
  }

  const BigInteger &b = (this == &a) ? c : a;

  clear();
  if (b.sign) {
    for (unsigned int i = 0; i < b.mag.size(); i++) {
      if (b.mag[i] != 0) {
        *this += (temp * b.mag[i]);
      }
      temp <<= 1;
    }
    sign *= b.sign;
  }
  return *this;
}

BigInteger &BigInteger::operator*=(Digit a)
{
  if (a <= -BASE || a >= BASE) {
    BigInteger b(a);
    return (*this *= b);
  }

  if (isZero()) {
    return *this;
  } else if (a == 0) {
    clear();
    return *this;
  } else if (a < 0) {
    sign *= -1;
    a = -a;
  }

  Digit carry = 0;
  for (unsigned int i = 0; i < mag.size(); i++) {
```



```
    Digit prod = a * mag[i];
    mag[i] = (carry + prod) % BASE;
    carry = (carry + prod) / BASE;
  }
  if (carry) {
    mag.push_back(carry);
  }
  return *this;
}

BigInteger &BigInteger::operator<<=(Digit a)
{
  assert(a >= 0);
  if (sign) {
    while (a-- > 0) {
      mag.insert(mag.begin(), 0);
    }
  }
  return *this;
}

BigInteger &BigInteger::operator/=(const BigInteger &a)
{
  BigInteger temp(*this), r;
  temp.divide(a, *this, r);
  return *this;
}

BigInteger &BigInteger::operator/=(Digit a)
{
  BigInteger temp(*this);
  Digit r;
  temp.divide(a, *this, r);
  return *this;
}

BigInteger &BigInteger::operator%=(const BigInteger &a)
{
  BigInteger temp(*this), q;
  temp.divide(a, q, *this);
  return *this;
}

void BigInteger::divide(const BigInteger &b, BigInteger &q,
                        BigInteger &r) const
{
  // reference Knuth v.2 Algorithm D

  assert(!b.isZero());

  if (b.mag.size() == 1) {
    Digit r2;
    divide(b.sign*b.mag[0], q, r2);
    r = r2;
    return;
  }

  r = *this;
  if (r.sign < 0) {
    r.sign = 1;
  }
  q.clear();

  int n = b.mag.size();
  int m = mag.size() - n;
  if (m >= 0) {
    BigInteger v(b);
    q.mag.resize(m+1);
    q.sign = 1;
```

```
    // D1: normalize
    Digit d = BASE / (v.mag[n-1] + 1);  // Book is wrong.  See errata on web
    r *= d;
    v *= d;
    while ((int)r.mag.size() < m+n+1) {
      r.mag.push_back(0);
    }


    // loop
    for (int j = m; j >= 0; j--) {
      // D3: calculate q2
      Digit t = r.mag[j+n] * BASE + r.mag[j+n-1];
      Digit q2 = t / v.mag[n-1];
      Digit r2 = t - q2 * v.mag[n-1];
      if (q2 == BASE || q2 * v.mag[n-2] > BASE * r2 + r.mag[j+n-2]) {
        q2--;
        r2 += v.mag[n-1];
        if (r2 < BASE &&
            (q2 == BASE || q2 * v.mag[n-2] > BASE * r2 + r.mag[j+n-2])) {
          q2--;
          r2 += v.mag[n-1];
        }
      }

      // D4: multiply and subtract
      Digit carry, borrow, diff;
      carry = borrow = 0;
      for (int i = 0; i <= n; i++) {
        t = q2 * ((i < n) ? v.mag[i] : 0) + carry;
        carry = t / BASE;
        t %= BASE;
        diff = r.mag[j+i] - t - borrow;
        r.mag[j+i] = (diff >= 0 || i == n) ? diff : diff + BASE;
        borrow = (diff < 0);
      }

      // D5: test remainder
      q.mag[j] = q2;
      if (r.mag[n+j] < 0) {
        // D6: add back
        q.mag[j]--;
        carry = 0;
        for (int i = 0; i < n; i++) {
          t = r.mag[j+i] + v.mag[i] + carry;
          r.mag[j+i] = (t < BASE) ? t : t - BASE;
          carry = (t >= BASE);
        }
        r.mag[j+n] += carry;
      }
    }

    q.normalize();
    r.normalize();

    // D8: unnormalize
    r /= d;
  }

  // normalize
  if (sign < 0 && b.sign > 0) {
    q.sign *= -1;
    r *= -1;
    if (!r.isZero()) {
      r += b;
      q -= 1;
    }
  } else if (sign > 0 && b.sign < 0) {
```



```
    q.sign *= -1;
  } else if (sign < 0 && b.sign < 0 && !r.isZero()) {
    r += b;
    r *= -1;
    q += 1;
  }
}

void BigInteger::divide(Digit b, BigInteger &q, Digit &r) const
{
  if (b <= -BASE || b >= BASE) {
    BigInteger bb(b), rr;
    divide(bb, q, rr);
    r = rr.toLongLong();
    return;
  }

  int bsign = 1;
  if (b < 0) {
    b *= -1;
    bsign = -1;
  }
  q.clear();

  r = 0;
  for (int i = mag.size()-1; i >= 0; i--) {
    Digit t = r * BASE + mag[i];
    if (t / b > 0) {
      q.sign = 1;
    }
    q.mag.insert(q.mag.begin(), t / b);
    r = t - q.mag[0] * b;
  }

  // normalize
  q.normalize();

  if (sign < 0 && bsign > 0) {
    q.sign *= -1;
    r *= -1;
    if (r) {
      r += b;
      q -= 1;
    }
  } else if (sign > 0 && bsign < 0) {
    q.sign *= -1;
  } else if (sign < 0 && bsign < 0 && r) {
    r = b - r;
    q += 1;
  }
}

int BigInteger::sqrt(BigInteger &root) const
{
  assert(sign >= 0);
  root.clear();
  if (sign == 0) {
    return 1;
  }

  // figure out how many digits there are
  BigInteger x, r, t2;
  r.sign = 1;
  int d = mag.size();

  int root_d = (d % 2) ? (d+1)/2 : d / 2;

  if (d % 2) {
    r.mag.resize(1);
```

```
    r.mag[0] = mag[--d];
  } else {
    r.mag.resize(2);
    r.mag[1] = mag[--d];
    r.mag[0] = mag[--d];
  }

  root.sign = 1;

  // figure out one digit at a time
  for (int k = root_d - 1; k >= 0; k--) {
    // invariant: result is the sqrt (integer part) of the digits processed
    // so far

    // look for next digit in result by binary search
    x = root * 2;
    x <<= 1;
    Digit t;

    Digit lo = 0, hi = BASE;
    while (hi - lo > 1) {
      Digit mid = (lo + hi) / 2;
      x.mag[0] = t = mid;
      t2 = x * t;
      if (t2 < r || t2 == r) {
        lo = mid;
      } else {
        hi = mid;
      }
    }
    root <<= 1;
    root.mag[0] = lo;

    // form the next r
    x.mag[0] = t = lo;
    t2 = x * t;
    r -= t2;
    r <<= 1;
    r += (d > 0) ? mag[--d] : 0;
    r <<= 1;
    r += (d > 0) ? mag[--d] : 0;
  }

  return r.isZero();
}

BigInteger operator+(const BigInteger &a, const BigInteger &b)
{
  BigInteger r(a);
  r += b;
  return r;
}

BigInteger operator-(const BigInteger &a, const BigInteger &b)
{
  BigInteger r(a);
  r -= b;
  return r;
}

BigInteger operator*(const BigInteger &a, const BigInteger &b)
{
  BigInteger r(a);
  r *= b;
  return r;
}

BigInteger operator*(const BigInteger &a, Digit b)
{
```



```
  BigInteger r(a);
  r *= b;
  return r;
}

BigInteger operator<<(const BigInteger &a, Digit b)
{
  BigInteger r(a);
  r <<= b;
  return r;
}

BigInteger operator/(const BigInteger &a, const BigInteger &b)
{
  BigInteger r(a);
  r /= b;
  return r;
}

BigInteger operator/(const BigInteger &a, Digit b)
{
  BigInteger r(a);
  r /= b;
  return r;
}

BigInteger operator%(const BigInteger &a, const BigInteger &b)
{
  BigInteger r(a);
  r %= b;
  return r;
}

Digit operator%(const BigInteger &a, Digit b)
{
  Digit r;
  if (b > 0 && b < BASE) {
    r = 0;
    for (int i = a.mag.size()-1; i >= 0; i--) {
      r = ((r * BASE) + a.mag[i]) % b;
    }
    if (a.sign < 0) {
      r = (b - r) % b;
    }
    return r;
  }

  BigInteger q;

  a.divide(b, q, r);
  return r;
}

BigInteger power(BigInteger x, Digit y)
{
  BigInteger result(1), sx(x);

  assert(y >= 0);
  while (y > 0) {
    if (y & 0x01) {
      y--;
      result *= sx;
    } else {
      sx *= sx;
      y >>= 1;
    }
  }
  return result;
}
```

```
istream &operator>>(istream &is, BigInteger &a)
{
  string s;
  char c = ' ';

  is.get(c);

  while (!is.eof() && isspace(c)) {
    is.get(c);
  }
  if (is.eof()) {
    if (isdigit(c)) {
      a = (int)(c - '0');
      is.clear();
    }
    return is;
  }

  if (c == '-') {
    s = "-";
  } else {
    is.unget();
    if (!isdigit(c)) {
      return is;
    }
  }

  is.get(c);
  while (!is.eof() && isdigit(c)) {
    s += c;
    is.get(c);
  }
  if (!is.eof()) {
    is.unget();
  }
  a = s;
  is.clear();
  return is;
}

ostream &operator<<(ostream &os, const BigInteger &a)
{
  return (os << a.toString());
}


int main()
{
  BigInteger a, b;

  while (cin >> a >> b && (!(a == 0) || !(b == 0))) {
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    if (!(a < 0)) {
      if (a.sqrt(b)) {
        cout << "perfect square" << endl;
      }
      cout << "sqrt(a) = " << b << endl;
    }
  }
  return 0;
}
```

```java
// Java template for using BigInteger class.
//
// Note that there is also a similar BigDecimal class which may be useful.
//
// Name of the file must be NAME.java where NAME is the class name.
//
// To compile:
//
// javac NAME.java
//
// To run:
//
// java NAME
//
//
// Note: in Java, all non-native types (including arrays) need to be
// allocated by new.  Multidimensional arrays can be allocated in one
// call.  See below.
//

// for importing IO routines
import java.io.*;
import java.util.Scanner;
import java.math.BigInteger;

class bignumber {

  // this is main
  public static void main(String argv[])
  {
    // A scanner can be used to read many different types
    Scanner sc = new Scanner(System.in);

    // checking whether there is a next token can be done before
    // actually reading it and fail (vs. I/O model in C++)
    while (sc.hasNextInt()) {
      int N = sc.nextInt();
      int K = sc.nextInt();

      // here is how to allocate two dimensional arrays
      BigInteger binom[][] = new BigInteger[N+1][N+1];
      for (int n = 0; n <= N; n++) {
        // here is how you construct from an integer.
        binom[n][0] = binom[n][n] = BigInteger.valueOf(1);
        for (int k = 1; k < n; k++) {
          binom[n][k] = binom[n-1][k-1].add(binom[n-1][k]);
        }
      }


      // to print something, use System.out.println().
      // Arguments are strings (in double quotes), and most data types
      // can be converted to strings and concatenated.
      //
      // Call it with no argument to produce a blank line, or use print()
      // to print without a trailing end-of-line
      System.out.println("C(" + N + "," + K + ")=" + binom[N][K]);
    }

    /*

    Here are a bunch of other things you can do with BigIntegers.
    Assuming a, b, c, d are BigIntegers, n is an int

    a = BigInteger.ZERO;               a = 0
    a = BigInteger.ONE;                a = 1
    a = new BigInteger("FF", 16);      a = 255
    a = sc.nextBigInteger();           cin >> a
    s = a.toString();                  convert to string representation
```



```java
    s = a.toString(base);                  convert to string representation in
                                           given base

    x = a.intValue();                      convert to smaller types, but may
    x = a.longValue();                     lose precision
    x = a.floatValue();
    x = a.doubleValue();

    a = b.abs();                           a = |b|
    n = a.signum();                        n = 0, +1, -1 depending on sign of a

    a = b.negate();                        a = -b;
    a = b.add(c);                          a = b+c
    a = b.subtract(c);                     a = b-c
    a = b.multiply(c);                     a = b*c
    a = b.divide(c);                       a = b/c
    a = b.remainder(c);                    a = b%c
    a = b.mod(c);                          a = b%c, but c must be positive
                                                    and a >= 0

    if (a.compareTo(b))                    -1 if a < b
                                           0 if a == b
                                           1 if a > b
    if (a.equals(b))                       true iff a == b

    a = b.min(c);                          a = min(b, c)
    a = b.max(c);                          a = max(b, c)

    a = b.pow(n);                          a = pow(b, n)
    a = b.modpow(n, c);                    a = pow(b, n) mod c, c > 0

    a = b.gcd(c);                          a = gcd(|b|, |c|)
    a = b.modInverse(c);                   a = b^(-1) mod c

    n = a.bitLength();                     number of bits in 2's complement
                                           representation, minus the sign bit

    if (a.isProbablePrime(n))              whether a is prime, with error
                                           of (1/2)^n

    */


    System.exit(0);
  }
}
```

```
// Binomial Coefficients
//
// Two ways to compute binomial coefficients:
//
//   - one way computes all binomial coefficients with n <= MAX_N O(MAX_N^2)
//   - one way computes a single binomial coefficient O(k)
//
// Author: Howard Cheng and Cody Barnson
//

#include <iostream>

using namespace std;

typedef long long ll;

// computes all binomial coefficients up to MAX_N.  Read them off the table
// after calling precomp().  O(MAX_N^2)
const int MAX_N = 10;
ll binom[MAX_N+1][MAX_N+1];
void precomp()
{
  for (int n = 0; n <= MAX_N; n++) {
    binom[n][0] = binom[n][n] = 1;
    for (int k = 1; k < n; k++) {
      binom[n][k] = binom[n-1][k] + binom[n-1][k-1];
    }
  }
}

// computes single binomial coefficient C(n, k)   O(k)
ll binom(int n, int k)
{
  if (k == 0 || k == n) return 1;
  k = min(k, n - k);
  ll ans = 1;
  for (ll i = 1; i <= k; i++) {
    ans *= (n - k + i);
    ans /= i;
  }
  return ans;
}
```

```
/*
 * Binary Search
 *
 * Author: Howard Cheng
 *
 * Note: you may wish to use the STL functions lower_bound and upper_bound
 * instead.
 *
 * Given a sorted array A of size n, it tries to find an item x in the
 * the array using binary search.  The function returns non-zero if
 * x is found, and zero otherwise.  Furthermore, if it is found, then
 * A[index] = x.  If it is not found, then index is the place x should
 * be inserted into A.
 *
 * ie.  A[i] <= x           for 0 <= i < index
 *             x < A[i]     for index <= i < n
 *
 * This routine is written for integer arrays, but can be adapted to
 * other types by changing the comparison operator.
 *
 * There is also an insert routine here that will insert the element into
 * the right place after the array has been reallocated (if necessary) to
 * store n+1 elements.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

bool bin_search(const int A[], int n, int x, int &index)
{
   int l, u, m;

   if (n <= 0 || x < A[0]) {  // check the first element, but only if it exists
      index = 0;
      return false;
   }
   if (A[n-1] < x) {
      index = n;
      return false;
   }
   if (x == A[n-1]) {
      index = n-1;
      return true;
   }
   l = 0;
   u = n-1;
   while (l+1 < u) {
      assert(A[l] <= x && x < A[u]);
      m = (l+u)/2;
      if (A[m] <= x) {
         l = m;
      } else {
         u = m;
      }
   }
   if (A[l] == x) {
      index = l;
      return true;
   } else {
      index = u;
      return false;
   }
}

void insert(int A[], int n, int x, int index)
{
```



```
   int i;
   for (i = n-1; i >= index+1; i--) {
      A[i] = A[i-1];
   }
   A[index] = x;
}


int main(void)
{
   int A[10000];
   int n, i, x, index;

   // implements binary insertion sort, but only keeps the unique elements
   n = 0;
   while (cin >> x && n < 10000) {
      if (!bin_search(A, n, x, index)) {
         n++;
         insert(A, n, x, index);
      }
      cout << "List:";
      for (i = 0; i < n; i++) {
        cout << " " << A[i];
        if (i == index) {
          cout << "*";        // show which one is just inserted
        }
      }
      cout << endl;
   }
   return 0;
}
```

```
/*
 * Orientation analysis
 *
 * Author: Howard Cheng
 * Reference:
 *   http://wilma.cs.brown.edu/courses/cs016/packet/node18.html
 *
 * Given three points a, b, c, it returns whether the path from a to b to c
 * is counterclockwise, clockwise, or undefined.
 *
 * Undefined is returned if the 3 points are colinear, and c is between
 * a and b.
 */

#include <iostream>
#include <cmath>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

/* counterclockwise, clockwise, or undefined */
enum Orientation {CCW, CW, CNEITHER};

Orientation ccw(Point a, Point b, Point c)
{
  double dx1 = b.x - a.x;
  double dx2 = c.x - b.x;
  double dy1 = b.y - a.y;
  double dy2 = c.y - b.y;
  double t1 = dy2 * dx1;
  double t2 = dy1 * dx2;

  if (fabs(t1 - t2) < EPSILON) {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) {
      if (dx1*dx1 + dy1*dy1 >= dx2*dx2 + dy2*dy2 - EPSILON) {
        return CNEITHER;
      } else {
        return CW;
      }
    } else {
      return CCW;
    }
  } else if (t1 > t2) {
    return CCW;
  } else {
    return CW;
  }
}

int main(void)
{
  Point a, b, c;
  Orientation res;

  while (cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y) {
    res = ccw(a,b,c);
    if (res == CW) {
      cout << "CW" << endl;
    } else if (res == CCW) {
      cout << "CCW" << endl;
    } else if (res == CNEITHER) {
      cout << "CNEITHER" << endl;
    } else {
```



```
      printf("Help, I am in trouble!\n");
      exit(1);
    }
  }
  return 0;
}
```

```
/*
 * Parameters of circle from 3 points
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/
 *
 * This routine computes the parameters of a circle (center and radius)
 * from 3 points.  Returns non-zero if successful, zero if the three
 * points are colinear.
 *
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

int circle(Point p1, Point p2, Point p3, Point &center, double &r)
{
  double a,b,c,d,e,f,g;

  a = p2.x - p1.x;
  b = p2.y - p1.y;
  c = p3.x - p1.x;
  d = p3.y - p1.y;
  e = a*(p1.x + p2.x) + b*(p1.y + p2.y);
  f = c*(p1.x + p3.x) + d*(p1.y + p3.y);
  g = 2.0*(a*(p3.y - p2.y) - b*(p3.x - p2.x));
  if (fabs(g) < EPSILON) {
    return 0;
  }
  center.x = (d*e - b*f) / g;
  center.y = (a*f - c*e) / g;
  r = sqrt((p1.x-center.x)*(p1.x-center.x) + (p1.y-center.y)*(p1.y-center.y));
  return 1;
}

int main(void)
{
  Point a, b, c, center;
  double r;

  while (cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y) {
    if (circle(a, b, c, center, r)) {
      cout << fixed << setprecision(3);
      cout << "center=(" << center.x << "," << center.y << ")" << endl;
      cout << "radius=" << r << endl;
    } else {
      cout << "colinear" << endl;
    }
  }
  return 0;
}
```

```
/*
 * Longest common subsequence
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.ics.uci.edu/~eppstein/161/960229.html
 *
 * Given two arrays A and B with sizes n and m respectively, compute the
 * length of the longest common subsequence.  It also returns in s a longest
 * common subsequence (it may not be unique).  One can specify which one
 * to choose when multiple longest common subsequences exist.
 *
 * Running time and space requirement is O(mn).
 *
 */

#include <iostream>
#include <algorithm>
#include <cassert>

using namespace std;

const int MAX_LEN = 20;

int LCS(int A[], int n, int B[], int m, int s[])
{
  int L[MAX_LEN+1][MAX_LEN+1];
  int i, j, k;

  for (i = n; i >= 0; i--) {
    for (j = m; j >= 0; j--) {
      if (i == n || j == m) {
        L[i][j] = 0;
      } else if (A[i] == B[j]) {
        L[i][j] = 1 + L[i+1][j+1];
      } else {
        L[i][j] = max(L[i+1][j], L[i][j+1]);
      }
    }
  }

  /* the following is not needed if you are not interested in the sequence */
  k = 0;
  i = j = 0;
  while (i < n && j < m) {
    if (A[i] == B[j]) {
      s[k++] = A[i];
      i++;
      j++;
    } else if (L[i+1][j] > L[i][j+1]) {
      i++;
    } else if (L[i+1][j] < L[i][j+1]) {
      j++;
    } else {
      /* put tie-breaking conditions here */

      /* eg. pick the one that starts at the first one the earliest */
      j++;
    }
  }
  return L[0][0];
}

int main(void)
{
  int A[MAX_LEN], B[MAX_LEN], s[MAX_LEN];
  int m, n, i, l;

  while (cin >> n >> m && 1 <= n && 1 <= m &&
```



```
         n <= MAX_LEN && m <= MAX_LEN) {
    for (i = 0; i < n; i++) {
      cin >> A[i];
    }
    for (i = 0; i < m; i++) {
      cin >> B[i];
    }
    l = LCS(A, n, B, m, s);
    for (i = 0; i < l; i++) {
      cout << s[i] << " ";
    }
    cout << endl << "Len = " << l << endl;
  }
  return 0;
}
```

```
/*
 * Convex hull
 *
 * Author: Howard Cheng
 * Reference:
 *   http://wilma.cs.brown.edu/courses/cs016/packet/node25.html
 *
 * Given a list of n (n >= 1) points in an array, it returns the vertices of
 * the convex hull in counterclockwise order.  Also returns the number of
 * vertices in the convex hull.  Assumes that the hull array has been
 * allocated to store the right number of elements (n elements is safe).
 * The points in the original polygon will be re-ordered.
 *
 * Note: The hull contains a maximum number of points.  ie. all colinear
 *       points and non-distinct points are included in the hull.
 *
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;

  bool operator<(const Point &p) const {
    return y < p.y || (y == p.y && x < p.x);
  }

};

/* counterclockwise, clockwise, or undefined */
enum Orientation {CCW, CW, CNEITHER};

/* Global point for computing convex hull */
Point start_p, max_p;

bool colinear(Point a, Point b, Point c)
{
  double dx1 = b.x - a.x;
  double dx2 = c.x - b.x;
  double dy1 = b.y - a.y;
  double dy2 = c.y - b.y;
  double t1 = dy2 * dx1;
  double t2 = dy1 * dx2;
  return fabs(t1 - t2) < EPSILON;
}

Orientation ccw(Point a, Point b, Point c)
{
  double dx1 = b.x - a.x;
  double dx2 = c.x - b.x;
  double dy1 = b.y - a.y;
  double dy2 = c.y - b.y;
  double t1 = dy2 * dx1;
  double t2 = dy1 * dx2;

  if (fabs(t1 - t2) < EPSILON) {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) {
      if (dx1*dx1 + dy1*dy1 >= dx2*dx2 + dy2*dy2 - EPSILON) {
        return CNEITHER;
      } else {
```



```
        return CW;
      }
    } else {
      return CCW;
    }
  } else if (t1 > t2) {
    return CCW;
  } else {
    return CW;
  }
}

bool ccw_cmp(const Point &a, const Point &b)
{
  return ccw(start_p, a, b) == CCW;
}

bool sort_cmp(const Point &a, const Point &b)
{
  if (colinear(start_p, a, max_p) && colinear(start_p, b, max_p)) {
    double dx1 = abs(start_p.x - a.x);
    double dx2 = abs(start_p.x - b.x);
    double dy1 = abs(start_p.y - a.y);
    double dy2 = abs(start_p.y - b.y);
    return dx1 > dx2 || (dx1 == dx2 && dy1 > dy2);
  } else {
    return ccw(start_p, a, b) == CCW;
  }
}

int convex_hull(Point polygon[], int n, Point hull[]) {
  int count, best_i, i;

  sort(polygon, polygon+n);
  for (int i = n-1; i >= 1; i--) {
    if (fabs(polygon[i].x - polygon[i-1].x) < EPSILON &&
        fabs(polygon[i].y - polygon[i-1].y) < EPSILON) {
      for (int j = i; j < n-1; j++) {
        polygon[j] = polygon[j+1];
      }
      n--;
    }
  }

  assert(n > 0);

  if (n == 1) {
    hull[0] = polygon[0];
    return 1;
  }

  /* find the first point: min y, and then min x */
  best_i = min_element(polygon, polygon+n) - polygon;
  swap(polygon[0], polygon[best_i]);
  start_p = polygon[0];

  /* find the maximum angle wrt start_p and positive x-axis */
  best_i = 1;
  for (i = 2; i < n; i++) {
    if (ccw_cmp(polygon[best_i], polygon[i])) {
      best_i = i;
    }
  }
  max_p = polygon[best_i];

  /* get simple closed polygon */
  sort(polygon+1, polygon+n, sort_cmp);

  /* do convex hull */
```

```
  count = 0;
  hull[count] = polygon[count]; count++;
  hull[count] = polygon[count]; count++;
  for (i = 2; i < n; i++) {
    while (count > 1 &&
           ccw(hull[count-2], hull[count-1], polygon[i]) == CW) {
      /* pop point */
      count--;
    }
    hull[count++] = polygon[i];
  }
  return count;
}

int main(void)
{
  Point *polygon, *hull;
  int n, hull_size;
  int i;

  while (cin >> n && n > 0) {
    polygon = new Point[n];
    hull = new Point[n];
    assert(polygon && hull);
    for (i = 0; i < n; i++) {
      cin >> polygon[i].x >> polygon[i].y;
    }
    hull_size = convex_hull(polygon, n, hull);
    cout << "Sorted:" << endl;
    for (i = 0; i < n; i++) {
      cout << fixed << setprecision(2);
      cout << "(" << polygon[i].x << "," << polygon[i].y << ")" << endl;
    }
    cout << endl;
    cout << "Hull size = " << hull_size << endl;
    for (i = 0; i < hull_size; i++) {
      cout << "(" << hull[i].x << "," << hull[i].y << ")" << endl;
    }
    cout << endl;
    delete[] polygon;
    delete[] hull;
  }
  return 0;
}
```

```
/*
 * Chinese Remainder Theorem
 *
 * Author: Howard Cheng
 * Reference:
 *   Geddes, K.O., Czapor, S.R., and Labahn, G.  Algorithms for Computer
 *   Algebra, Kluwer Academic Publishers, 1992, p. 180
 *
 * Given n relatively prime modular in m[0], ..., m[n-1], and right-hand
 * sides a[0], ..., a[n-1], the routine solves for the unique solution
 * in the range 0 <= x < m[0]*m[1]*...*m[n-1] such that x = a[i] mod m[i]
 * for all 0 <= i < n.  The algorithm used is Garner's algorithm, which
 * is not the same as the one usually used in number theory textbooks.
 *
 * It is assumed that m[i] are positive and pairwise relatively prime.
 * a[i] can be any integer.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int gcd(int a, int b, int &s, int &t)
{
  int r, r1, r2, a1, a2, b1, b2, q;
  int A = a;
  int B = b;

  a1 = b2 = 1;
  a2 = b1 = 0;

  while (b) {
    assert(a1*A + a2*B == a);
    q = a / b;
    r = a % b;
    r1 = a1 - q*b1;
    r2 = a2 - q*b2;
    a = b;
    a1 = b1;
    a2 = b2;
    b = r;
    b1 = r1;
    b2 = r2;
  }

  s = a1;
  t = a2;
  assert(a >= 0);
  return a;
}

int cra(int n, int m[], int a[])
{
  int x, i, k, prod, temp;
  int *gamma, *v;

  gamma = new int[n];
  v = new int[n];
  assert(gamma && v);

  /* compute inverses */
  for (k = 1; k < n; k++) {
    prod = m[0] % m[k];
    for (i = 1; i < k; i++) {
      prod = (prod * m[i]) % m[k];
    }
    gcd(prod, m[k], gamma[k], temp);
```



```
    gamma[k] %= m[k];
    if (gamma[k] < 0) {
      gamma[k] += m[k];
    }
  }

  /* compute coefficients */
  v[0] = a[0];
  for (k = 1; k < n; k++) {
    temp = v[k-1];
    for (i = k-2; i >= 0; i--) {
      temp = (temp * m[i] + v[i]) % m[k];
      if (temp < 0) {
        temp += m[k];
      }
    }
    v[k] = ((a[k] - temp) * gamma[k]) % m[k];
    if (v[k] < 0) {
      v[k] += m[k];
    }
  }

  /* convert from mixed-radix representation */
  x = v[n-1];
  for (k = n-2; k >= 0; k--) {
    x = x * m[k] + v[k];
  }

  delete[] gamma;
  delete[] v;

  return x;
}


int main(void)
{
  int n, *m, *a, i, x;

  while (cin >> n && n > 0) {
    m = new int[n];
    a = new int[n];
    assert(m && a);
    cout << "Enter moduli:" << endl;
    for (i = 0; i < n; i++) {
      cin >> m[i];
    }
    cout << "Enter right-hand side:" << endl;
    for (i = 0; i < n; i++) {
      cin >> a[i];
    }
    x = cra(n, m, a);
    cout << "x = " << x << endl;

    for (i = 0; i < n; i++) {
      assert((x-a[i]) % m[i] == 0);
    }

    delete[] m;
    delete[] a;
  }
  return 0;
}
```

```
//
// Date class
//
// This is an implementation of some common functionalities for dates.
// It can represent dates from Jan 1, 1753 to after (dates before that
// time are complicated...).
//

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>
#include <cctype>

using namespace std;
using namespace std::rel_ops;

struct Date {

  int yyyy;
  int mm;
  int dd;

  // no dates before 1753
  static int const BASE_YEAR = 1753;

  // Enumerated type for names of the days of the week
  enum dayName {SUN,MON,TUE,WED,THU,FRI,SAT};

  // Is a date valid
  static bool validDate(int yr, int mon, int day)
  {
    return yr >= BASE_YEAR && mon >= 1 && mon <= 12 &&
      day > 0 && day <= daysIn(mon, yr);
  }

  bool isValid() const
  {
    return validDate(yyyy, mm, dd);
  }

  // Constructor to create a specific date.  If the date is invalid,
  // the behaviour is undefined
  Date(int yr = 1970, int mon = 1, int day = 1)
  {
    yyyy = yr;
    mm = mon;
    dd = day;
  }

  // Returns the day of the week for this
  dayName dayOfWeek() const
  {
    int a = (14 - mm) / 12;
    int y = yyyy - a;
    int m = mm + 12 * a - 2;
    return (dayName)((dd + y + y/4 - y/100 + y/400 + 31 * m / 12) % 7);
  }

  // comparison operators
  bool operator==(const Date &d) const
  {
    return dd == d.dd && mm == d.mm && yyyy == d.yyyy;
  }

  bool operator<(const Date &d) const
  {
    return yyyy < d.yyyy || (yyyy == d.yyyy && mm < d.mm) ||
      (yyyy == d.yyyy && mm == d.mm && dd < d.dd);
```



```
  }

  // Returns true if yr is a leap year
  static bool leapYear(int y)
  {
    return (y % 400 ==0 || (y % 4 == 0 && y % 100 != 0));
  }

  // number of days in this month
  static int daysIn(int m, int y)
  {
    switch (m) {
    case 4  :
    case 6  :
    case 9  :
    case 11 :
      return 30;
    case 2  :
      if (leapYear(y)) {
        return 29;
      }
      else {
        return 28;
      }
    default :
      return 31;
    }
  }

  // increment by day, month, or year
  //
  // Use negative argument to decrement
  //
  // If adding a month/year results in a date before BASE_YEAR, the result
  // is undefined.
  //
  // If adding a month/year results in an invalid date (Feb 29 on a non-leap
  // year, Feb 31, Jun 31, etc.), the results are automatically "rounded down"
  // to the last valid date

  // add n days to the date: complexity is about n/30 iterations
  void addDay(int n = 1)
  {
    dd += n;
    while (dd > daysIn(mm,yyyy)) {
      dd -= daysIn(mm,yyyy);
      if (++mm > 12) {
        mm = 1;
        yyyy++;
      }
    }

    while (dd < 1) {
      if (--mm < 1) {
        mm = 12;
        yyyy--;
      }
      dd += daysIn(mm,yyyy);
    }
  }

  // add n months to the date: complexity is about n/12 iterations
  void addMonth(int n = 1)
  {
    mm += n;
    while (mm > 12) {
      mm -= 12;
      yyyy++;
    }
```

```
    while (mm < 1)  {
      mm += 12;
      yyyy--;
    }

    if (dd > daysIn(mm,yyyy)) {
      dd = daysIn(mm,yyyy);
    }
  }

  // add n years to the date
  void addYear(int n = 1)
  {
    yyyy += n;
    if (!leapYear(yyyy) && mm == 2 && dd == 29) {
      dd = 28;
    }
  }

  // number of days since 1753/01/01, including the current date
  int daysFromStart() const
  {
    int c = 0;
    Date d(BASE_YEAR, 1, 1);
    Date d2(d);

    d2.addYear(1);
    while (d2 < *this) {
      c += leapYear(d.yyyy) ? 366 : 365;
      d = d2;
      d2.addYear(1);
    }

    d2 = d;
    d2.addMonth(1);
    while (d2 < *this) {
      c += daysIn(d.mm, d.yyyy);
      d = d2;
      d2.addMonth(1);
    }
    while (d <= *this) {
      d.addDay();
      c++;
    }
    return c;
  }
};

// Reads a date in yyyy/mm/dd format, assumes date is valid and in the
// right format
istream& operator>>(istream &is, Date &d)
{
  char c;
  return is >> d.yyyy >> c >> d.mm >> c >> d.dd;
}

// print date in yyyy/mm/dd format
ostream& operator<< (ostream &os, const Date &d) {
  char t = os.fill('0');
  os << d.yyyy << '/' << setw(2) << d.mm << '/' << setw(2) << d.dd;
  os.fill(t);
  return os;
}
```

```
/*
 * Dijkstra's Algorithm
 *
 * Author: Howard Cheng
 * Reference:
 *   Ian Parberry's "Problems on Algorithms", page 102.
 *
 * Given a weight matrix representing a graph and a source vertex, this
 * algorithm computes the shortest distance, as well as path, to each
 * of the other vertices.  The paths are represented by an inverted list,
 * such that if v preceeds immediately before w in a path from the
 * source to vertex w, then the path P[w] is v.  The distances from
 * the source to v is given in D[v] (DISCONNECT if not connected).
 *
 * Call get_path to recover the path.
 *
 * Note: Dijkstra's algorithm only works if all weight edges are
 *       non-negative.
 *
 */

#include <iostream>
#include <algorithm>
#include <cassert>

using namespace std;

const int MAX_NODES = 10;
const int DISCONNECT = -1;

/* assume that D and P have been allocated */
void dijkstra(int graph[MAX_NODES][MAX_NODES], int n, int src, int D[],
              int P[])
{
  char used[MAX_NODES];
  int fringe[MAX_NODES];
  int f_size;
  int v, w, j, wj;
  int best, best_init;

  f_size = 0;
  for (v = 0; v < n; v++) {
    if (graph[src][v] != DISCONNECT && src != v) {
      D[v] = graph[src][v];
      P[v] = src;
      fringe[f_size++] = v;
      used[v] = 1;
    } else {
      D[v] = DISCONNECT;
      P[v] = -1;
      used[v] = 0;
    }
  }
  D[src] = 0;
  P[src] = -1;
  used[src] = 1;

  best_init = 1;
  while (best_init) {
    /* find unused vertex with smallest D */
    best_init = 0;
    for (j = 0; j < f_size; j++) {
      v = fringe[j];
      assert(D[v] != DISCONNECT);
      if (!best_init || D[v] < best) {
        best = D[v];
        w = v;
        wj = j;
        best_init = 1;
```



```
      }
    }

    if (best_init) {
      assert(D[w] != DISCONNECT);
      assert(fringe[wj] == w);

      /* get rid of w from fringe */
      f_size--;
      for (j = wj; j < f_size; j++) {
        fringe[j] = fringe[j+1];
      }

      /* update distances and add new vertices to fringe */
      for (v = 0; v < n; v++) {
        if (v != src && graph[w][v] != DISCONNECT) {
          if (D[v] == DISCONNECT || D[w] + graph[w][v] < D[v]) {
            D[v] = D[w] + graph[w][v];
            P[v] = w;
          } else if (D[w] + graph[w][v] == D[v]) {
            /* put tie-breaker here */
          }
          if (!used[v]) {
            used[v] = 1;
            fringe[f_size++] = v;
          }
        }
      }
    }
  }
  D[src] = 0;
}

int get_path(int v, int P[], int path[])
{
  int A[MAX_NODES];
  int i, k;

  k = 0;
  A[k++] = v;
  while (P[v] != -1) {
    v = P[v];
    A[k++] = v;
  }
  for (i = k-1; i >= 0; i--) {
    path[k-1-i] = A[i];
  }
  return k;
}

int main(void)
{
  int m, w, num;
  int i, j;
  int graph[MAX_NODES][MAX_NODES];
  int P[MAX_NODES][MAX_NODES], D[MAX_NODES][MAX_NODES];
  int path[MAX_NODES];

  /* clear graph */
  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      graph[i][j] = DISCONNECT;
    }
  }

  /* read graph */
  cin >> i >> j >> w;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
```

```
    graph[i][j] = graph[j][i] = w;
    cin >> i >> j >> w;
  }

  for (i = 0; i < MAX_NODES; i++) {
    dijkstra(graph, MAX_NODES, i, D[i], P[i]);
  }

  /* do queries */
  cin >> i >> j;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    cout << i << " " << j << ": " << D[i][j] << endl;
    for (m = j; m != -1; m = P[i][m]) {
      cout << " " << m;
    }
    cout << endl;
    num = get_path(j, P[i], path);
    for (m = 0; m < num; m++) {
      cout << " " << path[m];
    }
    cout << endl;
    cin >> i >> j;
  }

  return 0;
}
```

```
/*
 * Dijkstra's Algorithm for sparse graphs
 *
 * Author: Howard Cheng
 *
 * Given a weight matrix representing a graph and a source vertex, this
 * algorithm computes the shortest distance, as well as path, to each
 * of the other vertices.  The paths are represented by an inverted list,
 * such that if v preceeds immediately before w in a path from the
 * source to vertex w, then the path P[w] is v.  The distances from
 * the source to v is given in D[v] (-1 if not connected).
 *
 * Call get_path to recover the path.
 *
 * Note: Dijkstra's algorithm only works if all weight edges are
 *       non-negative.
 *
 * This version works well if the graph is not dense.  The complexity
 * is O((n + m) log (n + m)) where n is the number of vertices and
 * m is the number of edges.
 *
 */

#include <iostream>
#include <algorithm>
#include <vector>
#include <cassert>
#include <queue>

using namespace std;


struct Edge {
  int to;
  int weight;       // can be double or other numeric type
  Edge(int t, int w)
    : to(t), weight(w) { }
};

typedef vector<Edge>::iterator EdgeIter;

struct Graph {
  vector<Edge> *nbr;
  int num_nodes;
  Graph(int n)
    : num_nodes(n)
  {
    nbr = new vector<Edge>[num_nodes];
  }

  ~Graph()
  {
    delete[] nbr;
  }

  // note: There is no check on duplicate edge, so it is possible to
  // add multiple edges between two vertices
  //
  // If this is an undirected graph, be sure to add an edge both
  // ways
  void add_edge(int u, int v, int weight)
  {
    nbr[u].push_back(Edge(v, weight));
  }
};

/* assume that D and P have been allocated */
void dijkstra(const Graph &G, int src, vector<int> &D, vector<int> &P)
{
```



```
  typedef pair<int,int> pii;

  int n = G.num_nodes;
  vector<bool> used(n, false);
  priority_queue<pii, vector<pii>,  greater<pii> > fringe;

  D.resize(n);
  P.resize(n);
  fill(D.begin(), D.end(), -1);
  fill(P.begin(), P.end(), -1);

  D[src] = 0;
  fringe.push(make_pair(D[src], src));

  while (!fringe.empty()) {
    pii next = fringe.top();
    fringe.pop();
    int u = next.second;
    if (used[u]) continue;
    used[u] = true;

    for (EdgeIter it = G.nbr[u].begin(); it != G.nbr[u].end(); ++it) {
      int v = it->to;
      int weight = it->weight + next.first;
      if (used[v]) continue;
      if (D[v] == -1 || weight < D[v]) {
        D[v] = weight;
        P[v] = u;
        fringe.push(make_pair(D[v], v));
      }
    }
  }
}

int get_path(int v, const vector<int> &P, vector<int> &path)
{
  path.clear();
  path.push_back(v);
  while (P[v] != -1) {
    v = P[v];
    path.push_back(v);
  }
  reverse(path.begin(), path.end());
  return path.size();
}

int main(void)
{
  int n;
  while (cin >> n && n > 0) {
    Graph G(n);
    int u, v, w;

    while (cin >> u >> v >> w && !(u == -1 && v == -1 && w == -1)) {
      G.add_edge(u, v, w);
    }

    while (cin >> u >> v && !(u == -1 && v == -1)) {
      vector<int> D, P, path;
      dijkstra(G, u, D, P);
      get_path(v, P, path);

      cout << "distance = " << D[v] << endl;
      cout << "path = ";
      for (unsigned int i = 0; i < path.size(); i++) {
        cout << path[i] << ' ';
      }
      cout << endl;
    }
```

```
  }
  return 0;
}
```

```c
/*
 * Solution of system of linear diophantine equations
 *
 * Author: Howard Cheng
 * Date:   Nov 25, 2000
 * Reference:
 *
 * http://scicomp.ewha.ac.kr/netlib/tomspdf/
 *
 * Look at Algorithms 287 (sort of) and 288.
 *
 * Given a system of m linear diophantine equations in n unknowns,
 * this algorithm finds a particular solution as well as a basis for
 * the solution space of the homogeneous system, if they exist.  The
 * system is represented in matrix form as Ax = b where all entries
 * are integers.
 *
 * Function: diophantine_linsolve
 *
 * Input:
 *
 * A: an m x n matrix specifying the coefficients of each equation in
 *    each row (it is okay to have zero rows, or even have A = 0)
 * b: an m-dimensional vector specifying the right-hand side of the system
 * m: number of equations in the system
 * n: number of unknowns in the system
 *
 * Output:
 *
 * The function returns the dimension of the solution space of the
 * homogeneous system Ax = 0 (hom_dim) if it has a solution.
 * Otherwise, it returns -1.
 *
 * Other results returned in the parameters are:
 *
 * xp: an n-dimensional vector giving a particular solution
 * hom_basis: an n x n matrix whose first hom_dim columns form a basis
 *            of the solution space of the homogeneous system Ax = 0
 *
 * All solutions to Ax = b can be obtained by adding integer multiples
 * of the first hom_dim columns of hom_basis to xp.
 *
 * Note:
 *
 * The contents of A and b are not changed by this function.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#define MAX_N 50
#define MAX_M 50

int triangulate(int A[MAX_N+1][MAX_M+MAX_N+1], int m, int n, int cols)
{
  int ri, ci, i, j, k, pi, t;
  div_t d;

  ri = ci = 0;
  while (ri < m && ci < cols) {

    /* find smallest non-zero pivot */
    pi = -1;
    for (i = ri; i < m; i++) {
      if (A[i][ci] && (pi == -1 || abs(A[i][ci]) < abs(A[pi][ci]))) {
        pi = i;
      }
    }
```



```c
    if (pi == -1) {
      /* the entire column is 0, skip it */
      ci++;
    } else {
      k = 0;
      for (i = ri; i < m; i++) {
        if (i != pi) {
          d = div(A[i][ci], A[pi][ci]);
          if (d.quot) {
            k++;
            for (j = ci; j < n; j++) {
              A[i][j] -= d.quot*A[pi][j];
            }
          }
        }
      }
      if (!k) {
        /* swap the row to make it triangular...Alg 287 also switches the */
        /* sign, probably to preserve the sign of the minors.  I don’t    */
        /* think this is necessary for our purpose.                       */
        for (i = ci; i < n && ri != pi; i++) {
          t = A[ri][i];
          A[ri][i] = A[pi][i];
          A[pi][i] = t;
        }
        ri++;
        ci++;
      }
    }
  }
  return ri;
}

int diophantine_linsolve(int A[MAX_M][MAX_N], int b[MAX_M], int m, int n,
                         int xp[MAX_N], int hom_basis[MAX_N][MAX_N])
{
  int mat[MAX_N+1][MAX_M+MAX_N+1];
  int i, j, rank, d;

  /* form the work matrix */
  for (i = 0; i < m; i++) {
    mat[0][i] = -b[i];
  }
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      mat[j+1][i] = A[i][j];
    }
  }
  for (i = 0; i < n+1; i++) {
    for (j = 0; j < n+1; j++) {
      mat[i][j+m] = (i == j);
    }
  }

  /* triangluate the first n+1 x m+1 submatrix */
  rank = triangulate(mat, n+1, m+n+1, m+1);
  d = mat[rank-1][m];

  /* check for no solutions */
  if (d != 1 && d != -1) {
    /* no integer solutions */
    return -1;
  }
  /* check for inconsistent system */
  for (i = 0; i < m; i++) {
    if (mat[rank-1][i]) {
      return -1;
    }
  }
```

```c
  /* there is a solution, copy it to the result */
  for (i = 0; i < n; i++) {
    xp[i] = d*mat[rank-1][m+1+i];
    for (j = 0; j < n+1-rank; j++) {
      hom_basis[i][j] = mat[rank+j][m+1+i];
    }
  }

  return n+1-rank;
}

int main(void)
{
  int A[MAX_M][MAX_N], b[MAX_M], m, n, xp[MAX_N], hom_basis[MAX_N][MAX_N];
  int i, j, hom_dim;

  while (scanf("%d %d", &m, &n) == 2 && m > 0 && n > 0) {
    for (i = 0; i < m; i++) {
      printf("Enter equation %d:\n", i+1);
      for (j = 0; j < n; j++) {
        scanf("%d", &A[i][j]);
      }
      scanf("%d", &b[i]);
    }

    if ((hom_dim = diophantine_linsolve(A, b, m, n, xp, hom_basis)) >= 0) {
      printf("Particular solution:\n");
      for (i = 0; i < n; i++) {
        printf("%d ", xp[i]);
      }
      printf("\n");
      printf("hom_dim = %d\n", hom_dim);
      printf("Basis for Ax = 0:\n");
      for (j = 0; j < hom_dim; j++) {
        for (i = 0; i < n; i++) {
          printf("%d ", hom_basis[i][j]);
        }
        printf("\n");
      }
    } else {
      printf("No solution.\n");
    }

  }

  return 0;
}
```

```
//
// 3-D distances between point to point, point to line segment,
// line segment to line segment, and point to triangle.
//
// There are corresponding versions of the same code for distances
// between point to infinite lines, infinite line to infinite line,
// and point to infinite plane.
//
// It is assumed that segments/lines/triangles/plane are defined by
// distinct points (so the objects are not degenerate).
//
// They can be used for 2-D objects as well by setting the z coordinates
// to 0.
//
// Author: Howard Cheng
//

#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <cassert>

using namespace std;

const double PI = acos(-1.0);

struct Vector {
  double x, y, z;

  Vector(double xx = 0, double yy = 0, double zz = 0)
    : x(xx), y(yy), z(zz) { }

  Vector(const Vector &p1, const Vector &p2)
    : x(p2.x - p1.x), y(p2.y - p1.y), z(p2.z - p1.z) { }

  Vector(const Vector &p1, const Vector &p2, double t)
    : x(p1.x + t*p2.x), y(p1.y + t*p2.y), z(p1.z + t*p2.z) { }

  double norm() const {
    return sqrt(x*x + y*y + z*z);
  }

};

istream &operator>>(istream &is, Vector &p)
{
  return is >> p.x >> p.y >> p.z;
}

ostream &operator<<(ostream &os, const Vector &p)
{
  return os << "(" << p.x << "," << p.y << "," << p.z << ")";
}

double dot(const Vector &p1, const Vector &p2)
{
  return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
}

Vector cross(const Vector &p1, const Vector &p2)
{
  Vector v(p1.y*p2.z - p2.y*p1.z,
           p2.x*p1.z - p1.x*p2.z,
           p1.x*p2.y - p2.x*p1.y);
  return v;
}

// distance between two points
```



```
double dist_point_to_point(const Vector &p1, const Vector &p2)
{
  Vector p(p1, p2);
  return p.norm();
}

// angle between two vectors (in radians)
double angle(const Vector &p1, const Vector &p2)
{
  return acos(dot(p1, p2)/p1.norm()/p2.norm());
}

// distance from p to the line segment from a to b
double dist_point_to_segment(const Vector &p, const Vector &a,
                             const Vector &b)
{
  Vector u(a, p), v(a, b);
  double s = dot(u,v) / dot(v,v);

  if (s < 0 || s > 1) {
    return min(dist_point_to_point(p, a), dist_point_to_point(p, b));
  }

  Vector proj(a, v, s);
  return dist_point_to_point(proj, p);
}

// distance from p to the infinite line defined by a and b
double dist_point_to_line(const Vector &p, const Vector &a,
                          const Vector &b)
{
  Vector u(a, p), v(a, b);
  double s = dot(u,v) / dot(v,v);
  Vector proj(a, v, s);
  return dist_point_to_point(proj, p);
}

// distance from p to the triangle defined by a, b, c
double dist_point_to_triangle(const Vector &p, const Vector &a,
                              const Vector &b, const Vector &c)
{
  Vector u(a, p), v1(a, b), v2(a, c);
  Vector normal = cross(v1, v2);
  double s = dot(u, normal) / (normal.norm() * normal.norm());
  Vector proj(p, normal, -s);

  // check projection: inside if sum of angles is 2*pi
  Vector wa(proj, a), wb(proj, b), wc(proj, c);
  double a1 = angle(wa, wb);
  double a2 = angle(wa, wc);
  double a3 = angle(wb, wc);
  if (fabs(a1 + a2 + a3 - 2*PI) < 1e-8) {
    return dist_point_to_point(proj, p);
  } else {
    return min(dist_point_to_segment(p, a, b),
               min(dist_point_to_segment(p, a, c),
                   dist_point_to_segment(p, b, c)));
  }
}

// distance from p to the infinite plane defined by a, b, c
double dist_point_to_plane(const Vector &p, const Vector &a,
                           const Vector &b, const Vector &c)
{
  Vector u(a, p), v1(a, b), v2(a, c);
  Vector normal = cross(v1, v2);
  double s = dot(u, normal) / (normal.norm() * normal.norm());
  Vector proj(p, normal, -s);
  return dist_point_to_point(proj, p);
```

```
}

// distance from segment p1->q1 to p2->q2
double dist_segment_to_segment(const Vector &p1, const Vector &q1,
                               const Vector &p2, const Vector &q2)
{
  //
  // the points on the 1st line are p1 + t * v1
  // the points on the 2nd line are p2 + s * v2
  //
  //                             0 <= s, t <= 1
  //
  // squared distance is
  //
  // S = (p1.x - p2.x + t * v1.x - s * v2.x)^2 +
  //     (p1.y - p2.y + t * v1.y - s * v2.y)^2 +
  //     (p1.z - p2.z + t * v1.z - s * v2.z)^2
  //
  // deriviative wrt t and s are:
  //
  // 1/2 dS/dt = norm(v1)^2 * t - dot(v1, v2) * s + dot(v1, p1) - dot(v1, p2)
  // 1/2 dS/ds = -dot(v1, v2) * t + norm(v2)^2 * s - dot(v2, p1) + dot(v2, p2)
  //
  // solving for s and t with both derivatives = 0:
  //

  Vector v1(p1, q1), v2(p2, q2);
  Vector rhs(dot(v1, p2) - dot(v1, p1), dot(v2, p1) - dot(v2, p2));
  double det = v1.norm()*v1.norm()*v2.norm()*v2.norm() -
    dot(v1, v2)*dot(v1, v2);

  if (det < 1e-8) {
    // parallel lines (if v1 and v2 != 0)
    goto degenerate;
  } else {
    double t = (rhs.x*v2.norm()*v2.norm() + rhs.y * dot(v1, v2)) / det;
    double s = (v1.norm()*v1.norm()*rhs.y + dot(v1, v2) * rhs.x) / det;
    if (0 <= s && s <= 1 && 0 <= t && t <= 1) {
      Vector pp1(p1, v1, t), pp2(p2, v2, s);
      return dist_point_to_point(pp1, pp2);
    }
  }

 degenerate:
  return min(min(dist_point_to_segment(p1, p2, q2),
                 dist_point_to_segment(q1, p2, q2)),
             min(dist_point_to_segment(p2, p1, q1),
                 dist_point_to_segment(q2, p1, q1)));
}

// distance from infinite lines defined by p1->q1 and p2->q2
double dist_line_to_line(const Vector &p1, const Vector &q1,
                         const Vector &p2, const Vector &q2)
{
  //
  // the points on the 1st line are p1 + t * v1
  // the points on the 2nd line are p2 + s * v2
  //
  //                             0 <= s, t <= 1
  //
  // squared distance is
  //
  // S = (p1.x - p2.x + t * v1.x - s * v2.x)^2 +
  //     (p1.y - p2.y + t * v1.y - s * v2.y)^2 +
  //     (p1.z - p2.z + t * v1.z - s * v2.z)^2
  //
  // deriviative wrt t and s are:
  //
  // 1/2 dS/dt = norm(v1)^2 * t - dot(v1, v2) * s + dot(v1, p1) - dot(v1, p2)
```



```
  // 1/2 dS/ds = -dot(v1, v2) * t + norm(v2)^2 * s - dot(v2, p1) + dot(v2, p2)
  //
  // solving for s and t with both derivatives = 0:
  //

  Vector v1(p1, q1), v2(p2, q2);
  Vector rhs(dot(v1, p2) - dot(v1, p1), dot(v2, p1) - dot(v2, p2));
  double det = v1.norm()*v1.norm()*v2.norm()*v2.norm() -
    dot(v1, v2)*dot(v1, v2);

  if (det < 1e-8) {
    // parallel lines (if v1 and v2 != 0)
    return dist_point_to_line(p1, p2, q2);
  } else {
    double t = (rhs.x*v2.norm()*v2.norm() + rhs.y * dot(v1, v2)) / det;
    double s = (v1.norm()*v1.norm()*rhs.y + dot(v1, v2) * rhs.x) / det;
    Vector pp1(p1, v1, t), pp2(p2, v2, s);
    return dist_point_to_point(pp1, pp2);
  }
}

////////////////////////////////////////////////////////////////////////////
//
// This is the solution to 11836 (Star War)
//
////////////////////////////////////////////////////////////////////////////

void do_case()
{
  Vector t1[4], t2[4];
  for (int i = 0; i < 4; i++) {
    cin >> t1[i];
  }
  for (int i = 0; i < 4; i++) {
    cin >> t2[i];
  }

  double best = dist_point_to_point(t1[0], t2[0]);

  // vertex-face distance
  for (int i1 = 0; i1 < 4; i1++) {
    for (int j1 = 0; j1 < 4; j1++) {
      best = min(best, dist_point_to_triangle(t1[i1], t2[j1], t2[(j1+1)%4],
                                              t2[(j1+2)%4]));
      best = min(best, dist_point_to_triangle(t2[i1], t1[j1], t1[(j1+1)%4],
                                              t1[(j1+2)%4]));
    }
  }

  // edge-edge distance
  for (int i1 = 0; i1 < 4; i1++) {
    for (int i2 = i1+1; i2 < 4; i2++) {
      for (int j1 = 0; j1 < 4; j1++) {
        for (int j2 = j1+1; j2 < 4; j2++) {
          best = min(best, dist_segment_to_segment(t1[i1], t1[i2],
                                                   t2[j1], t2[j2]));
        }
      }
    }
  }

  cout << setprecision(2) << fixed << best << endl;
}

int main(void)
{
  int T;
  cin >> T;
  while (T-- > 0) {
```

```
    do_case();
  }
  return 0;
}
```

```
/*
 * Distance from a point to an (infinite) line.
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/cga1.html
 *
 * This routine computes the shortest distance from a point to a line.
 * ie. distance from point to its orthogonal projection onto the line.
 * Works even if the projection is not on the line (i.e. treat the line
 * as infinite line).
 *
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <cassert>

using namespace std;

struct Point {
  double x, y;
};

/* computes the distance from "c" to the line defined by "a" and "b" */
double dist_line(Point a, Point b, Point c)
{
  double L2, s;

  L2 = (b.x-a.x)*(b.x-a.x)+(b.y-a.y)*(b.y-a.y);
  assert(L2 > 0);
  s = ((a.y-c.y)*(b.x-a.x)-(a.x-c.x)*(b.y-a.y)) / L2;

  return fabs(s*sqrt(L2));
}

int main(void)
{
  Point a, b, c;

  while (cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y) {
    cout << "distance = " << fixed << setprecision(2) << dist_line(a, b, c)
         << endl;
  }
  return 0;
}
```

```c
/*
 * Computing the Day of the Week
 *
 * Author: Howard Cheng
 *
 * This routine computes the day of the week (Sunday = 0, Saturday = 6)
 * from the year, month, and day.
 *
 */

unsigned DOW(unsigned y, unsigned m, unsigned d)
{
      if (m < 3)
      {
            m += 13;
            y--;
      }
      else  m++;
      return (d + 26 * m / 10 + y + y / 4 - y / 100 + y / 400 + 6) % 7;
}

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
      int Day;
      void usage(void);
      unsigned d, m, y, days[] = {31, 29, 31, 30, 31, 30,
                                   31, 31, 30, 31, 30, 31};
      char *day[2][7] = {
            {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"},
            {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}
      };
      char *month[]  = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",};


      if (4 > argc)
            usage();
      y = atoi(argv[1]);
      m = atoi(argv[2]);
      d = atoi(argv[3]);
      if (!m || m > 12)
            usage();
      if (!d || d > days[m - 1])
            usage();
      if (y < 100)
            y += 1900;
      Day = DOW(y, m, d);
      printf("DOW returned %d, so %d %s %d is a %s\n",
             Day, d, month[m - 1], y, day[6 - 5][Day]);
      return 0;
}

void usage(void)
{
      puts("Usage: DOW yy[yy] mm dd");
      exit(-1);
}
```

```
/*
 * Euclidean Algorithm
 *
 * Author: Howard Cheng
 *
 * Given two integers, return their gcd.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int gcd(int a, int b)
{
  int r;

  /* unnecessary if a, b >= 0 */
  if (a < 0) {
    a = -a;
  }
  if (b < 0) {
    b = -b;
  }

  while (b) {
    r = a % b;
    a = b;
    b = r;
  }
  assert(a >= 0);
  return a;
}

int main(void)
{
  int a, b;

  while (cin >> a >> b) {
    cout << gcd(a, b) << endl;
  }
  return 0;
}
```

```
/*
 * Euler's Phi function:
 *
 * Author: Ethan Kim
 * Complexity: O(sqrt(n))
 *
 * Computes Euler's Phi(Totient) function; Given a positive n, computes
 * the number of positive integers that are <= n and relatively prime to n.
 *
 * For prime n, it is easy to see that phi(n)=n-1.
 * For powers of prime, phi(p^k)=p^(k-1) * (p-1).
 * Also, phi is multiplicative, so phi(pq)=phi(p)*phi(q), if p and q are
 * relatively prime.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int fast_exp(int b, int n)
{
  int res = 1;
  int x = b;

  while (n > 0) {
    if (n & 0x01) {
      n--;
      res *= x;
    } else {
      n >>= 1;
      x *= x;
    }
  }

  return res;
}

int phi(int n) {
  int k, res;
  long long p;

  assert(n > 0);

  res=1;
  for(k = 0; n % 2 == 0; k++) {
    n /= 2;
  }
  if (k)
    res *= fast_exp(2, k-1);

  for (p = 3; p*p <= n; p += 2) {
    for (k = 0; n % p == 0; k++) {
      n /= p;
    }
    if (k) {
      res *= fast_exp(p, k-1) * (p-1);
    }
  }
  if (n > 1) {
    res *= n-1;
  }
  return res;
}

int main(void) {
  int p;
  while(cin >> p && p) {
```



```
    cout << phi(p) << endl;
  }
  return 0;
}
```

```
/*
 * Finding an Eulerian Tour
 *
 * Author: Howard Cheng
 *
 * The routine eulerian() determines if a graph has an Eulerian tour.
 * That is, it checks that it is connected and all vertices have even
 * degree.  We assume that the graph is represented as an adjacency matrix
 * and the an auxillary array called "deg" gives the degree of the vertex.
 *
 * The routine eulerian_tour() returns one (arbitrary) Eulerian tour.
 * The tour is stored in an array of the vertices visited in the tour,
 * and the first and last vertex is the same.
 *
 * WARNING: eulerian_tour() destroys the graph as it uses edges.  If
 *          you need the graph back then you should save a copy.
 *
 * NOTE: converting this code for directed graphs should not be that much
 *       work.  You should also be able to convert this code for Eulerian
 *       paths.
 *
 */

#include <iostream>
#include <algorithm>
#include <cassert>

using namespace std;

const int NUM_VERTICES = 50;
const int NUM_EDGES = 1000;       /* maximum number of edges in graph */

int graph[NUM_VERTICES+1][NUM_VERTICES+1];
int deg[NUM_VERTICES+1];

void clear_graph(void)
{
  fill(deg, deg+NUM_VERTICES+1, 0);
  for (int i = 1; i <= NUM_VERTICES; i++) {
    fill(graph[i], graph[i]+NUM_VERTICES+1, 0);
  }
}

void visit(int v, char visited[])
{
  int w;

  visited[v] = 1;
  for (w = 1; w <= NUM_VERTICES; w++) {
    if (!visited[w] && graph[v][w] > 0) {
      visit(w, visited);
    }
  }
}

int connected(void)
{
  char visited[NUM_VERTICES+1];
  int i;

  fill(visited, visited+NUM_VERTICES+1, 0);
  for (i = 1; i <= NUM_VERTICES; i++) {
    if (deg[i] > 0) {
      visit(i, visited);
      break;
    }
  }
  for (i = 1; i <= NUM_VERTICES; i++) {
    if (deg[i] > 0 && !visited[i]) {
```



```
      return 0;
    }
  }
  return 1;
}

int eulerian(void)
{
  int i;
  for (i = 1; i <= NUM_VERTICES; i++) {
    if (deg[i] % 2 == 1) {
      return 0;
    }
  }
  return connected();
}

int find_tour(int start, int temp[])
{
  int len = 0;
  int next;

  temp[len++] = start;
  while (deg[start] > 0) {
    for (next = 1; next <= NUM_VERTICES; next++) {
      if (graph[start][next] > 0) {
        break;
      }
    }
    temp[len++] = next;
    graph[start][next]--; deg[start]--;
    graph[next][start]--; deg[next]--;
    start = next;
  }
  return len;
}

int graft_tour(int old[], int old_len, int tour[], int tour_len)
{
  int pos[NUM_VERTICES+1];
  int i, j, p1, p2;

  fill(pos, pos+NUM_VERTICES+1, -1);
  for (i = 0; i < old_len; i++) {
    pos[old[i]] = i;
  }
  for (i = 0; i < tour_len; i++) {
    if (pos[tour[i]] >= 0) {
      break;
    }
  }
  assert(i < tour_len);
  p1 = pos[tour[i]];
  p2 = i;
  for (i = old_len-1; i > p1; i--) {
    old[i+tour_len-1] = old[i];
  }
  for (i = p2+1, j = 0; i < tour_len-1; i++, j++) {
    old[p1+j+1] = tour[i];
  }
  for (i = 0; i <= p2; i++) {
    old[p1+j+1] = tour[i];
  }

  return old_len+tour_len-1;
}

int eulerian_tour(int tour[])
{
```

```
  int temp[NUM_EDGES+1];
  int tour_len, temp_len, first_time;
  int i, found;

  tour_len = temp_len = 0;
  first_time = 1;

  while (1) {
    found = 0;
    if (first_time) {
      for (i = 1; i <= NUM_VERTICES; i++) {
        if (deg[i] > 0) {
          found = 1;
          break;
        }
      }
    } else {
      /* this ensures that we can graft next tour on to existing one */
      for (i = 0; i < tour_len; i++) {
        if (deg[tour[i]] > 0) {
          found = 1;
          break;
        }
      }
      i = tour[i];
    }
    if (!found) {
      break;
    }

    if (first_time) {
      tour_len = find_tour(i, tour);
    } else {
      temp_len = find_tour(i, temp);
      tour_len = graft_tour(tour, tour_len, temp, temp_len);
    }
    first_time = 0;
  }
  return tour_len;
}

int main(void)
{
  int T, N, i, j, k;
  int u, v;
  int tour[NUM_EDGES+1], tour_len;

  cin >> T;
  for (i = 1; i <= T; i++) {
    clear_graph();
    if (i > 1) {
      cout << endl;
    }
    cout << "Case #" << i << endl;
    cin >> N;
    for (j = 0; j < N; j++) {
      cin >> u >> v;
      graph[u][v]++;
      graph[v][u]++;
      deg[u]++;
      deg[v]++;
    }

    if (eulerian()) {
      tour_len = eulerian_tour(tour);
      for (k = 0; k < tour_len-1; k++) {
        cout << tour[k] << " " << tour[k+1] << endl;
      }
    } else {
```



```
      cout << "some beads may be lost" << endl;
    }
  }
  return 0;
}
```

```
/*
 * Fast Exponentiation
 *
 * Author: Howard Cheng
 *
 * Given b and n, computes b^n quickly.
 *
 */

#include <iostream>

using namespace std;

int fast_exp(int b, int n)
{
  int res = 1;
  int x = b;

  while (n > 0) {
    if (n & 0x01) {
      n--;
      res *= x;
    } else {
      n >>= 1;
      x *= x;
    }
  }

  return res;
}

int main(void)
{
  int b, n;

  while (cin >> b >> n) {
    cout << b << "^" << n << " = " << fast_exp(b, n) << endl;
  }
  return 0;

}
```

```
/*
 * Fast Exponentiation mod m
 *
 * Author: Howard Cheng
 *
 * Given b, n, and m, computes b^n mod m quickly.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int fast_exp(int b, int n, int m)
{
  int res = 1;
  long long x = b;

  while (n > 0) {
    if (n & 0x01) {
      n--;
      res = (res * x) % m;
    } else {
      n >>= 1;
      x = (x * x) % m;
    }
  }

  return res;
}

int main(void)
{
  int b, n, m;

  while (cin >> b >> n >> m) {
    cout << b << "^" << n << " mod " << m << " = " << fast_exp(b, n, m)
         << endl;
  }
  return 0;

}
```

```
/*
 * Extended Euclidean Algorithm
 *
 * Author: Howard Cheng
 *
 * Given two integers, return their gcd and the cofactors to form the
 * gcd as a linear combination.
 *
 * a*s + b*t = gcd(a,b)
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int gcd(int a, int b, int &s, int &t)
{
  int r, r1, r2, a1, a2, b1, b2, q;
  int A = a;
  int B = b;

  /* unnecessary if a, b >= 0 */
  if (a < 0) {
    r = gcd(-a, b, s, t);
    s *= -1;
    return r;
  }
  if (b < 0) {
    r = gcd(a, -b, s, t);
    t *= -1;
    return r;
  }

  a1 = b2 = 1;
  a2 = b1 = 0;

  while (b) {
    assert(a1*A + a2*B == a);
    q = a / b;
    r = a % b;
    r1 = a1 - q*b1;
    r2 = a2 - q*b2;
    a = b;
    a1 = b1;
    a2 = b2;
    b = r;
    b1 = r1;
    b2 = r2;
  }

  s = a1;
  t = a2;
  assert(a >= 0);
  return a;
}

int main(void)
{
  int a, b, s, t, res;

  while (cin >> a >> b) {
    res = gcd(a, b, s, t);
    cout << res << " = " << a << " * " << s << " + "
         << b << " * " << t << endl;
  }
  return 0;
}
```

```
/*
 * Prime Factorization
 *
 * Author: Ethan Kim
 * Complexity: O(sqrt(n))
 *
 * Takes an integer and writes out the prime factorization in
 * ascending order. Prints -1 first, when given a negative integer.
 *
 * Note: you can change this code to store the factors in an array or process
 * the factors in other ways.
 *
 * Also, this code works for all integers even on INT_MIN (note that negating
 * INT_MIN does nothing, but it still works because INT_MIN is a power of 2).
 *
 */

#include <iostream>

using namespace std;

void factor(int n) {
  int printed = 0;
  long long p;

  if (n == 0 || n == 1) {
    cout << n << endl;
    return;
  }
  if (n < 0) {
    n *= -1;
    cout << "-1" << endl;
    printed = 1;
  }

  while (n % 2 == 0) {
    n/=2;
    cout << "2" << endl;
    printed = 1;
  }

  for (p = 3; p*p <= n; p += 2) {
    while (n % p == 0) {
      n /= p;
      cout << p << endl;
      printed = 1;
    }
  }

  if(n>1 || !printed)
    cout << n << endl;
}

int main(void) {
  int p;
  while(cin >> p && p != 0) {
    factor(p);
  }
  return 0;
}
```

```
// Gives the prime factorization of natural numbers (Uses probability)
//
// Author: Darcy Best
// Date  : January 7, 2010
//
// This should be used for factoring large integers. If you're
//   dealing with are small integers (N < 2^31), this is going
//   overboard. -- The normal sieve of Sieve of Eratosthenes is
//   usually good even for values up to 2^40.
//
// This implementation should only be used if you have numbers
//   larger than 2^40 (10^12) to factor.
//
// Notes:
//    - You need to handle N < 2 separately.
//    - Uses Miller-Rabin Primality Test
//         - This is a probabilistic test, there is a (1/4)^K
//             probability that a composite will return prime.
//             (K = 10 or 15 should be reasonably reliable).
//    - Uses Pollard's Rho algorithm to factor composites.
//         - I have also added Brent's improvement
//    - This program writes a number as a product of its primes,
//         with normal exponents (ie. "60 = 2^2 * 3 * 5")

#include <iostream>
#include <algorithm>
#include <set>
#include <map>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <vector>
using namespace std;

typedef long long int ll;

const ll MAX_NUM = 1e16;
const ll CB_RT = ll(pow(1.0*MAX_NUM,1.0/3)) + 2;
vector<ll> primes;
ll numPrimes;
ll c = 2;
const ll K = 10;

set<ll> lgPrimes;
map<ll,ll> semiPrimes;

ll gcd(ll a,ll b){
  ll r;
  while (b) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

ll mult_mod(ll x,ll y,ll n){
  ll res = 0;
  while(y){
    if(y % 2){
      y--;
      res += x;
      if(res >= n)
        res -= n;
    } else {
      x <<= 1;
      y >>= 1;
      if(x >= n)
        x -= n;
```



```
    }
  }
  return res;
}

ll fast_exp_mod(ll b, ll n,ll m){
  ll res = 1;
  ll x = b;

  while (n) {
    if (n % 2) {
      n--;
      res = mult_mod(res,x,m);
    } else {
      n >>= 1;
      x = mult_mod(x,x,m);
    }
  }

  return res;
}

void genSmallPrimes(){
  bool isPrime[CB_RT+7];
  for(int i=3;i<CB_RT;i+=2)
    isPrime[i] = true;

  primes.clear();
  primes.push_back(2);

  int i;
  for(i=3;i*i<CB_RT;i+=2)
    if(isPrime[i]){
      primes.push_back(i);
      for(int j=i*i;j<CB_RT;j+=(2*i))
        isPrime[j] = false;
    }

  for(;i<CB_RT;i+=2)
    if(isPrime[i])
      primes.push_back(i);
  numPrimes = primes.size();
}

ll F(ll x,ll n){
  x = mult_mod(x,x,n);
  x -= c;
  return (x < 0 ? x + n : x);
}

ll pollardRho(ll n){
  ll b,g,x,y,z;
 newC:
  c++;
  g = b = x = 1;
  while(g == 1){
    z = 1;
    y = x;
    for(ll i=0;i<b;i++){
      x = F(x,n);
      z = mult_mod(z,abs(x-y),n);
    }
    g = gcd(z,n);
    b <<= 1;
  }
  if(g == n || g == 0)
    goto newC;

  c = 2;
```

```
  return g;
}

bool miller(ll n){
  if (n == 2) return true;
  ll d = n-1;
  ll s = 0,a,x;
  while(d % 2 == 0){
    d >>= 1;
    s++;
  }
  for(int i=0;i<K;i++){
    a = (rand() % (n-2)) + 2; // [2,n-1];
    x = fast_exp_mod(a,d,n);
    if(x == 1 || x == n-1)
      continue;
    for(ll r=1;r<s;r++){
      x = mult_mod(x,x,n);
      if(x == 1)
        return false;
      if(x == n-1)
        goto nextK;
    }
    return false;
  nextK:;
  }
  return true;
}

void printEntry(bool& printed,ll prime,int ex){
  if(!printed)
    printed = true;
  else
    cout << " * ";
  cout << prime;
  if(ex > 1)
    cout << "^" << ex;
}

void factor(ll x){
  cout << x << " = ";
  bool printed = false;

  for(int i=0;i<numPrimes;i++)
    if(x % primes[i] == 0){
      int ex = 0; // Exponent
      do{
        x /= primes[i];
        ex++;
      } while(x % primes[i] == 0);
      printEntry(printed,primes[i],ex);
    }
  if(x == 1){
    cout << endl;
    return;
  }

  // lgPrimes and semiPrimes are useful if there
  //   is a lot repetition of large primes/semi-primes
  //   in the test data
  if(lgPrimes.find(x) != lgPrimes.end()){
    printEntry(printed,x,1);
    cout << endl;
    return;
  }

  if(semiPrimes.find(x) != semiPrimes.end()){
    ll lgFac = semiPrimes[x];
    printEntry(printed,x/lgFac,1);
```



```
    printEntry(printed,lgFac,1);
    cout << endl;
    return;
  }

  if(miller(x)){ // if x is prime
    printEntry(printed,x,1);
    cout << endl;
    lgPrimes.insert(x);
    return;
  }

  // Pollard's Rho does not work well with squares,
  //   so we'll check for it manually.
  ll sqrtX = ll(sqrt(x) + 0.1);
  if(sqrtX*sqrtX == x){
    printEntry(printed,sqrtX,2);
    cout << endl;
    return;
  }

  ll smFac = pollardRho(x);
  ll lgFac = x/smFac;
  if(lgFac < smFac)
    swap(smFac,lgFac);
  printEntry(printed,smFac,1);
  printEntry(printed,lgFac,1);
  cout << endl;
  semiPrimes[x] = lgFac;
}

int main(){
  genSmallPrimes();
  srand((unsigned int) time(NULL));
  ll T,N;
  cin >> T;
  while(T--){
    cin >> N;
    factor(N);
  }
  return 0;
}
```

```
/*
 * Fenwick Tree
 *
 * Author: Howard Cheng
 * Reference:
 *
 *   Fenwick, P.M. "A New Data Structure for Cumulative Frequency Tables."
 *   Software---Practice and Experience, 24(3), 327-336 (March 1994).
 *
 * This code has been tested on UVa 11525 and 11610.
 *
 * Fenwick trees are data structures that allows the maintainence of
 * cumulative sum tables dynamically.  The following operations
 * are supported:
 *
 * - Initialize the tree from a list of N integers:                O(N log N)
 *
 * - Read the cumulative sum at index 0 <= k < N:                   O(log k)
 *
 * - Read the entry at index 0 <= k < N:                            O(log N)
 *
 * - Increment/decrement an entry at index 0 <= k < N in the list:  O(log N)
 *
 * - Given a value, find an index such that the cumulative sum at
 *   that position is the value:                                    O(log N)
 *
 * The space usage is at most 2*N for N input entries.
 *
 * NOTE: it is assumed that all entries are non-negative (even after a
 *       decrement operation).
 *
 */

#include <vector>
#include <cassert>

using namespace std;

class FenwickTree
{
public:
  FenwickTree(int n = 0)
    : N(n), tree(n)
  {
    iBM = 1;
    while (iBM < N) {
      iBM *= 2;
    }
    tree.resize(iBM+1);
    fill(tree.begin(), tree.end(), 0);
  }

  // initialize the tree with the given array of values
  FenwickTree(int val[], int n)
    : N(n)
  {
    iBM = 1;
    while (iBM < N) {
      iBM *= 2;
    }

    tree.resize(iBM+1);
    fill(tree.begin(), tree.end(), 0);
    for (int i = 0; i < n; i++) {
      assert(val[i] >= 0);
      incEntry(i, val[i]);
    }
  }
```



```
  // increment the entry at position idx by val (use negative val for
  // decrement).  All affected cumulative sums are updated.
  void incEntry(int idx, int val)
  {
    assert(0 <= idx && idx < N);
    if (idx == 0) {
      tree[idx] += val;
    } else {
      do {
        tree[idx] += val;
        idx += idx & (-idx);
      } while (idx < (int)tree.size());
    }
  }

  // return the cumulative sum val[0] + val[1] + ... + val[idx]
  int cumulativeSum(int idx) const
  {
    assert(0 <= idx && idx < (int)tree.size());
    int sum = tree[0];
    while (idx > 0) {
      sum += tree[idx];
      idx &= idx-1;
    }
    return sum;
  }

  // return the entry indexed by idx
  int getEntry(int idx) const
  {
    assert(0 <= idx && idx < N);
    int val, parent;
    val = tree[idx];
    if (idx > 0) {
      parent = idx & (idx-1);
      idx--;
      while (parent != idx) {
        val -= tree[idx];
        idx &= idx-1;
      }
    }
    return val;
  }

  // return the largest index such that the cumulative frequency is
  // what is given, or -1 if it is not found
  //
  int getIndex(int sum) const
  {
    int orig = sum;
    if (sum < tree[0]) return -1;
    sum -= tree[0];

    int idx = 0;
    int bitmask = iBM;

    while (bitmask != 0 && idx < (int)tree.size()-1) {
      int tIdx = idx + bitmask;
      if (sum >= tree[tIdx]) {
        idx = tIdx;
        sum -= tree[tIdx];
      }
      bitmask >>= 1;
    }

    if (sum != 0) {
      return -1;
    }
```

```
    idx = min(N-1, idx);
    return (cumulativeSum(idx) == orig) ? idx : -1;
  }

private:
  int N, iBM;
  vector<int> tree;
};
```

```
/*
 * Solution of systems of linear equations over the integers
 *
 * Author: Howard Cheng
 * Reference:
 *  K.O. Geddes, S.R. Czapor, G. Labahn.  "Algorithms for Computer Algebra."
 *     Kluwer Academic Publishers, 1992, pages 393-399.  ISBN 0-7923-9259-0
 *
 * The routine fflinsolve solves the system Ax = b where A is an n x n matrix
 * of integers and b is an n-dimensional vector of integers.
 *
 * The inputs to fflinsolve are the matrix A, the dimension n, and an
 * output array to store the solution x_star = det(A)*x.  The function
 * also returns the det(A).  In the case that det(A) = 0, the solution
 * vector is undefined.
 *
 * Note that the matrix A and b may be modified.
 *
 */

#include <iostream>

using namespace std;

const int MAX_N = 10;

int fflinsolve(int A[MAX_N][MAX_N], int b[], int x_star[], int n)
{
  int sign, d, i, j, k, k_c, k_r, pivot, t;

  sign = d = 1;

  for (k_c = k_r = 0; k_c < n; k_c++) {
    /* eliminate column k_c */

    /* find nonzero pivot */
    for (pivot = k_r; pivot < n && !A[pivot][k_r]; pivot++)
      ;

    if (pivot < n) {
      /* swap rows pivot and k_r */
      if (pivot != k_r) {
        for (j = k_c; j < n; j++) {
          t = A[pivot][j];
          A[pivot][j] = A[k_r][j];
          A[k_r][j] = t;
        }
        t = b[pivot];
        b[pivot] = b[k_r];
        b[k_r] = t;

        sign *= -1;
      }

      /* do elimination */
      for (i = k_r+1; i < n; i++) {
        for (j = k_c+1; j < n; j++) {
          A[i][j] = (A[k_r][k_c]*A[i][j]-A[i][k_c]*A[k_r][j])/d;
        }
        b[i] = (A[k_r][k_c]*b[i]-A[i][k_c]*b[k_r])/d;
        A[i][k_c] = 0;
      }
      if (d) {
        d = A[k_r][k_c];
      }
      k_r++;
    } else {
      /* entire column is 0, det(A) = 0 */
      d = 0;
```



```
    }
  }

  if (!d) {
    for (k = k_r; k < n; k++) {
      if (b[k]) {
        /* inconsistent system */
        cout << "Inconsistent system." << endl;
        return 0;
      }
    }
    /* multiple solutions */
    cout << "More than one solution." << endl;
    return 0;
  }

  /* now backsolve */
  for (k = n-1; k >= 0; k--) {
    x_star[k] = sign*d*b[k];
    for (j = k+1; j < n; j++) {
      x_star[k] -= A[k][j]*x_star[j];
    }
    x_star[k] /= A[k][k];
  }

  return sign*d;
}

int main(void)
{
  int A[MAX_N][MAX_N], x_star[MAX_N], b[MAX_N];
  int n, i, j;
  int det;

  while (cin >> n && 0 < n && n <= MAX_N) {
    cout << "Enter A:" << endl;
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        cin >> A[i][j];
      }
    }
    cout << "Enter b:" << endl;
    for (i = 0; i < n; i++) {
      cin >> b[i];
    }
    if (det = fflinsolve(A, b, x_star, n)) {
      cout << "det = " << det << endl;
      cout << "x_star = ";
      for (i = 0; i < n; i++) {
        cout << x_star[i] << " ";
      }
      cout << endl;
    } else {
      cout << "A is singular." << endl;
    }
  }
  return 0;
}
```

```
// Compute nth Fibonacci number with matrix exponentiation
//
// Time complexity: O(log(n))
//
// Author: Cody Barnson
//
// Warning: 46th Fibonacci number (i.e. fib(46)) is largest
// that will fit into signed 32-bit integer; use long long if need
// longer.  Or perhaps the problem asks for Fibonacci number mod m

int f[1000];
int fib(int n) {
  if (n < 2) return n;
  if (f[n]) return f[n];

  int k = (n + 1) / 2;
  f[n] = (n & 1) ? fib(k) * fib(k) + fib(k - 1) * fib(k - 1)
                 : (2 * fib(k - 1) + fib(k)) * fib(k);
  return f[n];
}
```

```
/*
 * Floyd's Algorithm
 *
 * Author: Howard Cheng
 *
 * The following code takes a graph stored in an adjacency matrix "graph",
 * and returns the shortest distance from node i to node j in dist[i][j].
 * We assume that the weights of the edges is not DISCONNECT, and the
 * DISCONNECT constant is used to indicate the absence of an edge.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

const int MAX_NODES = 26;
const int DISCONNECT = -1;

int graph[MAX_NODES][MAX_NODES];
int dist[MAX_NODES][MAX_NODES];

void floyd(void)
{
  int i, j, k;

  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      dist[i][j] = graph[i][j];
    }
  }

  for (k = 0; k < MAX_NODES; k++) {
    for (i = 0; i < MAX_NODES; i++) {
      for (j = 0; j < MAX_NODES; j++) {
        if (dist[i][k] != DISCONNECT && dist[k][j] != DISCONNECT) {
          int temp = dist[i][k] + dist[k][j];
          if (dist[i][j] == DISCONNECT || dist[i][j] > temp) {
            dist[i][j] = temp;
          }
        }
      }
    }
  }

  for (i = 0; i < MAX_NODES; i++) {
    dist[i][i] = 0;
  }
}

int main(void)
{
  int w;
  int i, j;

  /* clear graph */
  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      graph[i][j] = DISCONNECT;
    }
  }

  /* read graph */
  cin >> i >> j >> w;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    graph[i][j] = graph[j][i] = w;
    cin >> i >> j >> w;
```



```
  }

  floyd();

  /* do queries */
  cin >> i >> j;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    cout << i << " " << j << ":" << dist[i][j] << endl;
    cin >> i >> j;
  }

  return 0;
}
```

```
/*
 * Floyd's Algorithm
 *
 * Author: Howard Cheng
 *
 * The following code takes a graph stored in an adjacency matrix "graph",
 * and returns the shortest distance from node i to node j in dist[i][j].
 * We assume that the weights of the edges is not DISCONNECT, and the
 * DISCONNECT constant is used to indicate the absence of an edge.
 *
 * Call extract_path to return the path, as well as its length (in terms
 * of vertices).  The length is -1 if no such path exists.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

const int MAX_NODES = 26;
const int DISCONNECT = -1;

int graph[MAX_NODES][MAX_NODES];
int dist[MAX_NODES][MAX_NODES];
int succ[MAX_NODES][MAX_NODES];

void floyd(void)
{
  int i, j, k;

  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      dist[i][j] = graph[i][j];
      if (i == j || graph[i][j] == DISCONNECT) {
        succ[i][j] = -1;
      } else {
        succ[i][j] = j;
      }
    }
  }

  for (k = 0; k < MAX_NODES; k++) {
    for (i = 0; i < MAX_NODES; i++) {
      for (j = 0; j < MAX_NODES; j++) {
        if (i != k && dist[i][k] != DISCONNECT && dist[k][j] != DISCONNECT) {
          int temp = dist[i][k] + dist[k][j];
          if (dist[i][j] == DISCONNECT || dist[i][j] > temp) {
            dist[i][j] = temp;
            succ[i][j] = succ[i][k];
          } else if (dist[i][j] == temp && succ[i][k] < succ[i][j]) {
            /* put tie-breaking on paths here */

            /* e.g. the test above chooses lexicographically smallest */
            /*      paths, but ignores the number of vertices in the  */
            /*      path.  To really do lexicographically sorting     */
            /*      properly, you also need to have len[i][j] which   */
            /*      can be computed easily as well.                   */
            succ[i][j] = succ[i][k];
          }
        }
      }
    }
  }

  for (i = 0; i < MAX_NODES; i++) {
    dist[i][i] = 0;
  }
}
```



```
int extract_path(int u, int v, int path[])
{
  int len = 0;

  if (dist[u][v] == DISCONNECT) {
    return -1;
  }

  path[len++] = u;
  while (u != v) {
    u = succ[u][v];
    path[len++] = u;
  }

  return len;
}

int main(void)
{
  int m, w, i, j;
  int path[MAX_NODES], len;

  /* clear graph */
  for (i = 0; i < MAX_NODES; i++) {
    for (j = 0; j < MAX_NODES; j++) {
      graph[i][j] = DISCONNECT;
    }
  }

  /* read graph */
  cin >> i >> j >> w;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    graph[i][j] = /*graph[j][i] =*/ w;
    cin >> i >> j >> w;
  }

  floyd();

  /* do queries */
  cin >> i >> j;
  while (!(i == -1 && j == -1)) {
    assert(0 <= i && i < MAX_NODES && 0 <= j && j < MAX_NODES);
    cout << i << " " << j << ":" << dist[i][j] << endl;
    len = extract_path(i, j, path);
    for (m = 0; m < len; m++) {
      if (m) {
        cout << " ";
      }
      cout << path[m];
    }
    cout << endl;
    cin >> i >> j;
  }

  return 0;
}
```

```
// Converts a fraction (with integral numerator and denominator)
//   to its decimal expansion.
//
// Author: Darcy Best
// Date  : August 22, 2010
//
// Since we are dealing with rational numbers, one of two cases
//   occur:
//      1. The number will terminate
//      2. The number will repeat
//
// The algorithm is O(D) where D is the absolute value of the
//   denominator.

#include <iostream>
#include <string>
#include <algorithm>
#include <cstdlib>
#include <cassert>
using namespace std;

const int MAX_DENOM = 1001;

string itoa(int x){
  string ans;
  while(x){
    ans += (x % 10) + '0';
    x /= 10;
  }
  reverse(ans.begin(),ans.end());
  return (ans.length() ? ans : "0");
}

int firstSeen[MAX_DENOM];

void frac2dec(int numer,int denom,string& decimal,int& numRepDigs){
  assert(denom != 0);

  // Determine if it is a plus or a minus
  decimal = "";
  if(numer < 0 && denom >= 0 || numer >= 0 && denom < 0){
    decimal += "-";
  } else {
    decimal += "+";
  }
  numer = abs(numer);
  denom = abs(denom);

  // Left of the decimal point
  decimal += itoa(numer / denom);
  numer %= denom;
  if(!numer){
    numRepDigs = 0;
    return;
  }

  // Add the decimal point
  decimal += '.';

  // Right of the decimal point
  fill(firstSeen,firstSeen+denom,-1);
  int rem = numer;
  while(rem != 0 && firstSeen[rem] == -1){
    firstSeen[rem] = decimal.length();
    rem *= 10;

    decimal += itoa(rem / denom);
    rem %= denom;
  }
```



```
  numRepDigs = (rem ? decimal.length() - firstSeen[rem] : 0);
}

int main(){
  int numerator,denominator,repDigs;
  string decimal;
  while(cin >> numerator >> slash >> denominator){
    frac2dec(numerator,denominator,decimal,repDigs);
    cout << numerator << "/" << denominator << "=" << decimal << endl;
    if(repDigs == 0)
      cout << "This expansion terminates." << endl;
    else
      cout << "The last " << repDigs << " digits repeat forever." << endl;
    cout << endl;
  }
  return 0;
}
```

```
//
// Fraction implementation
//
// Author: Darcy Best
//
// Does NOT ever check for division by 0.
// Division by 0 will only cause a runtime error if you use the
//   toDouble() function.
//

#include <iostream>
#include <cstdlib>
using namespace std;

// Change this to whatever integer data type will prevent overflow
//   - BigInteger works with this class
typedef long long int dataType;

class Fraction{
public:
  Fraction(dataType num=0,dataType denom=1);

  double toDouble() const;

  void reduce();

  // Changes the fraction itself.
  void selfReciprocal();

  // Returns a new fraction, leaving the original.
  Fraction reciprocal() const;

  Fraction& operator+=(const Fraction& x);
  Fraction& operator-=(const Fraction& x);
  Fraction& operator*=(const Fraction& x);
  Fraction& operator/=(const Fraction& x);

  bool operator<(const Fraction& x) const;
  bool operator==(const Fraction& x) const;

  dataType num,denom;
};

Fraction operator+(const Fraction& x,const Fraction& y);
Fraction operator-(const Fraction& x,const Fraction& y);
Fraction operator*(const Fraction& x,const Fraction& y);
Fraction operator/(const Fraction& x,const Fraction& y);

istream& operator>>(istream& is,Fraction& x);
ostream& operator<<(ostream& os,const Fraction& x);

Fraction::Fraction(dataType n,dataType d){
  if(d < 0){
    num = -n;
    denom = -d;
  } else {
    num = n;
    denom = d;
  }
  reduce();
}

double Fraction::toDouble() const{
  return 1.0*num/denom;
}

// Howard's GCD function with no checks
dataType gcd(dataType a, dataType b)
{
```



```
  dataType r;
  while (b) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

void Fraction::reduce(){
  dataType g = gcd(abs(num),denom);
  num /= g;
  denom /= g;
}

void Fraction::selfReciprocal(){
  swap(num,denom);
  if (denom < 0) {
    num = -num;
    denom = -denom;
  }
}

Fraction Fraction::reciprocal() const{
  return Fraction(denom,num);
}

// Overflow potential in the denominator.
// I've tried to factor out as much as possible before,
//   But be careful.
//
//   (w)/(a*g) + (z)/(b*g)
// = (w*b)/(a*g*b) + (a*z)/(a*g*b)
// = (w*b + a*z)/(a*g*b)
Fraction& Fraction::operator+=(const Fraction& x){
  dataType g = gcd(denom,x.denom);

  dataType a = denom / g;
  dataType b = x.denom / g;

  num = num * b + x.num * a;
  denom *= b;

  reduce();

  return (*this);
}

Fraction& Fraction::operator-=(const Fraction& x){
  dataType g = gcd(denom,x.denom);
  dataType a = denom / g;
  dataType b = x.denom / g;

  num = num * b - x.num * a;
  denom *= b;

  reduce();
  return (*this);
}

Fraction& Fraction::operator*=(const Fraction& x){
  num *= x.num;
  denom *= x.denom;
  reduce();
  return (*this);
}

Fraction& Fraction::operator/=(const Fraction& x){
  num *= x.denom;
```

```
  denom *= x.num;

  if(denom < 0){
    num = -num;
    denom = -denom;
  }
  reduce();
  return (*this);
}

// Careful with overflow. If it is an issue, you can compare the
// double values, but you SHOULD check for equality BEFORE converting
bool Fraction::operator<(const Fraction& x) const{
  return (num*x.denom) < (x.num*denom);
}

bool Fraction::operator==(const Fraction& x) const{
  return (num == x.num) && (denom == x.denom);
}

Fraction operator+(const Fraction& x,const Fraction& y){
  Fraction a(x);
  a += y;
  return a;
}

Fraction operator-(const Fraction& x,const Fraction& y){
  Fraction a(x);
  a -= y;
  return a;
}

Fraction operator*(const Fraction& x,const Fraction& y){
  Fraction a(x);
  a *= y;
  return a;
}

Fraction operator/(const Fraction& x,const Fraction& y){
  Fraction a(x);
  a /= y;
  return a;
}

// Note that you can read in Fractions of two forms:
// a/b (With any number of space between a,/,b) - The input "points" to
//      the NEXT character after the denom (White space or not)
// c   (Just an integer - The input "points" to the next NON-WHITE SPACE
//      character. Careful when mixing this with getline.)
istream& operator>>(istream& is,Fraction& x){
  is >> x.num;

  char ch;
  is >> ch;
  if(ch != '/'){
    is.putback(ch);
    x.denom = 1;
  } else {
    is >> x.denom;
    if(x.denom < 0){
      x.num = -x.num;
      x.denom = -x.denom;
    }
    x.reduce();
  }

  return is;
}
```



```
// Will output 5 for 5/1 and 0 for 0/1. If you want always
//    fractions, get rid of the if statement
ostream& operator<<(ostream& os,const Fraction& x){
  os << x.num;
  if(x.num != 0 && x.denom != 1)
    os << '/' << x.denom;
  return os;
}

int main(){
  Fraction x,y;
  while(cin >> x >> y){
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "x+y= " << x+y << endl;
    cout << "x-y= " << x-y << endl;
    cout << "x*y= " << x*y << endl;
    cout << "x/y= " << x/y << endl;
    cout << endl;
  }
  return 0;
}
```

```
// Great Circle distance between two points using Heaverside formula
//
// Author: Howard Cheng
// Reference: http://mathforum.org/library/drmath/view/51879.html
//
// Given two points specified by their latitudes and longitudes, as well
// as the radius of the sphere, return the shortest distance between the
// two points along the surface of the sphere.
//
// latitude should be between -90 to 90 degrees (inclusive), and
// longitude should be between -180 to 180 degrees (inclusive)
//
// There are also routines that will convert between cartesian coordinates
// (x,y,z) and spherical coordinates (latitude, longitude, radius).
//

#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

const double PI = acos(-1.0);

double greatcircle(double lat1, double long1, double lat2, double long2,
                   double radius)
{
  lat1 *= PI/180.0;
  lat2 *= PI/180.0;
  long1 *= PI/180.0;
  long2 *= PI/180.0;

  double dlong = long2 - long1;
  double dlat = lat2 - lat1;
  double a = sin(dlat/2)*sin(dlat/2) +
    cos(lat1)*cos(lat2)*sin(dlong/2)*sin(dlong/2);
  return radius * 2 * atan2(sqrt(a), sqrt(1-a));
}

void longlat2cart(double lat, double lon, double radius,
                  double &x, double &y, double &z)
{
  lat *= PI/180.0;
  lon *= PI/180.0;
  x = radius * cos(lat) * cos(lon);
  y = radius * cos(lat) * sin(lon);
  z = radius * sin(lat);
}

void cart2longlat(double x, double y, double z,
                  double &lat, double &lon, double &radius)
{
  radius = sqrt(x*x + y*y + z*z);
  lat = (PI/2 - acos(z / radius)) * 180.0 / PI;
  lon = atan2(y, x) * 180.0 / PI;
}

int main(void)
{
  int T;
  cin >> T;
  while (T-- > 0) {
    const double radius = 6371009;
    double lat1, long1, lat2, long2;

    cin >> lat1 >> long1 >> lat2 >> long2;

    double x1, y1, z1, x2, y2, z2;
```



```
    longlat2cart(lat1, long1, radius, x1, y1, z1);
    longlat2cart(lat2, long2, radius, x2, y2, z2);

    double d1 = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2));
    double d2 = greatcircle(lat1, long1, lat2, long2, radius);

    cout << fixed << setprecision(0) << d2 - d1 << endl;

    double radius1;
    cart2longlat(x1, y1, z1, lat1, long1, radius1);
    cout << lat1 << ' ' << long1 << ' ' << radius1 << endl;
  }
  return 0;
}
```

```
// Heron's formula
//
// Computes the area of a triangle given the lengths of the three sides.
//
// Author: Howard Cheng
//

#include <iostream>
#include <iomanip>
#include <utility>
#include <cmath>

using namespace std;

// the lengths of the three sides are a, b, and c.  The routine returns
// the area of the triangle, or -1 if the three lengths do not make a
// triangle.
double area_heron(double a, double b, double c)
{
  if (a < b) swap(a, b);
  if (a < c) swap(a, c);
  if (b < c) swap(b, c);
  if (c < a - b) return -1;
  return sqrt((a+b+c)*(c-a+b)*(c+a-b)*(a+b-c))/4.0;
}

int main(void)
{
  double a, b, c;

  while (cin >> a >> b >> c) {
    cout << fixed << setprecision(4) << area_heron(a, b, c) << endl;
  }

  return 0;
}
```

```
/*
 * Maximum/minimum weight bipartite matching
 *
 * Author: Howard Cheng
 * Reference:
 *  http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=hungarianAlgorithm
 *
 * This file contains routines for computing the maximum/minimum weight
 * bipartite matching.
 *
 * It is assumed that each half of the graph has exactly N vertices, labelled
 * 0 to N-1.  The weight between vertex i on the left and vertex j on the
 * right is stored in G[i][j].  The cost of the optimal matching is returned,
 * and matching[i] is the vertex on the right that is matched to vertex i
 * on the left.
 *
 * If an edge is absent, the corresponding edge weight should be:
 *
 *   INT_MIN   if maximum weight matching is desired
 *   INT_MAX   if minimum weight matching is desired
 *
 * This is an implementation of the Hungarian algorithm.  The complexity
 * is O(N^3).
 *
 */

#include <iostream>
#include <algorithm>
#include <queue>
#include <cassert>
#include <climits>

using namespace std;

const int MAX_N = 3;

void update_labels(int lx[MAX_N], int ly[MAX_N], bool S[MAX_N], bool T[MAX_N],
                   int slack[MAX_N], int N)
{
  int delta;
  bool delta_init = false;

  for (int y = 0; y < N; y++) {
    if (T[y]) continue;
    delta = delta_init ? min(delta, slack[y]) : slack[y];
    delta_init = true;
  }
  for (int x = 0; x < N; x++) {
    if (S[x]) lx[x] -= delta;
  }
  for (int y = 0; y < N; y++) {
    if (T[y]) {
      ly[y] += delta;
    } else {
      slack[y] -= delta;
    }
  }
}

void add_to_tree(int x, int prevx, int G[MAX_N][MAX_N], bool S[MAX_N],
                 int prev[MAX_N], int lx[MAX_N], int ly[MAX_N],
                 int slack[MAX_N], int slackx[MAX_N], int N)
{
  S[x] = true;
  prev[x] = prevx;
  for (int y = 0; y < N; y++) {
    int temp = (G[x][y] == INT_MIN) ? INT_MAX : lx[x] + ly[y] - G[x][y];
    if (temp < slack[y]) {
      slack[y] = temp;
```



```
      slackx[y] = x;
    }
  }
}

int max_weight_matching(int G[MAX_N][MAX_N], int N, int matching[MAX_N])
{
  int revmatch[MAX_N];         // match from right to left
  int max_match = 0;           // number of vertices in current matching

  fill(matching, matching+N, -1);
  fill(revmatch, revmatch+N, -1);

  // find an initial feasible labelling
  int lx[MAX_N], ly[MAX_N];
  fill(ly, ly+N, 0);
  for (int x = 0; x < N; x++) {
    lx[x] = *max_element(G[x], G[x]+N);
  }

  // now repeatedly find alternating tree, augment, and relabel
  while (max_match < N) {
    queue<int> q;
    bool S[MAX_N], T[MAX_N];
    int prev[MAX_N];
    fill(S, S+N, false);
    fill(T, T+N, false);
    fill(prev, prev+N, -1);

    // find root of alternating tree
    int root = find(matching, matching+N, -1) - matching;
    q.push(root);
    prev[root] = -2;
    S[root] = true;

    int slack[MAX_N], slackx[MAX_N];
    for (int y = 0; y < N; y++) {
      slack[y] = (G[root][y] == INT_MIN) ? INT_MAX :
        lx[root] + ly[y] - G[root][y];
      slackx[y] = root;
    }

    bool path_found = false;
    int x, y;
    while (!path_found) {

      // build alternating tree with BFS
      while (!path_found && !q.empty()) {
        x = q.front();
        q.pop();
        for (y = 0; y < N; y++) {
          // go through edges in equality graph
          if (G[x][y] == lx[x] + ly[y] && !T[y]) {
            if (revmatch[y] == -1) {
              path_found = true;
              break;
            }
            T[y] = true;
            q.push(revmatch[y]);
            add_to_tree(revmatch[y], x, G, S, prev, lx, ly, slack, slackx, N);
          }
        }
      }
      if (path_found) break;

      // no augmenting path, update the labels
      update_labels(lx, ly, S, T, slack, N);
      while (!q.empty()) {
        q.pop();
```

```
      }
      for (y = 0; y < N; y++) {
        if (!T[y] && slack[y] == 0) {
          if (revmatch[y] == -1) {
            x = slackx[y];
            path_found = true;
            break;
          } else {
            T[y] = true;
            if (!S[revmatch[y]]) {
              q.push(revmatch[y]);
              add_to_tree(revmatch[y], slackx[y], G, S, prev, lx, ly, slack,
                          slackx, N);
            }
          }
        }
      }
    }

    assert(path_found);
    max_match++;

    // augment along the path
    for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty) {
      ty = matching[cx];
      revmatch[cy] = cx;
      matching[cx] = cy;
    }
  }

  // return the final answer
  int weight = 0;
  for (int x = 0; x < N; x++) {
    weight += G[x][matching[x]];
  }
  return weight;
}

int min_weight_matching(int G[MAX_N][MAX_N], int N, int matching[MAX_N])
{
  int M = INT_MIN;

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (G[i][j] != INT_MAX) {
        M = max(M, G[i][j]);
      }
    }
  }

  int newG[MAX_N][MAX_N];
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      newG[i][j] = (G[i][j] == INT_MAX) ? INT_MIN : M - G[i][j];
    }
  }
  int weight = max_weight_matching(newG, N, matching);
  return N*M - weight;
}

int main(void)
{
  int G[3][3] = { {INT_MAX,4,5}, {5,7,6}, {5,8,8} };
  int matching[3];

  int w = min_weight_matching(G, 3, matching);
  cout << "weight = " << w << endl;
  for (int i = 0; i < 3; i++) {
    cout << i << " is matched to " << matching[i] << endl;
```



```
  }
  return 0;
}
```

```
/*
 * Infix expressions evaluation
 *
 * Author: Howard Cheng
 *
 * The evaluate() routine takes a string containing an infix arithmetic
 * expression, and return the numeric result after evaluation.  The
 * parameter error indicates whether an error has occurred (syntax
 * error, illegal operation, etc.).  If there is an error the result
 * returned is meaningless.
 *
 * The routine assumes that the operands in the input are integers
 * with no leading signs.  It supports the standard +, -, *, / and
 * parentheses.  If you need to support more operators, operand types,
 * etc., you will need to modify the code.  See comments below.
 *
 */

#include <iostream>
#include <string>
#include <stack>
#include <cctype>
#include <cstdlib>

using namespace std;

// What is a token?  Modify if needed (e.g. to support variables, extra
// operators, etc.)
struct Token
{
  enum Type {NUMBER, PLUS, MINUS, TIMES, DIVIDE, LEFT_PAREN, RIGHT_PAREN};

  // priority of the operators: bigger number means higher priority
  // e.g. */ has priority 2, +- has priority 1, ( has priority 0
  int priority[7];

  // is the operator left associative?  It's assumed that all operators
  // of the same priority level has the same left/right associative property
  bool left_assoc[7];

  Type type;
  long val;

  Token()
  {
    priority[1] = priority[2] = 1;
    priority[3] = priority[4] = 2;
    priority[5] = 0;
    left_assoc[1] = left_assoc[2] = left_assoc[3] = left_assoc[4] = true;
  }

  int get_priority() {
    return priority[type];
  }

  bool is_left_assoc() {
    return left_assoc[type];
  }

  // returns true if there is a next token
  bool next_token(string &expr, int &start, bool &error)
  {
    int len = expr.length();

    error = false;
    while (start < len && isspace(expr[start])) {
      start++;
    }
    if (start >= len) {
```



```
      return false;
    }

    switch (expr[start]) {
    case '(':
      type = LEFT_PAREN;
      break;
    case ')':
      type = RIGHT_PAREN;
      break;
    case '*':
      type = TIMES;
      break;
    case '/':
      type = DIVIDE;
      break;
    case '+':
      type = PLUS;
      break;
    case '-':
      type = MINUS;
      break;
    default:
      // check for number
      const char *s = expr.c_str() + start;
      char *p;
      val = strtol(s, &p, 10);
      if (s == p) {
        error = true;
        return false;
      }
      type = NUMBER;
      start += (p - s);
    }
    if (type != NUMBER) {
      start++;
    }
    return true;
  }
};

// Modify this if you need to support more operators or change their
// meanings.
//
// returns true if operation is successful
bool apply_op(stack<long> &operands, Token token)
{
  long a, b;

  if (operands.size() < 2) {
    return false;
  }
  if (token.type == Token::PLUS) {
    b = operands.top(); operands.pop();
    a = operands.top(); operands.pop();
    operands.push(a+b);
  } else if (token.type == Token::MINUS) {
    b = operands.top(); operands.pop();
    a = operands.top(); operands.pop();
    operands.push(a-b);
  } else if (token.type == Token::TIMES) {
    b = operands.top(); operands.pop();
    a = operands.top(); operands.pop();
    operands.push(a*b);
  } else if (token.type == Token::DIVIDE) {
    b = operands.top(); operands.pop();
    a = operands.top(); operands.pop();
    if (b == 0) {
      return false;
```

```
    }
    operands.push(a/b);
  } else {
    return false;
  }
  return true;
}

long evaluate(string expr, bool &error)
{
  stack<Token> s;
  stack<long> operands;
  int i;
  Token token;

  error = false;
  i = 0;
  while (token.next_token(expr, i, error) && !error) {
    switch (token.type) {
    case Token::NUMBER:
      operands.push(token.val);
      break;
    case Token::LEFT_PAREN:
      s.push(token);
      break;
    case Token::RIGHT_PAREN:
      while (!(error = s.empty()) && s.top().type != Token::LEFT_PAREN) {
        if ((error = !apply_op(operands, s.top()))) {
          break;
        }
        s.pop();
      }
      if (!error) {
        s.pop();
      }
      break;
    default:      // arithmetic operators
      while (!error && !s.empty() &&
             (token.get_priority() < s.top().get_priority() ||
              token.get_priority() == s.top().get_priority() &&
              token.is_left_assoc())) {
        error = !apply_op(operands, s.top());
        s.pop();
      }
      if (!error) {
        s.push(token);
      }
    }
    if (error) {
      break;
    }
  }
  while (!error && !s.empty()) {
    error = !apply_op(operands, s.top());
    s.pop();
  }
  error |= (operands.size() != 1);
  if (error) {
    return 0;
  }
  return operands.top();
}

int main(void)
{
  int result;
  string expr;
  bool error;
```



```
  getline(cin, expr);
  while (!cin.eof()) {
    result = evaluate(expr, error);
    if (error) {
      cout << "Invalid expression" << endl;
    } else {
      cout << " = " << result << endl;
    }
    getline(cin, expr);
  }
  return 0;

}
```

```
// Determines the point(s) of intersection if a circle and a circle
//
// Author: Darcy Best
// Date  : October 1, 2010
// Source: http://local.wasp.uwa.edu.au/~pbourke/geometry/2circle/
//
// Note: A circle of radius 0 must be considered independently.
// See comments in the implementation.

#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
using namespace std;

#define SQR(X) ((X) * (X))

// How close to call equal
const double EPS = 1e-4;

bool dEqual(double x,double y){
  return fabs(x-y) < EPS;
}

struct Point{
  double x,y;
  bool operator<(const Point& a) const{
    if(dEqual(x,a.x))
      return y < a.y;
    return x < a.x;
  }
};

// Prints out the ordered pair. This also accounts for the negative 0.
void print(const Point& a){
  cout << "(";
  if(fabs(a.x) < 1e-4)
    cout << "0.000";
  else
    cout << a.x;
  cout << ",";
  if(fabs(a.y) < 1e-4)
    cout << "0.000";
  else
    cout << a.y;
  cout << ")";
}

struct Circle{
  double r,x,y;
};

// Input:
//  Two circles to intersect
//
// Output:
//  Number of points of intersection points
//  If 1 (or 2), then ans1 (and ans2) contain those points.
//  If 3, then there are infinitely many. (They're the same circle)
int intersect_circle_circle(Circle c1,Circle c2,Point& ans1,Point& ans2){

  // If we have two singular points
  if(fabs(c1.r) < EPS && fabs(c2.r) < EPS){
    if(dEqual(c1.x,c2.x) && dEqual(c1.y,c2.y)){
      ans1.x = c1.x;
      ans1.y = c1.y;
      // Here, you need to know what the intersection of two exact points is:
      //  "return 1;" - If the points intersect at only 1 point
      //  "return 3;" - If the circles are the same
```



```
      // Note that both are true -- It all depends on the problem
      return 1;
    } else {
      return 0;
    }
  }

  double d = hypot(c1.x-c2.x,c1.y-c2.y);

  // Check if the circles are exactly the same.
  if(dEqual(c1.x,c2.x) && dEqual(c1.y,c2.y) && dEqual(c1.r,c2.r))
    return 3;

  // The circles are disjoint
  if(d > c1.r + c2.r + EPS)
    return 0;

  // One circle is contained inside the other -- No intersection
  if(d < abs(c1.r-c2.r) - EPS)
    return 0;

  double a = (SQR(c1.r) - SQR(c2.r) + SQR(d)) / (2*d);
  double h = sqrt(abs(SQR(c1.r) - SQR(a)));

  Point P;
  P.x = c1.x + a / d * (c2.x - c1.x);
  P.y = c1.y + a / d * (c2.y - c1.y);

  ans1.x = P.x + h / d * (c2.y - c1.y);
  ans1.y = P.y - h / d * (c2.x - c1.x);

  if(fabs(h) < EPS)
    return 1;

  ans2.x = P.x - h / d * (c2.y - c1.y);
  ans2.y = P.y + h / d * (c2.x - c1.x);

  return 2;
}

int main(){
  cout << fixed << setprecision(3);
  Circle C1,C2;
  Point a1,a2;

  while(cin >> C1.x >> C1.y >> C1.r >> C2.x >> C2.y >> C2.r){
    int num = intersect_circle_circle(C1,C2,a1,a2);
    switch(num){
    case 0:
      cout << "NO INTERSECTION" << endl;
      break;
    case 1:
      print(a1); cout << endl;
      break;
    case 2:
      if(a2 < a1)
        swap(a1,a2);
      print(a1);print(a2);cout << endl;
      break;
    case 3:
      cout << "THE CIRCLES ARE THE SAME" << endl;
      break;
    }
  }
  return 0;
}
```

```
/*
 * 2-D Line Intersection
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/cga1.html
 *
 * This routine takes two infinite lines specified by two points, and
 * determines whether they intersect at one point, infinitely points,
 * or no points.  In the first case, the point of intersection is also
 * returned.  The points of a line must be different (otherwise,
 * the line is not defined).
 *
 */

#include <iostream>
#include <cmath>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

/* returns 1 if intersect at a point, 0 if not, -1 if the lines coincide */
int intersect_iline(Point a, Point b, Point c, Point d, Point &p)
{
  double r;
  double denom, num1, num2;

  assert((a.x != b.x || a.y != b.y) && (c.x != d.x || c.y != d.y));

  num1 = (a.y - c.y)*(d.x - c.x) - (a.x - c.x)*(d.y - c.y);
  num2 = (a.y - c.y)*(b.x - a.x) - (a.x - c.x)*(b.y - a.y);
  denom = (b.x - a.x)*(d.y - c.y) - (b.y - a.y)*(d.x - c.x);

  if (fabs(denom) >= EPSILON) {
    r = num1 / denom;
    p.x = a.x + r*(b.x - a.x);
    p.y = a.y + r*(b.y - a.y);
    return 1;
  } else {
    if (fabs(num1) >= EPSILON) {
      return 0;
    } else {
      return -1;
    }
  }
}

int main(void)
{
  Point a, b, c, d, p;
  int res;

  while (cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y >> d.x >> d.y) {
    res = intersect_iline(a, b, c, d, p);
    if (res == 1) {
      cout << "Intersect at (" << p.x << "," << p.y << ")" << endl;
    } else if (res == 0) {
      cout << "Don't intersect" << endl;
    } else {
      cout << "Infinite number of intersections" << endl;
    }
  }
```



```
  return 0;
}
```

```
// Determines the point(s) of intersection if a circle and a line
//
// Author: Darcy Best
// Date  : May 1, 2010
// Source: http://mathworld.wolfram.com/Circle-LineIntersection.html

#include <iostream>
#include <cmath>
using namespace std;

#define SQR(X) ((X) * (X))

// How close to call equal
const double EPS = 1e-7;

bool dEqual(double x,double y){
  return fabs(x-y) < EPS;
}

struct Point{
  double x,y;
};

struct Line{
  Point p1,p2;
};

struct Circle{
  Point centre;
  double radius;
};

// Input of:
//  - 2 distinct points on the line
//  - The centre of the circle
//  - The radius of the circle
// Output:
//  Number of points of intersection points
//  If 1 or 2, then ans1 and ans2 contain those points.
int intersect_iline_circle(Line l,Circle c,Point& ans1,Point& ans2){
  Point p1 = l.p1;
  Point p2 = l.p2;

  Point circCentre = c.centre;
  double rad = c.radius;

  p1.x -= circCentre.x;
  p2.x -= circCentre.x;
  p1.y -= circCentre.y;
  p2.y -= circCentre.y;

  double dx = p2.x - p1.x;
  double dy = p2.y - p1.y;
  double dr = SQR(dx) + SQR(dy);
  double D  = p1.x*p2.y - p2.x*p1.y;

  double desc = SQR(rad)*dr - SQR(D);

  if(dEqual(desc,0)){
    ans1.x = circCentre.x + (D*dy) / dr;
    ans1.y = circCentre.y + (-D*dx) / dr;
    return 1;
  } else if(desc < 0){
    return 0;
  }

  double sgn = (dy < -EPS ? -1 : 1);

  ans1.x = circCentre.x + (D*dy + sgn*dx*sqrt(desc)) / dr;
```



```
  ans1.y = circCentre.y + (-D*dx + abs(dy)*sqrt(desc)) / dr;

  ans2.x = circCentre.x + (D*dy - sgn*dx*sqrt(desc)) / dr;
  ans2.y = circCentre.y + (-D*dx - abs(dy)*sqrt(desc)) / dr;

  return 2;
}

int main(){
  Line L;
  Circle C;
  Point a1,a2;

  cin >> L.p1.x >> L.p1.y >> L.p2.x >> L.p2.y;
  cin >> C.centre.x >> C.centre.y >> C.radius;

  int num = intersect_iline_circle(L,C,a1,a2);
  if(num == 0)
    cout << "NO INTERSECTION." << endl;
  else if(num == 1)
    cout << "ONE INTERSECTION: (" << a1.x << "," << a1.y << ")" << endl;
  else if(num == 2)
    cout << "TWO INTERSECTIONS: (" << a1.x << "," << a1.y << ")"
         << "(" << a2.x << "," << a2.y << ")" << endl;

  return 0;
}
```

```cpp
/*
 * 2-D Line Intersection
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/cga1.html
 *
 * This routine takes two line segments specified by endpoints, and
 * determines whether they intersect at one point, infinitely points,
 * or no points.  In the first case, the point of intersection is also
 * returned.  The endpoints of a line must be different (otherwise,
 * the line is not defined).
 *
 */

#include <iostream>
#include <cmath>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

/* returns 1 if intersect at a point, 0 if not, -1 if the lines coincide */
int intersect_line(Point a, Point b, Point c, Point d, Point &p)
{
  Point t;
  double r, s;
  double denom, num1, num2;

  assert((a.x != b.x || a.y != b.y) && (c.x != d.x || c.y != d.y));

  num1 = (a.y - c.y)*(d.x - c.x) - (a.x - c.x)*(d.y - c.y);
  num2 = (a.y - c.y)*(b.x - a.x) - (a.x - c.x)*(b.y - a.y);
  denom = (b.x - a.x)*(d.y - c.y) - (b.y - a.y)*(d.x - c.x);

  if (fabs(denom) >= EPSILON) {
    r = num1 / denom;
    s = num2 / denom;
    if (0-EPSILON <= r && r <= 1+EPSILON &&
        0-EPSILON <= s && s <= 1+EPSILON) {
      /* always do this part if we are interested in lines instead */
      /* of line segments                                          */
      p.x = a.x + r*(b.x - a.x);
      p.y = a.y + r*(b.y - a.y);
      return 1;
    } else {
      return 0;
    }
  } else {
    if (fabs(num1) >= EPSILON) {
      return 0;
    } else {
      /* I am not using "fuzzy comparisons" here, because the comparisons */
      /* are based on the input, not some derived quantities.  You may    */
      /* want to change that if the input points are computed somehow.    */

      /* two lines are the "same".  See if they overlap */
      if (a.x > b.x || (a.x == b.x && a.y > b.y)) {
        t = a;
        a = b;
        b = t;
      }
      if (c.x > d.x || (c.x == d.x && c.y > d.y)) {
```



```cpp
        t = c;
        c = d;
        d = t;
      }
      if (a.x == b.x) {
        /* vertical lines */
        if (b.y == c.y) {
          p = b;
          return 1;
        } else if (a.y == d.y) {
          p = a;
          return 1;
        } else if (b.y < c.y || d.y < a.y) {
          return 0;
        } else {
          return -1;
        }
      } else {
        if (b.x == c.x) {
          p = b;
          return 1;
        } else if (a.x == d.x) {
          p = a;
          return 1;
        } else if (b.x < c.x || d.x < a.x) {
          return 0;
        } else {
          return -1;
        }
      }

      return -1;
    }
  }
}

int main(void)
{
  Point a, b, c, d, p;
  int res;

  while (cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y >> d.x >> d.y) {
    res = intersect_line(a, b, c, d, p);
    if (res == 1) {
      cout << "Intersect at (" << p.x << "," << p.y << ")" << endl;
    } else if (res == 0) {
      cout << "Don't intersect" << endl;
    } else {
      cout << "Infinite number of intersections" << endl;
    }
  }

  return 0;
}
```

```
/*
 * Line Intersection
 *
 * Author: Howard Cheng
 * Reference:
 *   CLRS, "Introduction to Algorithms", 2nd edition, pages 936-939.
 *
 * Given two lines specified by their endpoints (a1, a2) and (b1, b2),
 * returns true if they intersect, and false otherwise.  The intersection
 * point is not known.
 *
 */

#include <iostream>
#include <cmath>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

double direction(Point p1, Point p2, Point p3)
{
  double x1 = p3.x - p1.x;
  double y1 = p3.y - p1.y;
  double x2 = p2.x - p1.x;
  double y2 = p2.y - p1.y;
  return x1*y2 - x2*y1;
}

int on_segment(Point p1, Point p2, Point p3)
{
  return ((p1.x <= p3.x && p3.x <= p2.x) || (p2.x <= p3.x && p3.x <= p1.x)) &&
    ((p1.y <= p3.y && p3.y <= p2.y) || (p2.y <= p3.y && p3.y <= p1.y));
}

int intersect(Point a1, Point a2, Point b1, Point b2)
{
  double d1 = direction(b1, b2, a1);
  double d2 = direction(b1, b2, a2);
  double d3 = direction(a1, a2, b1);
  double d4 = direction(a1, a2, b2);

  if (((d1 > EPSILON && d2 < -EPSILON) || (d1 < -EPSILON && d2 > EPSILON)) &&
      ((d3 > EPSILON && d4 < -EPSILON) || (d3 < -EPSILON && d4 > EPSILON))) {
    return 1;
  } else {
    return (fabs(d1) < EPSILON && on_segment(b1, b2, a1)) ||
      (fabs(d2) < EPSILON && on_segment(b1, b2, a2)) ||
      (fabs(d3) < EPSILON && on_segment(a1, a2, b1)) ||
      (fabs(d4) < EPSILON && on_segment(a1, a2, b2));
  }
}

int main(void)
{
  Point a, b, c, d;
  int a1, a2, a3, a4, a5, a6, a7, a8;

  while (cin >> a1 >> a2 >> a3 >> a4 >> a5  >> a6 >> a7 >> a8) {
    a.x = a1; a.y = a2;
    b.x = a3; b.y = a4;
    c.x = a5; c.y = a6;
    d.x = a7; d.y = a8;
    if (intersect(a, b, c, d)) {
```



```
      cout << "Yes" << endl;
    } else {
      cout << "No" << endl;
    }
  }
  return 0;
}
```

```
/*
 * Integer multiplication/division without overflow
 *
 * Author: Howard Cheng
 *
 * Given a list of factors in the numerator (num, size n) and a list
 * of factors in the denominator (dem, size m), it returns the product
 * of the numerator divided by the denominator.  It is assumed that
 * the numerator is divisible by the denominator (ie. the result
 * is an integer).  Overflow will not occur as long as the final result
 * is representable.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int gcd(int a, int b)
{
   int r;

   while (b) {
      r = a % b;
      a = b;
      b = r;
   }
   assert(a >= 0);
   return a;
}

int mult(int A[], int n, int B[], int m)
{
   int i, j, prod, d;
   int count = 0;

   /* unnecessary if the two lists are positive */
   for (i = 0; i < n; i++) {
     if (A[i] < 0) {
       A[i] *= -1;
       count++;
     }
   }
   for (i = 0; i < m; i++) {
     if (B[i] < 0) {
       B[i] *= -1;
       count++;
     }
   }

   for (i = 0; i < n; i++) {
      for (j = 0; j < m; j++) {
         d = gcd(A[i], B[j]);
         A[i] /= d;
         B[j] /= d;
      }
   }
   prod = 1;
   for (i = 0; i < n; i++) {
      prod *= A[i];
   }
   for (j = 0; j < m; j++) {
      assert(B[j] == 1);
   }
   return (count % 2 == 0) ? prod : -prod;
}

int main(void)
```



```
{
  int A[1000], B[1000], n, m, i;

  while (cin >> n >> m && n > 0 && m > 0) {
    for (i = 0; i < n; i++) {
      cin >> A[i];
    }
    for (i = 0; i < m; i++) {
      cin >> B[i];
    }
    cout << "prod = " << mult(A,n,B,m) << endl;
  }

  return 0;
}
```

```c
/*
 * All-integer programming
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.cs.sunysb.edu/~algorith/implement/syslo/distrib/processed/
 *
 * This algorithm is based on GOMORY cutting plane method.
 *
 * This algorithm solves the following INTEGER LP problem:
 *
 * minimize       SUM  (A[0][j] * x[j])         [cost function]
 *            (j=0 to n-1)
 *
 *     s.t.       SUM   (A[i][j]*x[j])  <=  A[i][n]    1 <= i <= m
 *            (j=0 to n-1)
 *
 *     and        x[j] >= 0         0 <= j <= n-1
 *
 * n = number of variables
 * m = number of constraints
 *
 * Input : An input array A with m+n+1 rows and n+1 columns.
 *         Store the cost function in row 0, and the constraints in rows
 *           1 to m.  Set A[0][n] = 0.
 *         A vector x allocated for n values to store returned value.
 *
 * Output: Returns 1 if a solution is found, 0 if no solution exists.
 *         The minimum value of the cost function is returned in
 *           value.
 *         The variable assignment to x[j] that gives the minimum is given
 *           in x[j], where 0 <= j <= n-1.
 *
 * Important Notes:
 *
 * 1. If we want to have constraints that are >=, just multiply all the
 *    coefficients by -1.
 * 2. If we want to have constraints that are ==, do both >= and <=.
 * 3. The contents of A is destroyed after this routine.
 * 4. The coefficients in the cost function must be positive.  If not,
 *    make a change of variable x'[i] = m[i]-x[i] where m[i] is the
 *    maximum value for variable[i] and adjust all constraints as well
 *    as the returned optimal value.  This is especially useful if you
 *    wish to maximize the cost function.
 *
 *    Usually there is some maximum for each variable if you wish to
 *    maximize the function (or the value could be infinity.
 *
 *    NOTE: if any coefficient in the objective function is negative or
 *          0, the routine will crash.
 *
 * 5. If one only wishes to know if there is any variable assignment
 *    satisfying the constraints, just put 1 in each coefficient
 *    of the objective function.
 */

#include <stdio.h>
#include <assert.h>


#define MAX_VARS  50
#define MAX_CONS  50
#define MAX_ROWS  MAX_VARS+MAX_CONS+1
#define MAX_COLS  MAX_VARS+1

int euclid(int u, int v)
{
  int w = u / v;
  if (w*v > u) {
```



```c
    w--;
  }
  if ((w+1)*v <= u) {
    w++;
  }
  return w;
}

int int_prog(int A[MAX_ROWS][MAX_COLS], int n, int m, int *value, int *x)
{
  int iter, nosol;
  int b, c, i, j, k, l, r, r1, s, t, denom, num;

  for (j = 0; j < n; j++) {
    if (A[0][j] <= 0) {
      // BAD objective function coefficient: make sure it is positive
      assert(false);
    }
  }

  /* set constraints that x[j] >= 0, and clear output */
  for (i = 0; i < n; i++) {
    for (j = 0; j < n+1; j++) {
      A[m+1+i][j] = 0;
    }
    A[m+1+i][i] = -1;
  }
  A[0][n] = 0;

  nosol = 0;
  do {
    r = 0;
    do {
      iter = (A[++r][n] < 0);
    } while (!iter && r != n+m);
    if (iter) {
      for (k = iter = 0; k < n && !iter; k++) {
        iter = (A[r][k] < 0);
      }
      nosol = !iter;
      if (iter) {
        l = k-1;
        for (j = k; j < n; j++) {
          if (A[r][j] < 0) {
            for (i = 0; !(s = A[i][j] - A[i][l]); i++)
              ;
            if (s < 0) {
              l = j;
            }
          }
        }
        for (s = 0; !A[s][l]; s++)
          ;
        num = -A[r][l];
        denom = 1;
        for (j = 0; j < n; j++) {
          if (A[r][j] < 0 && j != l) {
            for (i = s-1, b = 1; b && i >= 0; i--) {
              b = (A[i][j] == 0);
            }
            if (b) {
              i = A[s][j];
              r1 = A[s][l];
              t = euclid(i, r1);
              if (t*r1 == i && t > 1) {
                for (i = s+1; !(r1 = t*A[i][l] - A[i][j]); i++)
                  ;
                if (r1 > 0) {
                  t--;
```

```
              }
            }
            c = -A[r][j];
            if (c*denom > t*num) {
              num = c;
              denom = t;
            }
          }
        }
      }
      for (j = 0; j <= n; j++) {
        if (j != l) {
          c = euclid(A[r][j]*denom, num);
          if (c) {
            for (i = 0; i <= n+m; i++) {
              A[i][j] += c*A[i][l];
            }
          }
        }
      }
    }
  }
} while (iter && !nosol);

*value = -A[0][n];
for (j = 0; j < n; j++) {
  x[j] = A[m+1+j][n];
}

return !nosol;
}

int main(void)
{
  int A[MAX_ROWS][MAX_COLS];
  int x[MAX_VARS];
  int val, t;
  int m, n, i, j;

  while (scanf("%d %d", &n, &m) == 2 && n > 0 && m > 0) {
    /* read cost function */
    printf("Input cost function to minimize:\n");
    for (i = 0; i < n; i++) {
      scanf("%d", &A[0][i]);
    }

    /* read constraints */
    for (i = 1; i <= m; i++) {
      printf("Input constraint #%d:\n", i);
      for (j = 0; j < n+1; j++) {
        scanf("%d", &A[i][j]);
      }
    }

    t = int_prog(A, n, m, &val, x);
    if (t) {
      printf("Minimum cost = %d\n", val);
      for (i = 0; i < n; i++) {
        printf("x[%2d] = %2d\n", i, x[i]);
      }
    } else {
      printf("No solution exists.\n");
    }
  }

  return 0;
}
```

```
//
// Josephus Problem
//
// Author: Darcy Best
// Date  : September 4, 2010
//
// The Josephus problem:
//   A group of n people are in a circle, and you start by killing
//     person f. Then, you kill every kth person until only one person
//     is left.
//
// Two implementations are given here (Note that neither depend on k):
//   1. Determine the survivor           -- O(n)
//   2. Determine the full killing order -- O(n^2)
//
// If there are 17 people, with every 5th person killed (killing the
//    1st person first), the kill order is:
//     1,6,11,16,5,12,2,9,17,10,4,15,14,3,8,13,7 (survivor = 7)
//
// NOTE: This is 1-based, not 0-based.

#include <iostream>
using namespace std;

const int MAX_N = 100;

int survivor(int n,int f,int k){
  return (n==1 ? 1 : (survivor(n-1,k,k) + (f-1)) % n + 1);
}

void killOrder(int n,int f,int k,int A[]){
  if(n == 0) return;
  A[0] = 0;
  killOrder(n-1,k,k,A+1);
  for(int i=0;i<n;i++)
    A[i] = (A[i] + (f-1)) % n + 1;
}

int main(){
  int n,f,k,kOrder[MAX_N];
  while(cin >> n >> f >> k && (n || f || k)){
    killOrder(n,f,k,kOrder);
    for(int i=0;i<n;i++)
      cout << kOrder[i] << endl;

    cout << "Survivor: " << survivor(n,f,k) << endl;
  }
  return 0;
}
```

```
/*
 * KMP String Matching
 *
 * Author: Howard Cheng
 *
 * The prepare_pattern routine takes in the pattern you wish to search
 * for, and perform some processing to give a "failure array" to be used
 * by the actual search.  The complexity is linear in the length of the
 * pattern.
 *
 * The find_pattern routine takes in a string s, a pattern pat, and a
 * vector T computed by prepare_pattern.  It returns the index of the
 * first occurrence of pat in s, or -1 if it does not occur in s.
 * The complexity is linear in the length of the string s.
 *
 */

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

void prepare_pattern(const string &pat, vector<int> &T)
{
  int n = pat.length();
  T.resize(n+1);
  fill(T.begin(), T.end(), -1);
  for (int i = 1; i <= n; i++) {
    int pos = T[i-1];
    while (pos != -1 && pat[pos] != pat[i-1]) {
      pos = T[pos];
    }
    T[i] = pos + 1;
  }
}

int find_pattern(const string &s, const string &pat, const vector<int> &T)
{
  int sp = 0, kp = 0;
  int slen = s.length(), plen = pat.length();
  while (sp < slen) {
    while (kp != -1 && (kp == plen || pat[kp] != s[sp])) {
      kp = T[kp];
    }
    kp++;   sp++;
    if (kp == plen) {
      // a match is found
      return sp - plen;

      // if you want more than one match (i.e. all matches), do not return
      // in the above but rather record the location of the match.  Continue
      // the loop with:
      //
      // kp = T[kp];
    }
  }
  return -1;
}

int main(void)
{
  string str, pat;

  while (cin >> str >> pat) {
    vector<int> T;
    prepare_pattern(pat, T);
    cout << "index = " << find_pattern(str, pat, T) << endl;
```



```
  }
  return 0;
}
```

```
/*
 * Solution of systems of linear equations
 *
 * Author: Howard Cheng
 * Reference:
 *   K.E. Atkinson. "An Introduction to Numerical Analysis."  2nd Ed., John
 *   Wiley & Sons, 1988, pages 520-521.  ISBN 0-471-62489-6
 *
 * To solve the system Ax = b where A is an n x n matrix, first call
 * LU_decomp on A to obtain its LU decomposition.  Once the LU
 * decomposition is obtained, it can be used to solve linear systems with
 * the same coefficient matrix A but different vectors of b using the
 * LU_solve routine.  This routine is numerically stable (provided that
 * the original coefficient matrix has a small condition number).
 *
 * The inputs to LU_decomp are the matrix A, the dimension n, an
 * output array pivot of n-1 elements such that pivot[i] = j means
 * that rows i and j were swapped during the i-th step, and an output
 * parameter to return the determinant of the matrix.  The function
 * returns 1 if successful, and 0 if the matrix is singular.  The
 * matrix A is overwritten by its LU decomposition on return.  If the
 * matrix is singular, the content of A should not be used (it represents
 * intermediate results during the decomposition).
 *
 * The inputs to LU_solve are the LU decomposition of A, the dimension
 * n, the pivot array from LU_decomp, and n-dimensional vectors b and
 * x.  This function should be called only if the original matrix A
 * has a small condition number.  You can check this by checking that
 * the determinant returned by LU_decomp is not too close to 0.  This is
 * only a crude check: you should really be computing the condition number
 * of the matrix.
 *
 */

#include <iostream>
#include <cmath>

using namespace std;

const int MAX_N = 10;

int LU_decomp(double A[MAX_N][MAX_N], int n, int pivot[], double &det)
{
  double s[MAX_N];            /* factors used in implicit scaling */
  double c, t;
  int i, j, k;

  det = 1.0;

  /* compute s[i] */
  for (i = 0; i < n; i++) {
    s[i] = 0.0;
    for (j = 0; j < n; j++) {
      if ((t = fabs(A[i][j])) > s[i]) {
        s[i] = t;
      }
    }
    if (s[i] == 0.0) {
      /* a row of zeroes: singular */
      det = 0.0;
      return 0;
    }
  }

  /* do the row reductions */
  for (k = 0; k < n-1; k++) {
    c = fabs(A[k][k]/s[k]);
    pivot[k] = k;
    for (i = k+1; i < n; i++) {
```



```
      t = fabs(A[i][k]/s[i]);
      if (t > c) {
        c = t;
        pivot[k] = i;
      }
    }

    if (c == 0) {
      /* pivot == 0: singular */
      det = 0.0;
      return 0;
    }

    /* do row exchange */
    if (k != pivot[k]) {
      det *= -1.0;
      for (j = k; j < n; j++) {
        t = A[k][j];
        A[k][j] = A[pivot[k]][j];
        A[pivot[k]][j] = t;
        t = s[k];
        s[k] = s[pivot[k]];
        s[pivot[k]] = t;
      }
    }

    /* do the row reduction */
    for (i = k+1; i < n; i++) {
      A[i][k] /= A[k][k];
      for (j = k+1; j < n; j++) {
        A[i][j] -= A[i][k] * A[k][j];
      }
    }

    det *= A[k][k];
  }

  /* note that the algorithm as state in the book is incorrect.  The */
  /* following is need to ensure that the last row is not all 0's.   */
  /* (maybe the book is correct, depending on what you think it's    */
  /* supposed to do.)                                                */
  if (A[n-1][n-1] == 0.0) {
    det = 0.0;
    return 0;
  } else {
    det *= A[n-1][n-1];
    return 1;
  }
}

void LU_solve(double A[MAX_N][MAX_N], int n, int pivot[], double b[],
              double x[])
{
  double t;
  int i, j, k;

  for (i = 0; i < n; i++) {
    x[i] = b[i];
  }
  for (k = 0; k < n-1; k++) {
    /* swap if necessary */
    if (k != pivot[k]) {
      t = x[k];
      x[k] = x[pivot[k]];
      x[pivot[k]] = t;
    }

    for (i = k+1; i < n; i++) {
      x[i] -= A[i][k] * x[k];
```

```
    }
  }

  x[n-1] /= A[n-1][n-1];

  for (i = n-2; i >= 0; i--) {
    for (j = i+1; j< n; j++) {
      x[i] -= A[i][j] * x[j];
    }
    x[i] /= A[i][i];
  }
}

int main(void)
{
  double A[MAX_N][MAX_N], x[MAX_N], b[MAX_N];
  int pivot[MAX_N];              /* only n-1 is needed, but what the heck */
  int n, i, j;
  double det;

  while (cin >> n && 0 < n && n <= MAX_N) {
    cout << "Enter A:" << endl;
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        cin >> A[i][j];
      }
    }
    cout << "Enter b:";
    for (i = 0; i < n; i++) {
      cin >> b[i];
    }
    if (LU_decomp(A, n, pivot, det)) {
      LU_solve(A, n, pivot, b, x);
      cout << "LU decomposition of A:" << endl;
      for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
          cout << A[i][j] << " ";
        }
        cout << endl;
      }
      cout << "det = " << det << endl;
      cout << "x = ";
      for (i = 0; i < n; i++) {
        cout << x[i] << " ";
      }
      cout << endl;
    } else {
      cout << "A is singular" << endl;
    }
  }
  return 0;
}
```

```c
/* unweighted matching in a bipartite graph.
 * author: Matthew McNaughton, Jan 16, 1999.
 * mcnaught@cs.ualberta.ca
 *
 * The bipartite graph G is split into two sets, U and V,
 * of user-defined maximum size MAXU and MAXV.
 * the input graph is in bipgraph[MAXU][MAXV].
 * there is an edge between node u \in U and node v \in V
 * iff bipgraph[u][v] != 0.
 *
 * The output is in matching[MAXU].
 * node u \in U and node v \in V are matched iff matching[u] == v.
 *
 * parameters match(int u, int v) mean: u is the number of vertices
 * in U, v the  number in V. They are assumed to be numbered 0 .. u-1
 * and 0 .. v-1, respectively.
 *
 * Technique: given a non-maximum matching M on G, find an "alternating path"
 * u_1 v_1 ... u_n v_n so that u_1 and v_n are not matched in M, but
 * v_k u_k+1 are matched with each other. Then "flip" the edges so
 * that edges on this path which were not in the matching are, and edges
 * which were are not. This increases the size of the matching by one.
 * It is a fact that if no such path exists, then M is maximum.
 *
 * This algorithm finds several alternating paths at once by performing
 * bfs starting at all unmatched nodes u \in U. Paths which do not
 * have intersecting nodes can be alternated in the same bfs run.
 * bfs is performed repeated until the matching cannot be expanded.
 */

#include <stdio.h>
#include <string.h>
#include <assert.h>

FILE *in, *out;

/* change these as necessary */
#define MAXU 100
#define MAXV 100

#define U(i) (i)
#define V(i) ((i) + MAXU)
#define isU(i) ((i) < MAXU)
#define isV(i) ((i) >= MAXU)

#define isMatched(i) (isU(i) ? flagUmatched[(i)] : flagVmatched[(i)-MAXU])
#define isUsed(i) (isU(i) ? flagUused[(i)] : flagVused[(i)-MAXU])
#define isVisited(i) (isU(i) ? flagUvisited[(i)] : flagVvisited[(i)-MAXU])

#define setMatched(i) (isU(i)?(flagUmatched[(i)]=1):(flagVmatched[(i)-MAXU]=1))
#define setUsed(i) (isU(i)?(flagUused[(i)]=1):(flagVused[(i)-MAXU]=1))
#define setVisited(i) (isU(i)?(flagUvisited[(i)]=1):(flagVvisited[(i)-MAXU]=1))

char bipgraph[MAXU][MAXV];
int matching[MAXU]; /* matching[u] == v, _not_ plus MAXU */
char flagUmatched[MAXU], flagVmatched[MAXV];
char flagUvisited[MAXU], flagVvisited[MAXV];
char flagUused[MAXU],    flagVused[MAXV];
int predecessor[MAXU+MAXV], queue[MAXU+MAXV];

/* u and v are the number of vertices in sets U, and V, respectively,
 * filling up bipgraph[0..u-1][0..v-1].
 * result:
 *  matching[u0]==v0 iff u0 and v0 are in the matching,
 * otherwise matching[u0] = -1 */
void
match(int u, int v) {
  int i,j, head,tail, bad, last, increased;
```



```c
  for( i = 0; i < u; i++ ) {
    matching[i] = -1;
    flagUmatched[i] = 0;
  }
  for( i = 0; i < v; i++ ) flagVmatched[i] = 0;

  do { /* find alternating paths by repeated bfs. */
    for( i = 0; i < MAXU+MAXV; i++ ) predecessor[i] = -1;
    for( i = 0; i < MAXU; i++ ) flagUused[i] = flagUvisited[i] = 0;
    for( i = 0; i < MAXV; i++ ) flagVused[i] = flagVvisited[i] = 0;

    head = tail = 0;

    /* put all the unmatched u's on the queue. They start the
     * alternating path. */
    for( i = 0; i < u; i++ ) {
      if( ! isMatched(U(i))) {
        queue[tail++] = U(i);
        predecessor[i] = -1; /* redundant statement */
        setVisited(U(i));
      }
    }

    /* flag that at least one path was found by the bfs.
     * when the bfs does not find an alternating path we are done. */
    increased = 0;

    while( head != tail ) {
      i = queue[head++];

      /* this node appeared on some previously found alternating path. */
      if( isUsed(i) ) continue;

      if( isV(i) && !isMatched(i) ) {
        /* we got to the end of an alternating path. see if
         * it is disjoint with other paths found so far. only
         * then can we mess it up a bit. */
        bad = 0;
        for( j = i; j != -1; j = predecessor[j]) {
          if( isUsed(j)) {
            bad = 1;
            break;
          }
        }

        if( ! bad ) {
          /* this path is pristine. switch "polarity" of edges
           * in the matching on this path. */

          /* flag and instrumention - whether (not) to quit,
           * and how many paths we found this bfs. */
          increased++;
          for( j = i; j != -1; last = j, j = predecessor[j] ) {
            if( isV(j) && !isMatched(j)) {
              /* the only unmatched v - actually this means we
               * are on the first iteration of this loop. */
              setMatched(j);

            } else if( isU(j) ) {
              if( isMatched(j) ) {
                /* the node we saw in the previous iteration of
                 * this loop must be a V. We will match with it
                 * instead of the one we used to match with, which
                 * must be the next node visited in this loop. */
                assert(isV(last));
                matching[j] = last - MAXU;
              } else {
                /* we are the very first u, one of the ones the
                 * bfs queue was "seeded" with. We should have ...*/
```

```
                assert(predecessor[j] == -1);
                setMatched(j);
                assert(isV(last));
                matching[j] = last - MAXU;
              }
            }
            setUsed(j); /* this node cannot be used for other
                         * paths we might run across in the future
                         * on this bfs. */
          } /* for */
        } /* if ! bad */
      } /* isV and !isMatched */

      else if( isV(i) ) {
        /* this must be a matched V - find the matching U and put it on
         * the queue if it is not visited or used. */

        bad = 1;

        for( j = 0; j < u; j++ ) {
          if( isMatched(U(j)) && matching[j] == i - MAXU ) {
            /* this is the one. */
            if( ! isVisited(U(j)) && !isUsed(U(j))) {
              setVisited(U(j));
              queue[tail++] = U(j);
              predecessor[U(j)] = i;
            }
            bad = 0;
            break;
          }
        }
        assert(!bad);
      } /* isV */
      else if( isU(i) ) {
        /* we are at U. whether it is unmatched (a "seed"),
         * or matched, we do the same thing - put on the queue
         * all V's which it is connected to in the graph but
         * which it is _not_ paired to in the current matching. */

        for( j = 0; j < v; j++ ) {
          if( bipgraph[i][j] &&
              !isVisited(V(j)) &&
              !isUsed(V(j)) &&
              matching[i] != j ) {
            /* we can put this one on the queue. */
            queue[tail++] = V(j);
            predecessor[V(j)] = i;
            setVisited(V(j));
          }
        }
      } else {
        assert(0); /* should be no other cases. */
      }
      /* this is the end of the bfs. */
    }
  } while( increased );

  return;
}


int
main() {
  int i,j,u,v,setnum;

  in = stdin; out = stdout; setnum = 0;

  while( fscanf(in, "%d %d", &u, &v) == 2 ) {
```



```
    for( i = 0; i < u; i++ ) for( j = 0; j < v; j++ ) bipgraph[i][j] = 0;

    while( fscanf(in, "%d %d", &i, &j) == 2 && i != -1 && j != -1 ) {
      bipgraph[i][j] = 1;
    }

    match(u,v);

    fprintf(out, "Problem #%d:\n", ++setnum);
    for( i = 0; i < u; i ++ ) {
      if( matching[i] != -1 )
        fprintf(out, "match %d to %d\n", i, matching[i]);
    }
  }
  return 0;
}
```

```
/*
 * Min Cost Max Flow for Dense graphs
 *
 * Authors: Frank Chu, Igor Naverniouk
 * http://shygypsy.com/tools/mcmf3.cpp
 *
 * Min cost max flow * (Edmonds-Karp relabelling + Dijkstra)
 *
 * This implementation takes a directed graph where each edge has a
 * capacity ('cap') and a cost per unit of flow ('cost') and returns a
 * maximum flow network of minimal cost ('fcost') from s to t.
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v] is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-existent edges.
 *      - cost (global): a matrix where cost[u][v] is the cost per unit
 *          of flow along the edge u->v. If cap[u][v] == 0, cost[u][v] is
 *          ignored. ALL COSTS MUST BE NON-NEGATIVE!
 *      - n: the number of vertices ([0, n-1] are considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - the total cost through 'fcost'
 *      - fnet contains the flow network. Careful: both fnet[u][v] and
 *          fnet[v][u] could be positive. Take the difference.
 * COMPLEXITY:
 *      - Worst case: O(n^2*flow  <?  n^3*fcost)
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's implementation.
 **/

#include <iostream>
#include <algorithm>
#include <climits>
using namespace std;

// the maximum number of vertices + 1
const int NN = 1024;

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's successor and depth
int par[NN], d[NN];        // par[source] = source;

// Labelling function
int pi[NN];

const int Inf = INT_MAX/2;

// Dijkstra's using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])

bool dijkstra(int n, int s, int t)
{
  for (int i = 0; i < n; i++) {
    d[i] = Inf;
    par[i] = -1;
  }
```



```
  d[s] = 0;
  par[s] = -n - 1;

  while (1) {
    // find u with smallest d[u]
    int u = -1, bestD = Inf;
    for (int i = 0; i < n; i++) {
      if (par[i] < 0 && d[i] < bestD) {
        bestD = d[u = i];
      }
    }
    if (bestD == Inf) break;

    // relax edge (u,i) or (i,u) for all i;
    par[u] = -par[u] - 1;
    for (int i = 0; i < deg[u]; i++) {
      // try undoing edge v->u
      int v = adj[u][i];
      if (par[v] >= 0) continue;
      if (fnet[v][u] && d[v] > Pot(u,v) - cost[v][u]) {
        d[v] = Pot( u, v ) - cost[v][u];
        par[v] = -u-1;
      }

      // try edge u->v
      if (fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v]) {
        d[v] = Pot(u,v) + cost[u][v];
        par[v] = -u - 1;
      }
    }
  }

  for (int i = 0; i < n; i++) {
    if (pi[i] < Inf) {
      pi[i] += d[i];
    }
  }

  return par[t] >= 0;
}

#undef Pot

int mcmf( int n, int s, int t, int &fcost )
{
  // build the adjacency list
  fill(deg, deg+NN, 0);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (cap[i][j] || cap[j][i]) {
        adj[i][deg[i]++] = j;
      }
    }
  }

  for (int i = 0; i < NN; i++) {
    fill(fnet[i], fnet[i]+NN, 0);
  }
  fill(pi, pi+NN, 0);
  int flow = fcost = 0;

  // repeatedly, find a cheapest path from s to t
  while (dijkstra(n, s, t)) {
    // get the bottleneck capacity
    int bot = INT_MAX;
    for (int v = t, u = par[v]; v != s; u = par[v = u]) {
      bot = min(bot, fnet[v][u] ? fnet[v][u] : (cap[u][v] - fnet[u][v]));
    }
```

```
    // update the flow network
    for (int v = t, u = par[v]; v != s; u = par[v = u]) {
      if (fnet[v][u]) {
        fnet[v][u] -= bot;
        fcost -= bot * cost[v][u];
      } else {
        fnet[u][v] += bot;
        fcost += bot * cost[u][v];
      }
    }

    flow += bot;
  }

  return flow;
}

//----------------- EXAMPLE USAGE -----------------
#include <iostream>
using namespace std;

int main()
{
  int numV;
  cin >> numV;
  for (int i = 0; i < NN; i++) {
    fill(cap[i], cap[i]+NN, 0);
  }

  int m, a, b, c, cp;
  int s, t;
  cin >> m;
  cin >> s >> t;

  // fill up cap with existing capacities.
  // if the edge u->v has capacity 6, set cap[u][v] = 6.
  // for each cap[u][v] > 0, set cost[u][v] to  the
  // cost per unit of flow along the edge i->v
  for (int i=0; i<m; i++) {
    cin >> a >> b >> cp >> c;
    cost[a][b] = c; // cost[b][a] = c;
    cap[a][b] = cp; // cap[b][a] = cp;
  }

  int fcost;
  int flow = mcmf( numV, s, t, fcost );
  cout << "flow: " << flow << endl;
  cout << "cost: " << fcost << endl;

  return 0;
}
```

```
/**
 *   ///////////////////////
 *   // MIN COST MAX FLOW //
 *   ///////////////////////
 *
 *   Authors: Frank Chu, Igor Naverniouk
 **/

/*********************
 * Min cost max flow * (Edmonds-Karp relabelling + fast heap Dijkstra)
 *********************
 * Takes a directed graph where each edge has a capacity ('cap') and a
 * cost per unit of flow ('cost') and returns a maximum flow network
 * of minimal cost ('fcost') from s to t. USE mcmf3.cpp FOR DENSE GRAPHS!
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v] is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-existent edges.
 *      - cost (global): a matrix where cost[u][v] is the cost per unit
 *          of flow along the edge u->v. If cap[u][v] == 0, cost[u][v] is
 *          ignored. ALL COSTS MUST BE NON-NEGATIVE!
 *      - n: the number of vertices ([0, n-1] are considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - the total cost through 'fcost'
 *      - fnet contains the flow network. Careful: both fnet[u][v] and
 *          fnet[v][u] could be positive. Take the difference.
 * COMPLEXITY:
 *      - Worst case: O(m*log(m)*flow  <?  n*m*log(m)*fcost)
 * FIELD TESTING:
 *      - Valladolid 10594: Data Flow
 * REFERENCE:
 *      Edmonds, J., Karp, R.  "Theoretical Improvements in Algorithmic
 *          Efficieincy for Network Flow Problems".
 *      This is a slight improvement of Frank Chu's implementation.
 **/

#include <iostream>
#include <algorithm>
#include <climits>
using namespace std;

// the maximum number of vertices + 1
#define NN 1024

// adjacency matrix (fill this up)
int cap[NN][NN];

// cost per unit of flow matrix (fill this up)
int cost[NN][NN];

// flow network and adjacency list
int fnet[NN][NN], adj[NN][NN], deg[NN];

// Dijkstra's predecessor, depth and priority queue
int par[NN], d[NN], q[NN], inq[NN], qs;

// Labelling function
int pi[NN];

#define Inf (INT_MAX/2)
#define BUBL { \
t = q[i]; q[i] = q[j]; q[j] = t; \
t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }

// Dijkstra's using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
```



```
bool dijkstra( int n, int s, int t )
{
  fill(d, d+NN, Inf);
  fill(par, par+NN, -1);
  fill(inq, inq+NN, -1);

  d[s] = qs = 0;
  inq[q[qs++] = s] = 0;
  par[s] = n;

  while (qs) {
    // get the minimum from q and bubble down
    int u = q[0];
    inq[u] = -1;
    q[0] = q[--qs];
    if( qs ) inq[q[0]] = 0;
    for (int i = 0, j = 2*i + 1, t; j < qs; i = j, j = 2*i + 1) {
      if (j + 1 < qs && d[q[j + 1]] < d[q[j]]) j++;
      if (d[q[j]] >= d[q[i]]) break;
      BUBL;
    }

    // relax edge (u,i) or (i,u) for all i;
    for (int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k]) {
      // try undoing edge v->u
      if (fnet[v][u] && d[v] > Pot(u,v) - cost[v][u])
        d[v] = Pot(u,v) - cost[v][par[v] = u];

      // try using edge u->v
      if (fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v])
        d[v] = Pot(u,v) + cost[par[v] = u][v];

      if (par[v] == u) {
        // bubble up or decrease key
        if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs++; }
        for( int i = inq[v], j = ( i - 1 )/2, t;
             d[q[i]] < d[q[j]]; i = j, j = ( i - 1 )/2 )
          BUBL;
      }
    }
  }

  for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i] += d[i];

  return par[t] >= 0;
}

int mcmf( int n, int s, int t, int &fcost )
{
  // build the adjacency list
  fill(deg, deg+NN, 0);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      if (cap[i][j] || cap[j][i]) adj[i][deg[i]++] = j;
  }
  for (int i = 0; i < NN; i++) {
    fill(fnet[i], fnet[i]+NN, 0);
  }
  fill(pi, pi+NN, 0);

  int flow = fcost = 0;

  // repeatedly, find a cheapest path from s to t
  while (dijkstra(n, s, t)) {
    // get the bottleneck capacity
    int bot = INT_MAX;
    for (int v = t, u = par[v]; v != s; u = par[v = u]) {
      bot = min(bot, fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] ));
    }
```

```
    // update the flow network
    for (int v = t, u = par[v]; v != s; u = par[v = u])
      if (fnet[v][u]) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
      else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }

    flow += bot;
  }

  return flow;
}

int main()
{
  int numV;
  int m, a, b, c, cp;
  int s, t;

  cin >> numV;
  cin >> m;
  cin >> s >> t;

  // fill up cap with existing capacities.
  // if the edge u->v has capacity 6, set cap[u][v] = 6.
  // for each cap[u][v] > 0, set cost[u][v] to  the
  // cost per unit of flow along the edge u->v
  for (int i=0; i<m; i++) {
    cin >> a >> b >> cp >> c;
    cost[a][b] = c; // cost[b][a] = c;
    cap[a][b] = cp; // cap[b][a] = cp;
  }

  int fcost;
  int flow = mcmf( numV, s, t, fcost );
  cout << "flow: " << flow << endl;
  cout << "cost: " << fcost << endl;

  return 0;
}
```

```
/*
 * Implementation of Kruskal's Minimum Spanning Tree Algorithm
 *
 * Author: Howard Cheng
 *
 * This is a routine to find the minimum spanning tree.  It takes as
 * input:
 *
 *      n: number of vertices
 *      m: number of edges
 *  elist: an array of edges (if (u,v) is in the list, there is no need
 *         for (v,u) to be in, but it wouldn't hurt, as long as the weights
 *         are the same).
 *
 * The following are returned:
 *
 *  index: an array of indices that shows which edges from elist are in
 *         the minimum spanning tree.  It is assumed that its size is at
 *         least n-1.
 *   size: the number of edges selected in "index".  If this is not
 *         n-1, the graph is not connected and we have a "minimum
 *         spanning forest."
 *
 * The weight of the MST is returned as the function return value.
 *
 * The run time of the algorithm is O(m log m).
 *
 * Note: the elements of elist may be reordered.
 *
 * Modified by Rex Forsyth using C++  Aug 28, 2003
 * This version defines the unionfind and edge as classes and  provides
 * constructors. The edge class overloads the < operator. So the sort does
 * not use a  * cmp function. It uses dynamic arrays.
 */

#include <cmath>
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cassert>
#include <algorithm>
using namespace std;

class UnionFind
{
      struct UF { int p; int rank; };

   public:
      UnionFind(int n) {          // constructor
         howMany = n;
         uf = new UF[howMany];
         for (int i = 0; i < howMany; i++) {
            uf[i].p = i;
            uf[i].rank = 0;
         }
      }

      ~UnionFind() {
         delete[] uf;
      }

      int find(int x) { return find(uf,x); }        // for client use

      bool merge(int x, int y) {
         int res1, res2;
         res1 = find(uf, x);
         res2 = find(uf, y);
         if (res1 != res2) {
            if (uf[res1].rank > uf[res2].rank) {
```



```
               uf[res2].p = res1;
            }
            else {
               uf[res1].p = res2;
               if (uf[res1].rank == uf[res2].rank) {
                  uf[res2].rank++;
               }
            }
            return true;
         }
         return false;
      }

   private:
      int howMany;
      UF* uf;

      int find(UF uf[], int x) {               // for internal use
         if (uf[x].p != x) {
            uf[x].p = find(uf, uf[x].p);
         }
         return uf[x].p;
      }
};

class Edge {

   public:
      Edge(int i=-1, int j=-1, double weight=0) {
         v1 = i;
         v2 = j;
         w = weight;
      }
      bool operator<(const Edge& e) const { return w < e.w; }

      int v1, v2;          /* two endpoints of edge                */
      double w;            /* weight, can be double instead of int */
};


double mst(int n, int m, Edge elist[], int index[], int& size)
{
  UnionFind uf(n);

  sort(elist, elist+m);

  double w = 0.0;
  size = 0;
  for (int i = 0; i < m && size < n-1; i++) {
    int c1 = uf.find(elist[i].v1);
    int c2 = uf.find(elist[i].v2);
    if (c1 != c2) {
      index[size++] = i;
      w += elist[i].w;
      uf.merge(c1, c2);
    }
  }

  return w;
}

int main(void)
{
   cout << fixed << setprecision(2);

   int n;
   cin >> n;
   double* x = new double[n];
   double* y = new double[n];
```

```
   int* index = new int[n];

   for (int i = 0; i < n; i++)  cin >> x[i] >> y[i];

   Edge* elist = new Edge[n*n];
   int k = 0;
   for (int i = 0; i < n; i++)
      for (int j = i+1; j < n; j++)
         elist[k++] = Edge(i,j,hypot(x[i]-x[j], y[i]-y[j]) );

   int t;  // number of edges in the mst
   cout << mst(n, k, elist, index, t) << endl;
   return 0;
}
```

```
/*
 * Multiplication/division without overflow
 *
 * Author: Howard Cheng
 *
 * Given a list of factors in the numerator (num, size n) and a list
 * of factors in the denominator (dem, size m), it returns the product
 * of the numerator divided by the denominator, while reducing the
 * result as soon as it is larger than some BOUND.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

const int BOUND = (1 << 16);

double mult(double num[], int n, double dem[], int m)
{
   int i, j;
   double prod = 1.0;
   i = j = 0;
   while (i < n || j < m) {
      if (prod >= BOUND && j < m) {
         prod /= dem[j++];
      } else if (i < n) {
         prod *= num[i++];
      } else {
         assert(j < m);
         prod /= dem[j++];
      }
   }
   return prod;
}

int main(void)
{
  double A[1000], B[1000];
  int n, m, i;

  while (cin >> n >> m && n > 0 && m > 0) {
    for (i = 0; i < n; i++) {
      cin >> A[i];
    }
    for (i = 0; i < m; i++) {
      cin >> B[i];
    }
    cout << "prod = " << mult(A, n, B, m) << endl;
  }

  return 0;
}
```

```
/*
 * Network Flow (Relabel-to-front)
 *
 * Author: Howard Cheng
 *
 * The routine network_flow() finds the maximum flow that can be
 * pushed from the source (s) to the sink (t) in a flow network
 * (i.e. directed graph with capacities on the edges).  The maximum
 * flow is returned.  The flow is given in the flow array (look for
 * positive flow).
 *
 * The complexity of this algorithm is O(n^3), which is good if the
 * graph is small but the maximum flow can be large.  Since the
 * algorithm is O(n^3) we are going to use the adjacency matrix
 * representation.
 *
 */

#include <iostream>
#include <list>
#include <cassert>

using namespace std;

const int MAX_NODE = 102;

void clear_graph(int graph[MAX_NODE][MAX_NODE], int n)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      graph[i][j] = 0;
    }
  }
}

void push(int graph[MAX_NODE][MAX_NODE], int flow[MAX_NODE][MAX_NODE],
          int e[], int u, int v)
{
  int cf = graph[u][v] - flow[u][v];
  int d = (e[u] < cf) ? e[u] : cf;
  flow[u][v] += d;
  flow[v][u] = -flow[u][v];
  e[u] -= d;
  e[v] += d;
}

void relabel(int graph[MAX_NODE][MAX_NODE], int flow[MAX_NODE][MAX_NODE],
             int n, int h[], int u)
{
  h[u] = -1;
  for (int v = 0; v < n; v++) {
    if (graph[u][v] - flow[u][v] > 0 &&
        (h[u] == -1 || 1 + h[v] < h[u])) {
      h[u] = 1 + h[v];
    }
  }
  assert(h[u] >= 0);
}

void discharge(int graph[MAX_NODE][MAX_NODE], int flow[MAX_NODE][MAX_NODE],
               int n, int e[], int h[], list<int>& NU,
               list<int>::iterator &current, int u)
{
  while (e[u] > 0) {
    if (current == NU.end()) {
      relabel(graph, flow, n, h, u);
      current = NU.begin();
    } else {
      int v = *current;
```



```
      if (graph[u][v] - flow[u][v] > 0 && h[u] == h[v] + 1) {
        push(graph, flow, e, u, v);
      } else {
        ++current;
      }
    }
  }
}

int network_flow(int graph[MAX_NODE][MAX_NODE], int flow[MAX_NODE][MAX_NODE],
                 int n, int s, int t)
{
  int e[MAX_NODE], h[MAX_NODE];
  int u, v, oh;
  list<int> N[MAX_NODE], L;
  list<int>::iterator current[MAX_NODE], p;

  for (u = 0; u < n; u++) {
    h[u] = e[u] = 0;
  }
  for (u = 0; u < n; u++) {
    for (v = 0; v < n; v++) {
      flow[u][v] = 0;
      if (graph[u][v] > 0 || graph[v][u] > 0) {
        N[u].push_front(v);
      }
    }
  }
  h[s] = n;
  for (u = 0; u < n; u++) {
    if (graph[s][u] > 0) {
      e[u] = flow[s][u] = graph[s][u];
      e[s] += flow[u][s] = -graph[s][u];
    }
    if (u != s && u != t) {
      L.push_front(u);
    }
    current[u] = N[u].begin();
  }

  p = L.begin();
  while (p != L.end()) {
    u = *p;
    oh = h[u];
    discharge(graph, flow, n, e, h, N[u], current[u], u);
    if (h[u] > oh) {
      L.erase(p);
      L.push_front(u);
      p = L.begin();
    }
    ++p;
  }

  int maxflow = 0;
  for (u = 0; u < n; u++) {
    if (flow[s][u] > 0) {
      maxflow += flow[s][u];
    }
  }
  return maxflow;
}

void print_flow(int flow[MAX_NODE][MAX_NODE], int n)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (flow[i][j] > 0) {
        cout << i << "->" << j << ":" << flow[i][j] << endl;
      }
```

```
    }
  }
}

int main(void)
{
  int graph[MAX_NODE][MAX_NODE];
  int s, t;
  int n, m, u, v, c;
  int flow[MAX_NODE][MAX_NODE];
  int maxflow;

  while (cin >> n && n > 0) {
    clear_graph(graph, n);
    cin >> m >> s >> t;
    while (m-- > 0) {
      cin >> u >> v >> c;
      graph[u][v] = c;
    }
    maxflow = network_flow(graph, flow, n, s, t);
    cout << "flow = " << maxflow << endl;
    print_flow(flow, n);
  }

  return 0;
}
```

```
/*
 * Network Flow
 *
 * Author: Howard Cheng
 *
 * The routine network_flow() finds the maximum flow that can be
 * pushed from the source (s) to the sink (t) in a flow network
 * (i.e. directed graph with capacities on the edges).  The maximum
 * flow is returned.  Note that the graph is modified.  If you wish to
 * recover the flow on an edge, it is in the "flow" field, as long as
 * is_real is set to true.
 *
 * Note: if you have an undirected network. simply call add_edge twice
 * with an edge in both directions (same capacity).  Note that 4 edges
 * will be added (2 real edges and 2 residual edges).  To discover the
 * actual flow between two vertices u and v, add up the flow of all
 * real edges from u to v and subtract all the flow of real edges from
 * v to u.  (In fact, for a residual edge the flow is always 0 in this
 * implementation.)
 *
 * This code can also be used for bipartite matching by setting up an
 * appropriate flow network.
 *
 * The code here assumes an adjacency list representation since most
 * problems requiring network flow have sparse graphs.
 *
 * This is the basic augmenting path algorithm and it is not the most
 * efficient.  But it should be good enough for most programming contest
 * problems.  The complexity is O(f m) where f is the size of the flow
 * and m is the number of edges.  This is good if you know that f
 * is small, but can be exponential if f is large.
 *
 */

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include <cassert>

using namespace std;

struct Edge;
typedef list<Edge>::iterator EdgeIter;

struct Edge {
  int to;
  int cap;
  int flow;
  bool is_real;
  EdgeIter partner;

  Edge(int t, int c, bool real = true)
    : to(t), cap(c), flow(0), is_real(real)
  {};

  int residual() const
  {
    return cap - flow;
  }
};

struct Graph {
  list<Edge> *nbr;
  int num_nodes;
  Graph(int n)
    : num_nodes(n)
  {
    nbr = new list<Edge>[num_nodes];
```



```
  }

  ~Graph()
  {
    delete[] nbr;
  }

  // note: this routine adds an edge to the graph with the specified capacity,
  // as well as a residual edge.  There is no check on duplicate edge, so it
  // is possible to add multiple edges (and residual edges) between two
  // vertices
  void add_edge(int u, int v, int cap)
  {
    nbr[u].push_front(Edge(v, cap));
    nbr[v].push_front(Edge(u, 0, false));
    nbr[v].begin()->partner = nbr[u].begin();
    nbr[u].begin()->partner = nbr[v].begin();
  }
};

void push_path(Graph &G, int s, int t, const vector<EdgeIter> &path, int flow)
{
  for (int i = 0; s != t; i++) {
    if (path[i]->is_real) {
      path[i]->flow += flow;
      path[i]->partner->cap += flow;
    } else {
      path[i]->cap -= flow;
      path[i]->partner->flow -= flow;
    }
    s = path[i]->to;
  }
}

// the path is stored in a peculiar way for efficiency: path[i] is the
// i-th edge taken in the path.
int augmenting_path(const Graph &G, int s, int t, vector<EdgeIter> &path,
                    vector<bool> &visited, int step = 0)
{
  if (s == t) {
    return -1;
  }
  for (EdgeIter it = G.nbr[s].begin(); it != G.nbr[s].end(); ++it) {
    int v = it->to;
    if (it->residual() > 0 && !visited[v]) {
      path[step] = it;
      visited[v] = true;
      int flow = augmenting_path(G, v, t, path, visited, step+1);
      if (flow == -1) {
        return it->residual();
      } else if (flow > 0) {
        return min(flow, it->residual());
      }
    }
  }
  return 0;
}

// note that the graph is modified
int network_flow(Graph &G, int s, int t)
{
  vector<bool> visited(G.num_nodes);
  vector<EdgeIter> path(G.num_nodes);
  int flow = 0, f;

  do {
    fill(visited.begin(), visited.end(), false);
    if ((f = augmenting_path(G, s, t, path, visited)) > 0) {
      push_path(G, s, t, path, f);
```

```
      flow += f;
    }
  } while (f > 0);

  return flow;
}

int main(void)
{
  Graph G(100);
  int s, t, u, v, cap, flow;

  cin >> s >> t;
  while (cin >> u >> v >> cap) {
    G.add_edge(u, v, cap);
  }

  flow = network_flow(G, s, t);
  cout << "maximum flow = " << flow << endl;

  return 0;
}
```

```
/*
 * Point-in-polygon test
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.exaflop.org/docs/cgafaq/cga2.html
 *
 * Given a polygon as a list of n vertices, and a point, it returns
 * whether the point is in the polygon or not.
 *
 * One has the option to define the behavior on the boundary.
 *
 */

#include <iostream>
#include <cmath>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

/* what should be returned on the boundary? */
const bool BOUNDARY = true;

struct Point {
  double x, y;
};

/* counterclockwise, clockwise, or undefined */
enum Orientation {CCW, CW, CNEITHER};

Orientation ccw(Point a, Point b, Point c)
{
  double dx1 = b.x - a.x;
  double dx2 = c.x - b.x;
  double dy1 = b.y - a.y;
  double dy2 = c.y - b.y;
  double t1 = dy2 * dx1;
  double t2 = dy1 * dx2;

  if (fabs(t1 - t2) < EPSILON) {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) {
      if (dx1*dx1 + dy1*dy1 >= dx2*dx2 + dy2*dy2 - EPSILON) {
        return CNEITHER;
      } else {
        return CW;
      }
    } else {
      return CCW;
    }
  } else if (t1 > t2) {
    return CCW;
  } else {
    return CW;
  }
}

bool point_in_poly(Point poly[], int n, Point p)
{
  int i, j, c = 0;

  /* first check to see if point is one of the vertices */
  for (i = 0; i < n; i++) {
    if (fabs(p.x - poly[i].x) < EPSILON && fabs(p.y - poly[i].y) < EPSILON) {
      return BOUNDARY;
    }
  }
```



```
  /* now check if it's on the boundary */
  for (i = 0; i < n-1; i++) {
    if (ccw(poly[i], poly[i+1], p) == CNEITHER) {
      return BOUNDARY;
    }
  }
  if (ccw(poly[n-1], poly[0], p) == CNEITHER) {
    return BOUNDARY;
  }

  /* finally check if it's inside */
  for (i = 0, j = n-1; i < n; j = i++) {
    if (((poly[i].y <= p.y && p.y < poly[j].y) ||
         (poly[j].y <= p.y && p.y < poly[i].y)) &&
        (p.x < (poly[j].x - poly[i].x) * (p.y - poly[i].y)
         / (poly[j].y - poly[i].y) + poly[i].x))
      c = !c;
  }
  return c;
}

int main(void)
{
  Point *polygon, p;
  int n;
  int i;

  while (cin >> n && n > 0) {
    polygon = new Point[n];
    assert(polygon);
    for (i = 0; i < n; i++) {
      cin >> polygon[i].x >> polygon[i].y;
    }
    while (cin >> p.x >> p.y) {
      if (point_in_poly(polygon, n, p)) {
        cout << "yes";
      } else {
        cout << "no";
      }
      cout << endl;
    }
    delete[] polygon;
  }
  return 0;
}
```

```
/*
 * Convex Polygon Intersection
 *
 * Author: Howard Cheng
 *
 * This routine takes two convex polygon, and returns the intersection
 * which is also convex.  If the intersection contains less than
 * 3 points, it is considered empty.
 *
 */

#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <cassert>

using namespace std;

/* how close to call equal */
const double EPSILON = 1E-8;

struct Point {
  double x, y;
};

const bool BOUNDARY = true;

/* counterclockwise, clockwise, or undefined */
enum Orientation {CCW, CW, CNEITHER};

/* Global point for computing convex hull */
Point start_p;

Orientation ccw(Point a, Point b, Point c)
{
  double dx1 = b.x - a.x;
  double dx2 = c.x - b.x;
  double dy1 = b.y - a.y;
  double dy2 = c.y - b.y;
  double t1 = dy2 * dx1;
  double t2 = dy1 * dx2;

  if (fabs(t1 - t2) < EPSILON) {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) {
      if (dx1*dx1 + dy1*dy1 >= dx2*dx2 + dy2*dy2 - EPSILON) {
        return CNEITHER;
      } else {
        return CW;
      }
    } else {
      return CCW;
    }
  } else if (t1 > t2) {
    return CCW;
  } else {
    return CW;
  }
}

bool ccw_cmp(const Point &a, const Point &b)
{
  return ccw(start_p, a, b) == CCW;
}

int convex_hull(Point polygon[], int n, Point hull[]) {
  int count, best_i, i;

  if (n == 1) {
```



```
    hull[0] = polygon[0];
    return 1;
  }

  /* find the first point: min y, and then min x */
  start_p = polygon[0];
  best_i = 0;
  for (i = 1; i < n; i++) {
    if ((polygon[i].y < start_p.y) ||
        (polygon[i].y == start_p.y && polygon[i].x < start_p.x)) {
      start_p = polygon[i];
      best_i = i;
    }
  }
  polygon[best_i] = polygon[0];
  polygon[0] = start_p;

  /* get simple closed polygon */
  sort(polygon+1, polygon+n, ccw_cmp);

  /* do convex hull */
  count = 0;
  hull[count] = polygon[count]; count++;
  hull[count] = polygon[count]; count++;
  for (i = 2; i < n; i++) {
    while (count > 1 &&
           ccw(hull[count-2], hull[count-1], polygon[i]) == CW) {
      /* pop point */
      count--;
    }
    hull[count++] = polygon[i];
  }
  return count;
}

bool point_in_poly(Point poly[], int n, Point p)
{
  int i, j, c = 0;

  /* first check to see if point is one of the vertices */
  for (i = 0; i < n; i++) {
    if (fabs(p.x - poly[i].x) < EPSILON && fabs(p.y - poly[i].y) < EPSILON) {
      return BOUNDARY;
    }
  }

  /* now check if it's on the boundary */
  for (i = 0; i < n-1; i++) {
    if (ccw(poly[i], poly[i+1], p) == CNEITHER) {
      return BOUNDARY;
    }
  }
  if (ccw(poly[n-1], poly[0], p) == CNEITHER) {
    return BOUNDARY;
  }

  /* finally check if it's inside */
  for (i = 0, j = n-1; i < n; j = i++) {
    if (((poly[i].y <= p.y && p.y < poly[j].y) ||
         (poly[j].y <= p.y && p.y < poly[i].y)) &&
        (p.x < (poly[j].x - poly[i].x) * (p.y - poly[i].y)
         / (poly[j].y - poly[i].y) + poly[i].x))
      c = !c;
  }
  return c;
}

/* returns 1 if intersect at a point, 0 if not, -1 if the lines coincide */
int intersect_line(Point a, Point b, Point c, Point d, Point &p)
```

```
{
  double r, s;
  double denom, num1, num2;

  num1 = (a.y - c.y)*(d.x - c.x) - (a.x - c.x)*(d.y - c.y);
  num2 = (a.y - c.y)*(b.x - a.x) - (a.x - c.x)*(b.y - a.y);
  denom = (b.x - a.x)*(d.y - c.y) - (b.y - a.y)*(d.x - c.x);

  if (fabs(denom) >= EPSILON) {
    r = num1 / denom;
    s = num2 / denom;
    if (-EPSILON <= r && r <= 1+EPSILON && -EPSILON <= s && s <= 1+EPSILON) {
      p.x = a.x + r*(b.x - a.x);
      p.y = a.y + r*(b.y - a.y);
      return 1;
    } else {
      return 0;
    }
  } else {
    if (fabs(num1) >= EPSILON) {
      return 0;
    } else {
      return -1;
    }
  }
}

int intersect_polygon(Point poly1[], int n1, Point poly2[], int n2,
                      Point *&out)
{
  Point *newpoly, p;
  char *used;
  int new_n = n1 + n2 + n1*n2;
  int count, i, i2, j, j2, new_count;
  int n;

  newpoly = new Point[new_n];
  out = new Point[new_n];
  used = new char[new_n];
  assert(newpoly && out && used);
  count = 0;
  fill(used, used+new_n, 0);

  for (i = 0; i < n1; i++) {
    if (point_in_poly(poly2, n2, poly1[i])) {
      newpoly[count++] = poly1[i];
    }
  }
  for (i = 0; i < n2; i++) {
    if (point_in_poly(poly1, n1, poly2[i])) {
      newpoly[count++] = poly2[i];
    }
  }

  for (i = 0; i < n1; i++) {
    i2 = (i+1 == n1) ? 0 : i+1;
    for (j = 0; j < n2; j++) {
      j2 = (j+1 == n2) ? 0 : j+1;
      if (intersect_line(poly1[i], poly1[i2], poly2[j], poly2[j2], p) == 1) {
        newpoly[count++] = p;
      }
    }
  }

  if (count >= 3) {
    n = convex_hull(newpoly, count, out);
    if (n < 3) {
      delete[] out;
      n = 0;
```



```
    }
  } else {
    delete[] out;
    n = 0;
  }

  /* eliminate duplicates */
  for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
      if (out[i].x == out[j].x && out[i].y == out[j].y) {
        used[j] = 1;
      }
    }
  }
  j = 0;
  new_count = 0;
  for (i = 0; i < n; i++) {
    if (!used[i]) {
      out[new_count++] = out[i];
    }
  }
  n = new_count;

  delete[] newpoly;
  delete[] used;
  return n;
}

int read_poly(Point *&poly)
{
  int n, i;

  cin >> n;
  if (n == 0) {
    return 0;
  }
  poly = new Point[n];
  assert(poly);
  for (i = 0; i < n; i++) {
    cin >> poly[i].x >> poly[i].y;
  }
  return n;

}

int main(void)
{
  Point *poly1, *poly2, *intersection;
  int n1, n2, n3, i;

  while ((n1 = read_poly(poly1))) {
    n2 = read_poly(poly2);
    n3 = intersect_polygon(poly1, n1, poly2, n2, intersection);
    delete[] poly1;
    delete[] poly2;
    if (n3 >= 3) {
      for (i = 0; i < n3; i++) {
        cout << fixed << setprecision(2);
        cout << "(" << intersection[i].x << "," << intersection[i].y
             << ")";
      }
      cout << endl;
      delete[] intersection;
    } else {
      cout << "Empty Intersection" << endl;
    }
  }

  return 0;
```

```
}
```

```
// Performs guassian elimination over the rationals.
//
// Author: Darcy Best
// Date  : September 22, 2010
//
// pair<int,int> means first = numerator, second = denominator

#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

#define pii pair<int,int>
const int MAX_N = 100;

pii *r_m,m_m;

void print(pii x){
  if(x.second == 1)
    cout << x.first;
  else
    cout << x.first <<  "/" << x.second;
}

void print(pii A[MAX_N][MAX_N],int m,int n){
  for(int i=0;i<m;i++){
    for(int j=0;j<n;j++){
      cout << setw(5);
      print(A[i][j]);
    }
    cout << endl;
  }
  cout << endl;
}

void read(pii& x){
  cin >> x.first;
  char ch;
  if(cin.peek() == '/')
    cin >> ch >> x.second;
  else
    x.second = 1;
}

int gcd(int a,int b){
  while (b) {
    int r = a % b;
    a = b;
    b = r;
  }
  return a;
}

pii reduce(pii a){
  if(a.first == 0){
    a.second = 1;
  } else {
    if(a.second < 0){
      a.first *= -1;
      a.second *= -1;
    }
    int g = gcd(abs(a.first),a.second);
    a.first /= g;
    a.second /= g;
  }
  return a;
}

pii operator*(pii a,pii b){
```



```
  return reduce(pii(a.first*b.first,a.second*b.second));
}

pii operator+(pii a,pii b){
  return reduce(pii(a.first*b.second+b.first*a.second,a.second*b.second));
}

void multRow(pii& x){
  x = x * m_m;
}

void addMultRow(pii& x){
  x = x + (m_m * (*r_m++));
}

int rowReduction(pii A[MAX_N][MAX_N],int rows,int cols){
  int rank = 0;
  for(int c=0;c<cols;c++){
    for(int r=rank;r<rows;r++){
      if(A[r][c].first){
        if(r != rank) // Swap rows
          swap_ranges(A[rank],A[rank]+cols,A[r]);
        if(c == cols-1) // Inconsistent
          return -1;

        // Make first entry 1
        m_m = pii(A[rank][c].second,A[rank][c].first);
        for_each(A[rank]+c+1,A[rank]+cols,multRow);
        A[rank][c] = pii(1,0);

        for(int i=(arb?rank+1:0);i<rows;i++)
          if(A[i][c].first && i != rank){
            // Make the other rows 0
            m_m = pii(-A[i][c].first,A[i][c].second);
            r_m = A[rank]+c+1;
            for_each(A[i]+c+1,A[i]+cols,addMultRow);
            A[i][c] = pii(0,1);
          }
        rank++;
        break;
      }
    }
  }
  return rank;
}

int main(){
  int C=0;
  int T,m,n,rank;
  pii A[MAX_N][MAX_N];
  while(cin >> T && T){
    if(C++)
      cout << endl;
    cout << "Solution for Matrix System # " << T << endl;
    cin >> n >> m;
    for(int i=0;i<m;i++)
      for(int j=0;j<=n;j++)
        read(A[i][j]);

    if((rank = rowReduction(A,m,n+1)) < 0){
      cout << "No Solution." << endl;
    } else {
      if(rank != n){
        cout << "Infinitely many solutions containing " << n-rank << " arbitrary constants." << endl
;
      } else {
        for(int i=0;i<n;i++){
          cout << "x[" << i+1 << "] = ";print(A[i][n]); cout << endl;
        }
```

```
      }
    }
  }
  return 0;
}
```

```
// Converts Roman Numerals to Arabic Numbers (and vice versa)
//
// Author: Darcy Best
// Date  : September 5, 2010
//
// If you are given a valid integer (0 < x < 4000), then it will give
//    the standard roman numeral representation of it. Note that if you give
//    it a number such that x >= 4000, then it will just append as many "M"s
//    as needed.
//
// If you are given a valid roman numberal, then it will give you the answer
//    as a base 10 number.

#include <iostream>
#include <string>
#include <map>
using namespace std;

const string Roman[13]  = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","I
V","I"};
const int Arabic[13] = {1000,900,500,400,100,90,50,40,10,9,5,4,1};

string toRoman(int x){
  string roman;
  for(int i=0;i<13;i++)
    while(x >= Arabic[i]){
      x -= Arabic[i];
      roman += Roman[i];
    }
  return roman;
}

int toInt(string s){
  int L1,L2,ind=0,ans=0;
  while(ind < 13){
    L1 = s.length();
    L2 = Roman[ind].length();
    if(s.substr(0,min(L1,L2)) == Roman[ind]){
      ans += Arabic[ind];
      s.erase(0,min(L1,L2));
    } else {
      ind++;
    }
  }
  return ans;
}

int main(){
  char c;
  int x;
  string s;

  // Checks to see if the line is Roman Numerals or Arabic Numbers,
  //   then converts to the opposite.
  while(cin >> c){
    cin.putback(c);
    if(c >= '0' && c <= '9'){
      cin >> x;
      cout << toRoman(x) << endl;
    } else {
      cin >> s;
      cout << toInt(s) << endl;
    }
  }
  return 0;
}
```

```
// Compresses a directed graph into its strongly connected components
//
// Author: Darcy Best
// Date  : October 1, 2010
//
// A set of nodes is "strongly connected" if for any pair of nodes in
// the set, there is a path from u to v AND from v to u.
//
// Compressing a graph into its strongly connected components means
// converting each strongly connected component into a super-node.
//
// We then build a "compressed" graph made with the super-nodes.  We
// add an edge in the compressed graph between U and V if there is a
// vertex u in U and v in V such that there was an edge from u to v in
// the original graph. The compressed graph will be a Directed Acyclic
// Graph (DAG), and the list of components will be in REVERSE
// topological order.
//
// If you are only concerned with the number of strongly connected
// components, you do not need to build the graph. See comments below
// on how to remove the SCC graph.
//
// The complexity of this algorithm is O(|V| + |E|).
//

#include <iostream>
#include <algorithm>
#include <stack>
#include <cassert>
#include <vector>
#include <set>
using namespace std;

const int MAX_NODES = 100005;

struct Graph{
  int numNodes;
  set<int> adj[MAX_NODES];
  void clear(){
    numNodes = 0;
    for(int i=0;i<MAX_NODES;i++)
      adj[i].clear();
  }
  void add_edge(int u,int v){
    adj[u].insert(v);
  }
};

int po[MAX_NODES],comp[MAX_NODES];

void DFS(int v,const Graph& G,Graph& G_scc,int& C,
         stack<int>& P,stack<int>& S){
  po[v] = C++;

  S.push(v);  P.push(v);
  for(auto w : G.adj[v]) {
    if(po[w] == -1){
      DFS(w,G,G_scc,C,P,S);
    } else if(comp[w] == -1){
      while(!P.empty() && (po[P.top()] > po[w]))
        P.pop();
    }
  }
  if(!P.empty() && P.top() == v){
    while(!S.empty()){
      int t = S.top();
      S.pop();
      comp[t] = G_scc.numNodes;
      if(t == v)
```



```
        break;
    }
    G_scc.numNodes++;
    P.pop();
  }
}

int SCC(const Graph& G,Graph& G_scc){
  G_scc.clear();
  int C=1;
  stack<int> P,S;
  fill(po,po+G.numNodes,-1);
  fill(comp,comp+G.numNodes,-1);
  for(int i=0;i<G.numNodes;i++)
    if(po[i] == -1)
      DFS(i,G,G_scc,C,P,S);

  // You do not need this if you are only interested in the number of
  //    strongly connected components.
  for(int i=0;i<G.numNodes;i++){
    for(auto w : G.adj[i]) {
      if(comp[i] != comp[w])
        G_scc.add_edge(comp[i],comp[w]);
    }
  }

  return G_scc.numNodes;
}

// Declare these as a global variable if MAX_NODES is large to
//   avoid Runtime Error.
Graph G,G_scc;

int main(){
  int u,v,m,n;
  int n_scc;
  while(cin >> n >> m && (n || m)){
    G.clear();
    G.numNodes = n;
    for(int i=0;i<m;i++){
      cin >> u >> v;
      G.add_edge(u,v);
    }
    n_scc = SCC(G,G_scc);

    cout << "# of Strongly Connected Components: " << n_scc << endl;
  }
  return 0;
}
```

```
#include <algorithm>

using namespace std;

const int MAX_CONSTRAINTS = 100;
const int MAX_VARS = 100;
const int MAXM = MAX_CONSTRAINTS + 1;
const int MAXN = MAX_VARS + 1;

const double EPS = 1e-9;
const double INF = 1.0/0.0;

double A[MAXM][MAXN];
int basis[MAXM], out[MAXN];

void pivot(int m, int n, int a, int b)
{
  int i, j;
  for (i = 0; i <= m; i++)
    if (i != a)
      for (j = 0; j <= n; j++)
        if (j != b)
          A[i][j] -= A[a][j] * A[i][b] / A[a][b];
  for (j = 0; j <= n; j++)
    if (j != b) A[a][j] /= A[a][b];
  for (i = 0; i <= m; i++)
    if (i != a) A[i][b] = -A[i][b] / A[a][b];
  A[a][b] = 1 / A[a][b];
  swap(basis[a], out[b]);
}

double simplex(int m, int n, double C[][MAXN], double X[])
{
  int i, j, ii, jj;
  for (i = 1; i <= m; i++)
    copy(C[i], C[i]+n+1, A[i]);
  for (j = 0; j <= n; j++)
    A[0][j] = -C[0][j];
  for (i = 0; i <= m; i++)
    basis[i] = -i;
  for (j = 0; j <= n; j++)
    out[j] = j;
  for (;;) {
    for (i = ii = 1; i <= m; i++)
      if (A[i][n] < A[ii][n] || (A[i][n] == A[ii][n] && basis[i] < basis[ii]))
        ii = i;
    if (A[ii][n] >= -EPS) break;
    for (j = jj = 0; j < n; j++)
      if (A[ii][j] < A[ii][jj] - EPS ||
          (A[ii][j] < A[ii][jj] - EPS && out[i] < out[j]))
        jj = j;
    if (A[ii][jj] >= -EPS) return -INF;
    pivot(m, n, ii, jj);
  }
  for (;;) {
    for (j = jj = 0; j < n; j++)
      if (A[0][j] < A[0][jj] || (A[0][j] == A[0][jj] && out[j] < out[jj]))
        jj = j;
    if (A[0][jj] > -EPS) break;
    for (i=1, ii=0; i <= m; i++)
      if ((A[i][jj]>EPS) &&
          (!ii || (A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]-EPS) ||
           ((A[i][n]/A[i][jj] < A[ii][n]/A[ii][jj]+EPS) &&
            (basis[i] < basis[ii]))))
        ii = i;
    if (A[ii][jj] <= EPS) return INF;
    pivot(m, n, ii, jj);
  }
  fill(X, X+n, 0);
```



```
  for (i = 1; i <= m; i++)
    if (basis[i] >= 0)
      X[basis[i]] = A[i][n];
  return A[0][n];
}

#include <iostream>
#include <iomanip>


int main(void)
{
  double C[MAXM][MAXN], X[MAX_VARS];

  C[0][0] = -1;   C[0][1] = -3;  C[0][2] = 0;  C[0][3] = 0;
  C[1][0] = -2;  C[1][1] = -3;  C[1][2] = -6; C[1][3] = 250;
  C[2][0] = -1;   C[2][1] = -5;   C[2][2] = -5;  C[2][3] = 400;
  C[3][0] = -59;   C[3][1] = -35;   C[3][2] = -160;  C[3][3] = 30;

  double val = simplex(2, 2, C, X);

  cout << fixed << setprecision(3);
  cout << "val = " << val << endl;
  cout << "X[0] = " << X[0] << endl;
  cout << "X[1] = " << X[1] << endl;
  //  cout << "X[2] = " << X[2] << endl;
  return 0;
}
```

```
/*
 * Finding the lexicographically least rotation of a string, and finding
 * the smallest period of a string.
 *
 * Author: Sumudu Fernando
 *
 * Given a string, the algorithm can be used to compute two things:
 *
 *   a) the position at which the lexicographically least rotation starts.
 *      If there are ties, give the first position.
 *   b) the length of the shortest substring such that the original string
 *      is a concatenation of copies of that substring
 *
 * Complexity: O(n) where n = length of the string
 *
 * Tested on: 719            Glass Beads
 *            10298          Power Strings
 *            ACPC 2011 H    Let's call a SPaDE a SPaDE
 */

#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

// pos = position of the start of the lexicographically least rotation
// period = the period
void compute(string s, int &pos, int &period)
{
  s += s;
  int len = s.length();
  int i = 0, j = 1;
  for (int k = 0; i+k < len && j+k < len; k++) {
    if (s[i+k] > s[j+k]) {
      i = max(i+k+1, j+1);
      k = -1;
    } else if (s[i+k] < s[j+k]) {
      j = max(j+k+1, i+1);
      k = -1;
    }
  }

  pos = min(i, j);
  period = (i > j) ? i - j : j - i;
}

int main(void)
{
  string s;
  while (cin >> s) {
    int pos, period;
    compute(s, pos, period);
    int n = s.length();
    s += s;
    cout << "least rotation = " << s.substr(pos, n) << endl;
    cout << "period = " << s.substr(0, period) << endl;
  }
  return 0;
}
```

```
/*
 * Suffix array
 *
 * Author: Howard Cheng
 * References:
 *   Manber, U. and Myers, G.  "Suffix Arrays: a New Method for On-line
 *   String Searches."  SIAM Journal on Computing.  22(5) p. 935-948, 1993.
 *
 *   T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park.  "Linear-time
 *   Longest-common-prefix Computation in Suffix Arrays and Its
 *   Applications."  Proc. 12th Annual Conference on Combinatorial
 *   Pattern Matching, LNCS 2089, p. 181-192, 2001
 *
 *   J. Kärkkäinen and P. Sanders. Simple linear work suffix array
 *   construction.  In Proc. 13th International Conference on Automata,
 *   Languages and Programming, Springer, 2003
 *
 * The build_sarray routine takes in a string str of n characters (null-
 * terminated), and construct an array sarray.  Optionally, you can also
 * construct an lcp array from the sarray computed.  The properties
 * are:
 *
 * - If p = sarray[i], then the suffix of str starting at p (i.e.
 *   str[p..n-1] is the i-th suffix when all the suffixes are sorted in
 *   lexicographical order
 *
 *   NOTE: the empty suffix is not included in this list, so sarray[0] != n.
 *
 * - lcp[i] contains the length of the longest common prefix of the suffixes
 *   pointed to by sarray[i-1] and sarray[i].  lcp[0] is defined to be 0.
 *
 * - To see whether a pattern P occurs in str, you can look for it as
 *   the prefix of a suffix.  This can be done with a binary search in
 *   O(|P| log n) time.  Call find() to return a pair <L, R> such that
 *   all occurrences of the pattern are at positions sarray[i] with
 *   L <= i < R.  If L == R then there is no match.
 *
 * The construction of the suffix array takes O(n) time.
 *
 */

#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
#include <climits>

using namespace std;

bool leq(int a1, int a2,   int b1, int b2)
{
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}

bool leq(int a1, int a2, int a3,   int b1, int b2, int b3)
{
  return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));
}

void radixPass(int* a, int* b, int* r, int n, int K)
{
  int* c = new int[K + 1];
  fill(c, c+K+1, 0);
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;
  for (int i = 0, sum = 0;  i <= K;  i++) {
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];
  delete [] c;
```



```
}

#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)

void sarray_int(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);

  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
      name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; }
    else                  { s12[SA12[i]/3 + n0] = name; }
  }

  if (name < n02) {
    sarray_int(s12, SA12, n02, name);
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
    int i = GetI();
    int j = SA0[p];
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    {
      SA[k] = i;  t++;
      if (t == n02) {
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  {
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}

void build_sarray(string str, int sarray[])
{
  int n = str.length();

  if (n <= 1) {
    for (int i = 0; i < n; i++) {
      sarray[i] = i;
    }
    return;
  }

  int *s = new int[n+3];
  int *SA = new int[n+3];
```

```
  for (int i = 0; i < n; i++) {
    s[i] = (int)str[i] - CHAR_MIN + 1;
  }
  s[n] = s[n+1] = s[n+2] = 0;
  sarray_int(s, SA, n, 256);
  copy(SA, SA+n, sarray);

  delete[] s;
  delete[] SA;
}

void compute_lcp(string str, int sarray[], int lcp[])
{
  int n = str.length();
  int *rank = new int[n];
  for (int i = 0; i < n; i++) {
    rank[sarray[i]] = i;
  }
  int h = 0;
  for (int i = 0; i < n; i++) {
    int k = rank[i];
    if (k == 0) {
      lcp[k] = -1;
    } else {
      int j = sarray[k-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) {
        h++;
      }
      lcp[k] = h;
    }
    if (h > 0) {
      h--;
    }
  }
  lcp[0] = 0;
  delete[] rank;
}

pair<int,int> find(const string &str, const int sarray[],
                   const string &pattern)
{
  int n = str.length(), p = pattern.length();
  int L, R;

  if (pattern <= str.substr(sarray[0], p)) {
    L = 0;
  } else if (pattern > str.substr(sarray[n-1], p)) {
    L = n;
  } else {
    int lo = 0, hi = n-1;
    while (hi - lo > 1) {
      int mid = lo + (hi - lo)/2;
      if (pattern <= str.substr(sarray[mid], p)) {
        hi = mid;
      } else {
        lo = mid;
      }
    }
    L = hi;
  }

  if (pattern < str.substr(sarray[0], p)) {
    R = 0;
  } else if (pattern >= str.substr(sarray[n-1], p)) {
    R = n;
  } else {
    int lo = 0, hi = n-1;
    while (hi - lo > 1) {
      int mid = lo + (hi - lo)/2;
```



```
      if (pattern < str.substr(sarray[mid], p)) {
        hi = mid;
      } else {
        lo = mid;
      }
    }
    R = hi;
  }

  if (L > R) R = L;
  return make_pair(L, R);
}

int main(void)
{
  string str;
  int sarray[100], lcp[100];
  unsigned int i;

  while (cin >> str) {
    build_sarray(str, sarray);
    compute_lcp(str, sarray, lcp);
    for (i = 0; i < str.length(); i++) {
      cout << setw(3) << i << ":" << setw(2) << lcp[i] << ","
           << str.substr(sarray[i], str.length()-sarray[i]) << endl;
    }
  }
  return 0;
}
```

```
/*
 * Topological sort
 *
 * Author: Howard Cheng
 *
 * Given a directed acyclic graph, the topological_sort routine
 * returns a vector of integers that gives the vertex number (0 to n-1)
 * such that if there is a path from v1 to v2, then v1 occurs earlier
 * than v2 in the order.  Note that the topological sort result is not
 * necessarily unique.
 *
 * topological_sort returns true if there is no cycle.  Otherwise it
 * returns false and the sorting is unsuccessful.
 *
 * The complexity is O(n + m).
 *
 */


#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;

typedef int Edge;
typedef vector<Edge>::iterator EdgeIter;

struct Graph {
  vector<Edge> *nbr;
  int num_nodes;
  Graph(int n)
    : num_nodes(n)
  {
    nbr = new vector<Edge>[num_nodes];
  }

  ~Graph()
  {
    delete[] nbr;
  }

  // note: There is no check on duplicate edge, so it is possible to
  // add multiple edges between two vertices
  void add_edge(int u, int v)
  {
    nbr[u].push_back(Edge(v));
  }
};

bool topological_sort(const Graph &G, vector<int> &order)
{
  vector<int> indeg(G.num_nodes);
  fill(indeg.begin(), indeg.end(), 0);
  for (int i = 0; i < G.num_nodes; i++) {
    for (int j = 0; j < G.nbr[i].size(); j++) {
      indeg[G.nbr[i][j]]++;
    }
  }

  // use a priority queue if you want to get a topological sort order
  // with ties broken by lexicographical ordering
  queue<int> q;
  for (int i = 0; i < G.num_nodes; i++) {
    if (indeg[i] == 0) {
      q.push(i);
    }
  }
```



```
  order.clear();
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    order.push_back(v);
    for (int i = 0; i < G.nbr[v].size(); i++) {
      if (--indeg[G.nbr[v][i]] == 0) {
        q.push(G.nbr[v][i]);
      }
    }
  }

  return order.size() == G.num_nodes;
}


int main(void)
{
  int n, m;

  while (cin >> n >> m && (n || m)) {
    Graph G(n);
    for (int i = 0; i < m; i++) {
      int u, v;
      cin >> u >> v;
      G.add_edge(u, v);
    }
    vector<int> order;
    if (topological_sort(G, order)) {
      for (int i = 0; i < n; i++) {
        if (i) cout << ' ';
        cout << order[i];
      }
      cout << endl;
    } else {
      cout << "there is a cycle" << endl;
    }
  }
  return 0;
}
```

```
// UnionFind class -- based on Howard Cheng's C code for UnionFind
// Modified to use C++ by Rex Forsyth, Oct 22, 2003
//
// Constuctor -- builds a UnionFind object of size n and initializes it
// find -- return index of x in the UnionFind
// merge -- updates relationship between x and y in the UnionFind


class UnionFind
{
      struct UF { int p; int rank; };

   public:
      UnionFind(int n) {          // constructor
         howMany = n;
         uf = new UF[howMany];
         for (int i = 0; i < howMany; i++) {
            uf[i].p = i;
            uf[i].rank = 0;
         }
      }

      ~UnionFind() {
         delete[] uf;
      }

      int find(int x) { return find(uf,x); }        // for client use

      bool merge(int x, int y) {
         int res1, res2;
         res1 = find(uf, x);
         res2 = find(uf, y);
         if (res1 != res2) {
            if (uf[res1].rank > uf[res2].rank) {
               uf[res2].p = res1;
            }
            else {
               uf[res1].p = res2;
               if (uf[res1].rank == uf[res2].rank) {
                  uf[res2].rank++;
               }
            }
            return true;
         }
         return false;
      }

   private:
      int howMany;
      UF* uf;

      int find(UF uf[], int x) {     // recursive funcion for internal use
         if (uf[x].p != x) {
            uf[x].p = find(uf, uf[x].p);
         }
         return uf[x].p;
      }
};
```

```
/*
 * Largest subvector sum
 *
 * Author: Howard Cheng
 * Reference: Programming Pearl, page 74
 *
 * Given an array of integers, we find the continguous subvector that
 * gives the maximum sum.  If all entries are negative, it returns
 * an empty vector with sum = 0.
 *
 * If we want the subvector to be nonempty, we should first scan for the
 * largest element in the vector (1-element subvector) and combine the
 * result in this routine.
 *
 * The sum is returned, as well as the start and the end position
 * (inclusive).  If start > end, then the subvector is empty.
 *
 */

#include <iostream>
#include <cassert>

using namespace std;

int vecsum(int v[], int n, int &start, int &end)
{
  int maxval = 0;
  int max_end = 0;
  int max_end_start, max_end_end;
  int i;

  start = max_end_start = 0;
  end = max_end_end = -1;
  for (i = 0; i < n; i++) {
    if (v[i] + max_end >= 0) {
      max_end = v[i] + max_end;
      max_end_end = i;
    } else {
      max_end_start = i+1;
      max_end_end = -1;
      max_end = 0;
    }

    if (maxval < max_end) {
      start = max_end_start;
      end = max_end_end;
      maxval = max_end;
    } else if (maxval == max_end) {
      /* put whatever preferences we have for a tie */
      /* eg. longest subvector, and then the one that starts the earliest */
      if (max_end_end - max_end_start > end - start ||
          (max_end_end - max_end_start == end - start &&
           max_end_start < start)) {
        start = max_end_start;
        end = max_end_end;
        maxval = max_end;
      }
    }
  }
  return maxval;
}

int main(void)
{
  int n;
  int *v;
  int i;
  int sum, start, end;
```



```
  while (cin >> n && n > 0) {
    v = new int[n];
    assert(v);
    for (i = 0; i < n; i++) {
      cin >> v[i];
    }
    sum = vecsum(v, n, start, end);
    cout << "Maximum sum " << sum << " from " << start << " to " << end << "."
         << endl;
    delete[] v;
  }

  return 0;
}
```

```c
/*
 * Zero-one programming
 *
 * Author: Howard Cheng
 * Reference:
 *   http://www.cs.sunysb.edu/~algorith/implement/syslo/distrib/processed/
 *
 * This algorithm is based on BALAS branching testing.
 *
 * This algorithm solves the BINARY linear program:
 *
 *      min  cx                 [cost function]
 *      s.t.
 *           Ax <= b            [constraints]
 *           x[i] = 0 or 1.
 *
 * where A is an m x n matrix,
 *       c and x are n-dimensional vectors,
 *       b is an m-dimensional vector.
 *
 * n = number of variables
 * m = number of constraints
 *
 * It returns whether there exists a solution.
 * The optimal value of the cost function is returned in value.
 * The assignment giving the optimal cost function value is returned in x.
 *
 * Important Notes:
 *
 * 1. The matrices and arrays start their indices at 1!!!!!!
 * 2. If we want to have constraints that are >=, just multiply all the
 *    coefficients by -1.
 * 3. If we want to have constraints that are ==, do both >= and <=.
 * 4. The content of A, b, and c is preserved after this routine.
 * 5. The coefficients in the cost vector c must be positive.  If not,
 *    make a change of variable x'[i] = 1-x[i] and adjust all constraints
 *    as well as the returned optimal value.  This is especially useful
 *    if you wish to maximize the cost function.
 *
 */

#include <stdio.h>
#include <limits.h>
#include <assert.h>

#define MAX_VAR 1000
#define MAX_CONS 100
#define MAX_ROWS MAX_CONS+1
#define MAX_COLS MAX_VAR+1

int zero_one(int A[MAX_ROWS][MAX_COLS], int *b, int *c, int n, int m,
             int *val, int *x)
{
  int exist;
  int alpha, beta, gamma, i, j, mnr, nr;
  int p, r, r1, r2, s, t, z;
  int y[MAX_ROWS], w[MAX_ROWS], zr[MAX_ROWS];
  int ii[MAX_COLS], jj[MAX_COLS], xx[MAX_COLS];
  int kk[MAX_COLS+1];

  for (i = 1; i <= m; i++) {
    y[i] = b[i];
  }
  z = 1;
  for (j = 1; j <= n; j++) {
    xx[j] = 0;
    z += c[j];
  }
  *val = z+z;
```



```c
  s = t = z = exist = 0;
  kk[1] = 0;
m10:
  p = mnr = 0;
  for (i = 1; i <= m; i++) {
    if ((r = y[i]) < 0) {
      p++;
      gamma = 0;
      alpha = r;
      beta = -INT_MAX;
      for (j = 1; j <= n; j++) {
        if (xx[j] <= 0) {
          if (c[j] + z >= *val) {
            xx[j] = 2;
            kk[s+1]++;
            jj[++t] = j;
          } else {
            if ((r1 = A[i][j]) < 0) {
              alpha -= r1;
              gamma += c[j];
              if (beta < r1) {
                beta = r1;
              }
            }
          }
        }
      }
      if (alpha < 0) {
        goto m20;
      }
      if (alpha + beta < 0) {
        if (gamma + z >= *val) {
          goto m20;
        }
        for (j = 1; j <= n; j++) {
          r1 = A[i][j];
          r2 = xx[j];
          if (r1 < 0) {
            if (!r2) {
              xx[j] = -2;
              for (nr = 1; nr <= mnr; nr++) {
                zr[nr] -= A[w[nr]][j];
                if (zr[nr] < 0) {
                  goto m20;
                }
              }
            }
          } else {
            if (r2 < 0) {
              alpha -= r1;
              if (alpha < 0) {
                goto m20;
              }
              gamma += c[j];
              if (gamma + z >= *val) {
                goto m20;
              }
            }
          }
        }
        mnr++;
        w[mnr] = i;
        zr[mnr] = alpha;
      }
    }
  }

  if (!p) {
    *val = z;
```

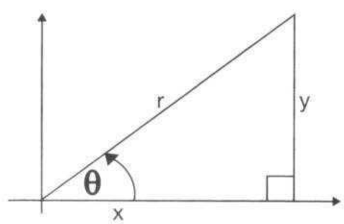
```
    exist = 1;
    for (j = 1; j <= n; j++) {
      x[j] = (xx[j] == 1) ? 1 : 0;
    }
    goto m20;
  }

  if (!mnr) {
    p = 0;
    gamma = -INT_MAX;
    for (j = 1; j <= n; j++) {
      if (!xx[j]) {
        beta = 0;
        for (i = 1; i <= m; i++) {
          r = y[i];
          r1 = A[i][j];
          if (r < r1) {
            beta += r - r1;
          }
        }
        r = c[j];
        if ((beta > gamma) ||
            (beta == gamma && r < alpha)) {
          alpha = r;
          gamma = beta;
          p = j;
        }
      }
    }
    if (!p) {
      goto m20;
    }
    s++;
    kk[s+1] = 0;
    jj[++t] = p;
    ii[s] = xx[p] = 1;
    z += c[p];
    for (i = 1; i <= m; i++) {
      y[i] -= A[i][p];
    }
  } else {
    s++;
    ii[s] = kk[s+1] = 0;
    for (j = 1; j <= n; j++) {
      if (xx[j] < 0) {
        jj[++t] = j;
        ii[s]--;
        z += c[j];
        xx[j] = 1;
        for (i = 1; i <= m; i++) {
          y[i] -= A[i][j];
        }
      }
    }
  }
  goto m10;

m20:
  for (j = 1; j <= n; j++) {
    if (xx[j] < 0) {
      xx[j] = 0;
    }
  }
  if (s > 0) {
    do {
      p = t;
      t -= kk[s+1];
      for (j = t+1; j <= p; j++) {
        xx[jj[j]] = 0;
```



```
      }
      p = (ii[s] >= 0) ? ii[s] : -ii[s];
      kk[s] += p;
      for (j = t-p+1; j <= t; j++) {
        p = jj[j];
        xx[p] = 2;
        z -= c[p];
        for (i = 1; i <= m; i++) {
          y[i] += A[i][p];
        }
      }
      s--;
      if (ii[s+1] >= 0) {
        goto m10;
      }
    } while (s);
  }

  return exist;
}

int main(void)
{
  int A[MAX_ROWS][MAX_COLS];
  int c[MAX_COLS], x[MAX_COLS], b[MAX_ROWS];
  int val, t;
  int m, n, i, j;

  while (scanf("%d %d", &n, &m) == 2 && n > 0 && m > 0) {
    /* read cost function */
    printf("Input cost function to minimize:\n");
    for (i = 1; i <= n; i++) {
      scanf("%d", &c[i]);
    }

    /* read constraints */
    for (i = 1; i <= m; i++) {
      printf("Input constraint #%d:\n", i);
      for (j = 1; j <= n; j++) {
        scanf("%d", &A[i][j]);
      }
      scanf("%d", &b[i]);
    }

    t = zero_one(A, b, c, n, m, &val, x);
    if (t) {
      printf("Minimum cost = %d\n", val);
      for (i = 1; i <= n; i++) {
        printf("x[%2d] = %2d\n", i, x[i]);
      }
    } else {
      printf("No solution exists.\n");
    }
  }

  return 0;
}
```

MATHEMATICS

Trigonometry Ratios:

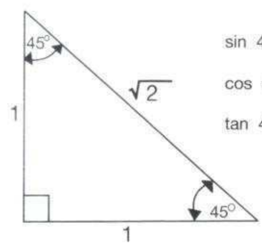


Memory Tip: SohCahToa

$\sin \theta = \frac{y}{r}$ (opp./hyp.) = $1 / \csc \theta$

$\cos \theta = \frac{x}{r}$ (adj./hyp.) = $1 / \sec \theta$

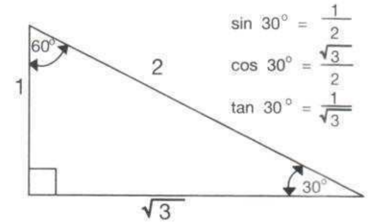
$\tan \theta = \frac{y}{x}$ (opp./adj.) = $1 / \cot \theta$



$\sin 45° = \frac{1}{\sqrt{2}}$

$\cos 45° = \frac{1}{\sqrt{2}}$

$\tan 45° = 1$

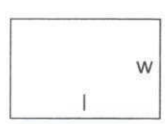


$\sin 30° = \frac{1}{2}$ $\quad \sin 60° = \frac{\sqrt{3}}{2}$

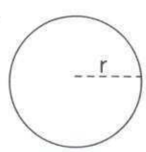
$\cos 30° = \frac{\sqrt{3}}{2}$ $\quad \cos 60° = \frac{1}{2}$

$\tan 30° = \frac{1}{\sqrt{3}}$ $\quad \tan 60° = \sqrt{3}$

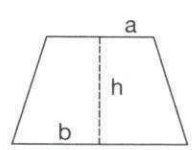
Geometry Formulas:



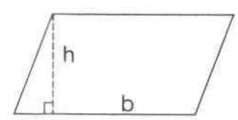
Rectangle
Perimeter = 2 (l+w)
Area = lw



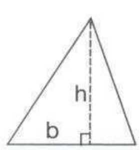
Circle
Circumference = $2\pi r$
Area = πr^2
r = radius



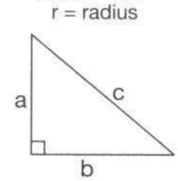
Trapezoid
Area = $\frac{1}{2}$ (a+b)h



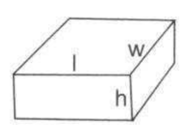
Parallelogram
Area = bh



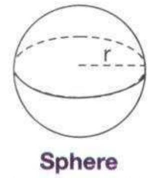
Triangle
Area = $\frac{bh}{2}$



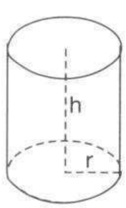
Right Triangle
(Pythagorean Theorem)
$c^2 = a^2 + b^2$



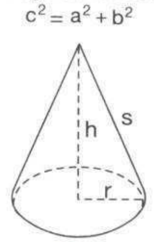
Rectangular Prism
Surface Area = 2lw+2wh+2lh
Volume = lwh



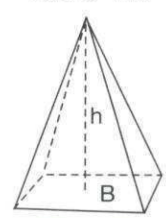
Sphere
Surface Area = $4 \pi r^2$
Volume = $\frac{4 \pi r^3}{3}$



Cylinder
Surface Area=$2 \pi rh + 2 \pi r^2$
Volume = $\pi r^2 h$



Cone
Surface Area = $\pi r^2 + \pi rs$
Volume = $\frac{\pi r^2 h}{3}$



Pyramid
Volume = $\frac{Bh}{3}$
(B = area of base)

Theoretical Computer Science Cheat Sheet

Definitions

$f(n) = O(g(n))$	iff $\exists$ positive c, n_0 such that $0 \le f(n) \le cg(n)\ \forall n \ge n_0$.
$f(n) = \Omega(g(n))$	iff $\exists$ positive c, n_0 such that $f(n) \ge cg(n) \ge 0\ \forall n \ge n_0$.
$f(n) = \Theta(g(n))$	iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
$f(n) = o(g(n))$	iff $\lim_{n\to\infty} f(n)/g(n) = 0$.
$\lim\limits_{n\to\infty} a_n = a$	iff $\forall \epsilon > 0$, $\exists n_0$ such that $\lvert a_n - a\rvert < \epsilon$, $\forall n \ge n_0$.
$\sup S$	least $b \in \mathbb{R}$ such that $b \ge s$, $\forall s \in S$.
$\inf S$	greatest $b \in \mathbb{R}$ such that $b \le s$, $\forall s \in S$.
$\liminf\limits_{n\to\infty} a_n$	$\lim\limits_{n\to\infty} \inf\{a_i \mid i \ge n, i \in \mathbb{N}\}$.
$\limsup\limits_{n\to\infty} a_n$	$\lim\limits_{n\to\infty} \sup\{a_i \mid i \ge n, i \in \mathbb{N}\}$.
$\binom{n}{k}$	Combinations: Size k subsets of a size n set.
$\begin{bmatrix} n\\k \end{bmatrix}$	Stirling numbers (1st kind): Arrangements of an n element set into k cycles.
$\begin{Bmatrix} n\\k \end{Bmatrix}$	Stirling numbers (2nd kind): Partitions of an n element set into k non-empty sets.
$\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle$	1st order Eulerian numbers: Permutations $\pi_1\pi_2\dots\pi_n$ on $\{1,2,\dots,n\}$ with k ascents.
$\left\langle\!\!\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\!\!\right\rangle$	2nd order Eulerian numbers.
C_n	Catalan Numbers: Binary trees with $n+1$ vertices.

Series

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \qquad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

In general:

$$\sum_{i=1}^{n} i^m = \frac{1}{m+1}\left[(n+1)^{m+1} - 1 - \sum_{i=1}^{n}\left((i+1)^{m+1} - i^{m+1} - (m+1)i^m\right)\right]$$

$$\sum_{i=1}^{n-1} i^m = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k n^{m+1-k}.$$

Geometric series:

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \ne 1, \qquad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}, \qquad \sum_{i=1}^{\infty} c^i = \frac{c}{1-c}, \quad |c| < 1,$$

$$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}, \quad c \ne 1, \qquad \sum_{i=0}^{\infty} ic^i = \frac{c}{(1-c)^2}, \quad |c| < 1.$$

Harmonic series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i}, \qquad \sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2}H_n - \frac{n(n-1)}{4}.$$

$$\sum_{i=1}^{n} H_i = (n+1)H_n - n, \qquad \sum_{i=1}^{n}\binom{i}{m}H_i = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right).$$

Identities

1. $\displaystyle \binom{n}{k} = \frac{n!}{(n-k)!k!}$, **2.** $\displaystyle \sum_{k=0}^{n}\binom{n}{k} = 2^n$, **3.** $\displaystyle \binom{n}{k} = \binom{n}{n-k}$,

4. $\displaystyle \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, **5.** $\displaystyle \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,

6. $\displaystyle \binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$, **7.** $\displaystyle \sum_{k=0}^{n}\binom{r+k}{k} = \binom{r+n+1}{n}$,

8. $\displaystyle \sum_{k=0}^{n}\binom{k}{m} = \binom{n+1}{m+1}$, **9.** $\displaystyle \sum_{k=0}^{n}\binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$,

10. $\displaystyle \binom{n}{k} = (-1)^k\binom{k-n-1}{k}$, **11.** $\displaystyle \begin{Bmatrix} n\\1 \end{Bmatrix} = \begin{Bmatrix} n\\n \end{Bmatrix} = 1$,

12. $\displaystyle \begin{Bmatrix} n\\2 \end{Bmatrix} = 2^{n-1} - 1$, **13.** $\displaystyle \begin{Bmatrix} n\\k \end{Bmatrix} = k\begin{Bmatrix} n-1\\k \end{Bmatrix} + \begin{Bmatrix} n-1\\k-1 \end{Bmatrix}$,

14. $\displaystyle \begin{bmatrix} n\\1 \end{bmatrix} = (n-1)!$, **15.** $\displaystyle \begin{bmatrix} n\\2 \end{bmatrix} = (n-1)!H_{n-1}$, **16.** $\displaystyle \begin{bmatrix} n\\n \end{bmatrix} = 1$, **17.** $\displaystyle \begin{bmatrix} n\\k \end{bmatrix} \ge \begin{Bmatrix} n\\k \end{Bmatrix}$,

18. $\displaystyle \begin{bmatrix} n\\k \end{bmatrix} = (n-1)\begin{bmatrix} n-1\\k \end{bmatrix} + \begin{bmatrix} n-1\\k-1 \end{bmatrix}$, **19.** $\displaystyle \begin{Bmatrix} n\\n-1 \end{Bmatrix} = \begin{bmatrix} n\\n-1 \end{bmatrix} = \binom{n}{2}$, **20.** $\displaystyle \sum_{k=0}^{n}\begin{bmatrix} n\\k \end{bmatrix} = n!$, **21.** $\displaystyle C_n = \frac{1}{n+1}\binom{2n}{n}$,

22. $\displaystyle \left\langle \begin{matrix} n\\0 \end{matrix} \right\rangle = \left\langle \begin{matrix} n\\n-1 \end{matrix} \right\rangle = 1$, **23.** $\displaystyle \left\langle \begin{matrix} n\\k \end{matrix} \right\rangle = \left\langle \begin{matrix} n\\n-1-k \end{matrix} \right\rangle$, **24.** $\displaystyle \left\langle \begin{matrix} n\\k \end{matrix} \right\rangle = (k+1)\left\langle \begin{matrix} n-1\\k \end{matrix} \right\rangle + (n-k)\left\langle \begin{matrix} n-1\\k-1 \end{matrix} \right\rangle$,

25. $\displaystyle \left\langle \begin{matrix} 0\\k \end{matrix} \right\rangle = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{otherwise} \end{cases}$ **26.** $\displaystyle \left\langle \begin{matrix} n\\1 \end{matrix} \right\rangle = 2^n - n - 1$, **27.** $\displaystyle \left\langle \begin{matrix} n\\2 \end{matrix} \right\rangle = 3^n - (n+1)2^n + \binom{n+1}{2}$,

28. $\displaystyle x^n = \sum_{k=0}^{n}\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\binom{x+k}{n}$, **29.** $\displaystyle \left\langle \begin{matrix} n\\m \end{matrix} \right\rangle = \sum_{k=0}^{m}\binom{n+1}{k}(m+1-k)^n(-1)^k$, **30.** $\displaystyle m!\begin{Bmatrix} n\\m \end{Bmatrix} = \sum_{k=0}^{n}\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\binom{k}{n-m}$,

31. $\displaystyle \left\langle \begin{matrix} n\\m \end{matrix} \right\rangle = \sum_{k=0}^{n}\begin{Bmatrix} n\\k \end{Bmatrix}\binom{n-k}{m}(-1)^{n-k-m}k!$, **32.** $\displaystyle \left\langle\!\!\left\langle \begin{matrix} n\\0 \end{matrix} \right\rangle\!\!\right\rangle = 1$, **33.** $\displaystyle \left\langle\!\!\left\langle \begin{matrix} n\\n \end{matrix} \right\rangle\!\!\right\rangle = 0 \quad \text{for } n \ne 0$,

34. $\displaystyle \left\langle\!\!\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\!\!\right\rangle = (k+1)\left\langle\!\!\left\langle \begin{matrix} n-1\\k \end{matrix} \right\rangle\!\!\right\rangle + (2n-1-k)\left\langle\!\!\left\langle \begin{matrix} n-1\\k-1 \end{matrix} \right\rangle\!\!\right\rangle$, **35.** $\displaystyle \sum_{k=0}^{n}\left\langle\!\!\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\!\!\right\rangle = \frac{(2n)^{\underline{n}}}{2^n}$,

36. $\displaystyle \begin{Bmatrix} x\\x-n \end{Bmatrix} = \sum_{k=0}^{n}\left\langle\!\!\left\langle \begin{matrix} n\\k \end{matrix} \right\rangle\!\!\right\rangle\binom{x+n-1-k}{2n}$, **37.** $\displaystyle \begin{Bmatrix} n+1\\m+1 \end{Bmatrix} = \sum_{k}\binom{n}{k}\begin{Bmatrix} k\\m \end{Bmatrix} = \sum_{k=0}^{n}\begin{Bmatrix} k\\m \end{Bmatrix}(m+1)^{n-k}$,

Theoretical Computer Science Cheat Sheet

Identities Cont.

38. $\left[{n+1 \atop m+1}\right] = \sum_k \left[{n \atop k}\right]\binom{k}{m} = \sum_{k=0}^{n} \left[{k \atop m}\right] n^{\underline{n-k}} = n! \sum_{k=0}^{n} \frac{1}{k!}\left[{k \atop m}\right]$,

39. $\left[{x \atop x-n}\right] = \sum_{k=0}^{n} \left\langle\!\!\left\langle{n \atop k}\right\rangle\!\!\right\rangle \binom{x+k}{2n}$,

40. $\left\{{n \atop m}\right\} = \sum_k \binom{n}{k}\left\{{k+1 \atop m+1}\right\}(-1)^{n-k}$,

41. $\left[{n \atop m}\right] = \sum_k \left[{n+1 \atop k+1}\right]\binom{k}{m}(-1)^{m-k}$,

42. $\left\{{m+n+1 \atop m}\right\} = \sum_{k=0}^{m} k\left\{{n+k \atop k}\right\}$,

43. $\left[{m+n+1 \atop m}\right] = \sum_{k=0}^{m} k(n+k)\left[{n+k \atop k}\right]$,

44. $\binom{n}{m} = \sum_k \left\{{n+1 \atop k+1}\right\}\left[{k \atop m}\right](-1)^{m-k}$,

45. $(n-m)!\binom{n}{m} = \sum_k \left[{n+1 \atop k+1}\right]\left\{{k \atop m}\right\}(-1)^{m-k}, \quad$ for $n \geq m$,

46. $\left\{{n \atop n-m}\right\} = \sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\left[{m+k \atop k}\right]$,

47. $\left[{n \atop n-m}\right] = \sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\left\{{m+k \atop k}\right\}$,

48. $\left\{{n \atop \ell+m}\right\}\binom{\ell+m}{\ell} = \sum_k \left\{{k \atop \ell}\right\}\left\{{n-k \atop m}\right\}\binom{n}{k}$,

49. $\left[{n \atop \ell+m}\right]\binom{\ell+m}{\ell} = \sum_k \left[{k \atop \ell}\right]\left[{n-k \atop m}\right]\binom{n}{k}$.

Trees

Every tree with n vertices has $n-1$ edges.

Kraft inequality: If the depths of the leaves of a binary tree are $d_1, \ldots, d_n$:
$$\sum_{i=1}^{n} 2^{-d_i} \leq 1,$$
and equality holds only if every internal node has 2 sons.

Recurrences

Master method:
$$T(n) = aT(n/b) + f(n), \quad a \geq 1, b > 1$$

If $\exists \epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$ then
$$T(n) = \Theta(n^{\log_b a}).$$

If $f(n) = \Theta(n^{\log_b a})$ then
$$T(n) = \Theta(n^{\log_b a} \log_2 n).$$

If $\exists \epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and $\exists c < 1$ such that $af(n/b) \leq cf(n)$ for large n, then
$$T(n) = \Theta(f(n)).$$

Substitution (example): Consider the following recurrence
$$T_{i+1} = 2^{2^i} \cdot T_i^2, \quad T_1 = 2.$$

Note that T_i is always a power of two. Let $t_i = \log_2 T_i$. Then we have
$$t_{i+1} = 2^i + 2t_i, \quad t_1 = 1.$$

Let $u_i = t_i/2^i$. Dividing both sides of the previous equation by 2^{i+1} we get
$$\frac{t_{i+1}}{2^{i+1}} = \frac{2^i}{2^{i+1}} + \frac{t_i}{2^i}.$$

Substituting we find
$$u_{i+1} = \tfrac{1}{2} + u_i, \qquad u_1 = \tfrac{1}{2},$$

which is simply $u_i = i/2$. So we find that T_i has the closed form $T_i = 2^{i2^{i-1}}$.

Summing factors (example): Consider the following recurrence
$$T(n) = 3T(n/2) + n, \quad T(1) = 1.$$

Rewrite so that all terms involving T are on the left side
$$T(n) - 3T(n/2) = n.$$

Now expand the recurrence, and choose a factor which makes the left side "telescope"

$$1\big(T(n) - 3T(n/2) = n\big)$$
$$3\big(T(n/2) - 3T(n/4) = n/2\big)$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$3^{\log_2 n - 1}\big(T(2) - 3T(1) = 2\big)$$

Let $m = \log_2 n$. Summing the left side we get $T(n) - 3^m T(1) = T(n) - 3^m = T(n) - n^k$ where $k = \log_2 3 \approx 1.58496$. Summing the right side we get
$$\sum_{i=0}^{m-1} \frac{n}{2^i} 3^i = n \sum_{i=0}^{m-1} \left(\tfrac{3}{2}\right)^i.$$

Let $c = \frac{3}{2}$. Then we have
$$\begin{aligned} n \sum_{i=0}^{m-1} c^i &= n \left(\frac{c^m - 1}{c - 1}\right) \\ &= 2n(c^{\log_2 n} - 1) \\ &= 2n(c^{(k-1)\log_c n} - 1) \\ &= 2n^k - 2n, \end{aligned}$$

and so $T(n) = 3n^k - 2n$. Full history recurrences can often be changed to limited history ones (example): Consider
$$T_i = 1 + \sum_{j=0}^{i-1} T_j, \quad T_0 = 1.$$

Note that
$$T_{i+1} = 1 + \sum_{j=0}^{i} T_j.$$

Subtracting we find
$$\begin{aligned} T_{i+1} - T_i &= 1 + \sum_{j=0}^{i} T_j - 1 - \sum_{j=0}^{i-1} T_j \\ &= T_i. \end{aligned}$$

And so $T_{i+1} = 2T_i = 2^{i+1}$.

Generating functions:

1. Multiply both sides of the equation by x^i.
2. Sum both sides over all i for which the equation is valid.
3. Choose a generating function $G(x)$. Usually $G(x) = \sum_{i=0}^{\infty} x^i g_i$.
3. Rewrite the equation in terms of the generating function $G(x)$.
4. Solve for $G(x)$.
5. The coefficient of x^i in $G(x)$ is g_i.

Example:
$$g_{i+1} = 2g_i + 1, \quad g_0 = 0.$$

Multiply and sum:
$$\sum_{i \geq 0} g_{i+1} x^i = \sum_{i \geq 0} 2g_i x^i + \sum_{i \geq 0} x^i.$$

We choose $G(x) = \sum_{i \geq 0} x^i g_i$. Rewrite in terms of $G(x)$:
$$\frac{G(x) - g_0}{x} = 2G(x) + \sum_{i \geq 0} x^i.$$

Simplify:
$$\frac{G(x)}{x} = 2G(x) + \frac{1}{1-x}.$$

Solve for $G(x)$:
$$G(x) = \frac{x}{(1-x)(1-2x)}.$$

Expand this using partial fractions:
$$\begin{aligned} G(x) &= x\left(\frac{2}{1-2x} - \frac{1}{1-x}\right) \\ &= x\left(2\sum_{i \geq 0} 2^i x^i - \sum_{i \geq 0} x^i\right) \\ &= \sum_{i \geq 0} (2^{i+1} - 1)x^{i+1}. \end{aligned}$$

So $g_i = 2^i - 1$.

Theoretical Computer Science Cheat Sheet

$\pi \approx 3.14159$, $e \approx 2.71828$, $\gamma \approx 0.57721$, $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$, $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -.61803$

i	2^i	p_i
1	2	2
2	4	3
3	8	5
4	16	7
5	32	11
6	64	13
7	128	17
8	256	19
9	512	23
10	1,024	29
11	2,048	31
12	4,096	37
13	8,192	41
14	16,384	43
15	32,768	47
16	65,536	53
17	131,072	59
18	262,144	61
19	524,288	67
20	1,048,576	71
21	2,097,152	73
22	4,194,304	79
23	8,388,608	83
24	16,777,216	89
25	33,554,432	97
26	67,108,864	101
27	134,217,728	103
28	268,435,456	107
29	536,870,912	109
30	1,073,741,824	113
31	2,147,483,648	127
32	4,294,967,296	131

Pascal's Triangle

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1

General

Bernoulli Numbers ($B_i = 0$, odd $i \neq 1$):

$$B_0 = 1,\ B_1 = -\tfrac{1}{2},\ B_2 = \tfrac{1}{6},\ B_4 = -\tfrac{1}{30},\ B_6 = \tfrac{1}{42},\ B_8 = -\tfrac{1}{30},\ B_{10} = \tfrac{5}{66}.$$

Change of base, quadratic formula:

$$\log_b x = \frac{\log_a x}{\log_a b}, \qquad \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Euler's number e:

$$e = 1 + \tfrac{1}{2} + \tfrac{1}{6} + \tfrac{1}{24} + \tfrac{1}{120} + \cdots$$

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$

$$\left(1 + \tfrac{1}{n}\right)^n < e < \left(1 + \tfrac{1}{n}\right)^{n+1}.$$

$$\left(1 + \tfrac{1}{n}\right)^n = e - \frac{e}{2n} + \frac{11e}{24n^2} - O\left(\frac{1}{n^3}\right).$$

Harmonic numbers:

$$1,\ \tfrac{3}{2},\ \tfrac{11}{6},\ \tfrac{25}{12},\ \tfrac{137}{60},\ \tfrac{49}{20},\ \tfrac{363}{140},\ \tfrac{761}{280},\ \tfrac{7129}{2520},\ldots$$

$$\ln n < H_n < \ln n + 1,$$

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right).$$

Factorial, Stirling's approximation:

1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

Ackermann's function and inverse:

$$a(i,j) = \begin{cases} 2^j & i = 1 \\ a(i-1, 2) & j = 1 \\ a(i-1, a(i, j-1)) & i, j \geq 2 \end{cases}$$

$$\alpha(i) = \min\{j \mid a(j,j) \geq i\}.$$

Binomial distribution:

$$\Pr[X = k] = \binom{n}{k} p^k q^{n-k}, \qquad q = 1 - p,$$

$$\mathrm{E}[X] = \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} = np.$$

Poisson distribution:

$$\Pr[X = k] = \frac{e^{-\lambda}\lambda^k}{k!}, \quad \mathrm{E}[X] = \lambda.$$

Normal (Gaussian) distribution:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \quad \mathrm{E}[X] = \mu.$$

The "coupon collector": We are given a random coupon each day, and there are n different types of coupons. The distribution of coupons is uniform. The expected number of days to pass before we to collect all n types is

$$nH_n.$$

Probability

Continuous distributions: If

$$\Pr[a < X < b] = \int_a^b p(x)\,dx,$$

then p is the probability density function of X. If

$$\Pr[X < a] = P(a),$$

then P is the distribution function of X. If P and p both exist then

$$P(a) = \int_{-\infty}^{a} p(x)\,dx.$$

Expectation: If X is discrete

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr[X = x].$$

If X continuous then

$$\mathrm{E}[g(X)] = \int_{-\infty}^{\infty} g(x)p(x)\,dx = \int_{-\infty}^{\infty} g(x)\,dP(x).$$

Variance, standard deviation:

$$\mathrm{VAR}[X] = \mathrm{E}[X^2] - \mathrm{E}[X]^2,$$

$$\sigma = \sqrt{\mathrm{VAR}[X]}.$$

For events A and B:

$$\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$$

$$\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B],$$

iff A and B are independent.

$$\Pr[A|B] = \frac{\Pr[A \wedge B]}{\Pr[B]}$$

For random variables X and Y:

$$\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y],$$

if X and Y are independent.

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y],$$

$$\mathrm{E}[cX] = c\,\mathrm{E}[X].$$

Bayes' theorem:

$$\Pr[A_i|B] = \frac{\Pr[B|A_i]\Pr[A_i]}{\sum_{j=1}^{n} \Pr[A_j]\Pr[B|A_j]}.$$

Inclusion-exclusion:

$$\Pr\left[\bigvee_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \Pr[X_i] + \sum_{k=2}^{n} (-1)^{k+1} \sum_{i_i < \cdots < i_k} \Pr\left[\bigwedge_{j=1}^{k} X_{i_j}\right].$$

Moment inequalities:

$$\Pr\left[|X| \geq \lambda\,\mathrm{E}[X]\right] \leq \frac{1}{\lambda},$$

$$\Pr\left[\left|X - \mathrm{E}[X]\right| \geq \lambda \cdot \sigma\right] \leq \frac{1}{\lambda^2}.$$

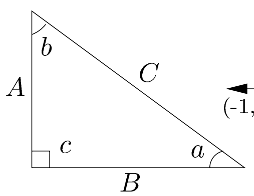
Geometric distribution:

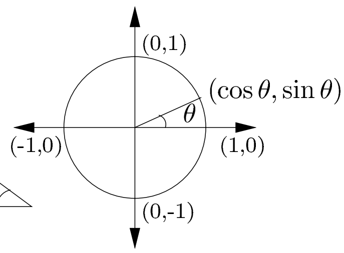
$$\Pr[X = k] = pq^{k-1}, \qquad q = 1 - p,$$

$$\mathrm{E}[X] = \sum_{k=1}^{\infty} kpq^{k-1} = \frac{1}{p}.$$

Theoretical Computer Science Cheat Sheet

Trigonometry





Pythagorean theorem:
$$C^2 = A^2 + B^2.$$

Definitions:
$$\sin a = A/C, \quad \cos a = B/C,$$
$$\csc a = C/A, \quad \sec a = C/B,$$
$$\tan a = \frac{\sin a}{\cos a} = \frac{A}{B}, \quad \cot a = \frac{\cos a}{\sin a} = \frac{B}{A}.$$

Area, radius of inscribed circle:
$$\tfrac{1}{2}AB, \quad \frac{AB}{A+B+C}.$$

Identities:
$$\sin x = \frac{1}{\csc x}, \qquad \cos x = \frac{1}{\sec x},$$
$$\tan x = \frac{1}{\cot x}, \qquad \sin^2 x + \cos^2 x = 1,$$
$$1 + \tan^2 x = \sec^2 x, \qquad 1 + \cot^2 x = \csc^2 x,$$
$$\sin x = \cos\left(\tfrac{\pi}{2} - x\right), \qquad \sin x = \sin(\pi - x),$$
$$\cos x = -\cos(\pi - x), \qquad \tan x = \cot\left(\tfrac{\pi}{2} - x\right),$$
$$\cot x = -\cot(\pi - x), \qquad \csc x = \cot\tfrac{x}{2} - \cot x,$$
$$\sin(x \pm y) = \sin x \cos y \pm \cos x \sin y,$$
$$\cos(x \pm y) = \cos x \cos y \mp \sin x \sin y,$$
$$\tan(x \pm y) = \frac{\tan x \pm \tan y}{1 \mp \tan x \tan y},$$
$$\cot(x \pm y) = \frac{\cot x \cot y \mp 1}{\cot x \pm \cot y},$$
$$\sin 2x = 2 \sin x \cos x, \qquad \sin 2x = \frac{2\tan x}{1 + \tan^2 x},$$
$$\cos 2x = \cos^2 x - \sin^2 x, \qquad \cos 2x = 2\cos^2 x - 1,$$
$$\cos 2x = 1 - 2\sin^2 x, \qquad \cos 2x = \frac{1 - \tan^2 x}{1 + \tan^2 x},$$
$$\tan 2x = \frac{2\tan x}{1 - \tan^2 x}, \qquad \cot 2x = \frac{\cot^2 x - 1}{2\cot x},$$
$$\sin(x+y)\sin(x-y) = \sin^2 x - \sin^2 y,$$
$$\cos(x+y)\cos(x-y) = \cos^2 x - \sin^2 y.$$

Euler's equation:
$$e^{ix} = \cos x + i \sin x, \qquad e^{i\pi} = -1.$$

v2.02 ©1994 by Steve Seiden
sseiden@acm.org
http://www.csc.lsu.edu/~seiden

Matrices

Multiplication:
$$C = A \cdot B, \quad c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.$$

Determinants: $\det A \neq 0$ iff A is non-singular.
$$\det A \cdot B = \det A \cdot \det B,$$
$$\det A = \sum_{\pi} \prod_{i=1}^{n} \operatorname{sign}(\pi) a_{i,\pi(i)}.$$

2×2 and 3×3 determinant:
$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc,$$
$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = g\begin{vmatrix} b & c \\ e & f \end{vmatrix} - h\begin{vmatrix} a & c \\ d & f \end{vmatrix} + i\begin{vmatrix} a & b \\ d & e \end{vmatrix}$$
$$= \begin{matrix} aei + bfg + cdh \\ -ceg - fha - ibd. \end{matrix}$$

Permanents:
$$\operatorname{perm} A = \sum_{\pi} \prod_{i=1}^{n} a_{i,\pi(i)}.$$

Hyperbolic Functions

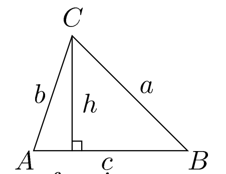
Definitions:
$$\sinh x = \frac{e^x - e^{-x}}{2}, \qquad \cosh x = \frac{e^x + e^{-x}}{2},$$
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \qquad \operatorname{csch} x = \frac{1}{\sinh x},$$
$$\operatorname{sech} x = \frac{1}{\cosh x}, \qquad \coth x = \frac{1}{\tanh x}.$$

Identities:
$$\cosh^2 x - \sinh^2 x = 1, \qquad \tanh^2 x + \operatorname{sech}^2 x = 1,$$
$$\coth^2 x - \operatorname{csch}^2 x = 1, \qquad \sinh(-x) = -\sinh x,$$
$$\cosh(-x) = \cosh x, \qquad \tanh(-x) = -\tanh x,$$
$$\sinh(x+y) = \sinh x \cosh y + \cosh x \sinh y,$$
$$\cosh(x+y) = \cosh x \cosh y + \sinh x \sinh y,$$
$$\sinh 2x = 2 \sinh x \cosh x,$$
$$\cosh 2x = \cosh^2 x + \sinh^2 x,$$
$$\cosh x + \sinh x = e^x, \qquad \cosh x - \sinh x = e^{-x},$$
$$(\cosh x + \sinh x)^n = \cosh nx + \sinh nx, \quad n \in \mathbb{Z},$$
$$2\sinh^2 \tfrac{x}{2} = \cosh x - 1, \qquad 2\cosh^2 \tfrac{x}{2} = \cosh x + 1.$$

θ	$\sin\theta$	$\cos\theta$	$\tan\theta$
0	0	1	0
$\frac{\pi}{6}$	$\frac{1}{2}$	$\frac{\sqrt{3}}{2}$	$\frac{\sqrt{3}}{3}$
$\frac{\pi}{4}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{2}}{2}$	1
$\frac{\pi}{3}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	$\sqrt{3}$
$\frac{\pi}{2}$	1	0	∞

… in mathematics you don't understand things, you just get used to them.
– J. von Neumann

More Trig.



Law of cosines:
$$c^2 = a^2 + b^2 - 2ab\cos C.$$

Area:
$$A = \tfrac{1}{2}hc,$$
$$= \tfrac{1}{2}ab\sin C,$$
$$= \frac{c^2 \sin A \sin B}{2\sin C}.$$

Heron's formula:
$$A = \sqrt{s \cdot s_a \cdot s_b \cdot s_c},$$
$$s = \tfrac{1}{2}(a+b+c),$$
$$s_a = s - a,$$
$$s_b = s - b,$$
$$s_c = s - c.$$

More identities:
$$\sin\tfrac{x}{2} = \sqrt{\frac{1 - \cos x}{2}},$$
$$\cos\tfrac{x}{2} = \sqrt{\frac{1 + \cos x}{2}},$$
$$\tan\tfrac{x}{2} = \sqrt{\frac{1 - \cos x}{1 + \cos x}},$$
$$= \frac{1 - \cos x}{\sin x},$$
$$= \frac{\sin x}{1 + \cos x},$$
$$\cot\tfrac{x}{2} = \sqrt{\frac{1 + \cos x}{1 - \cos x}},$$
$$= \frac{1 + \cos x}{\sin x},$$
$$= \frac{\sin x}{1 - \cos x},$$
$$\sin x = \frac{e^{ix} - e^{-ix}}{2i},$$
$$\cos x = \frac{e^{ix} + e^{-ix}}{2},$$
$$\tan x = -i\frac{e^{ix} - e^{-ix}}{e^{ix} + e^{-ix}},$$
$$= -i\frac{e^{2ix} - 1}{e^{2ix} + 1},$$
$$\sin x = \frac{\sinh ix}{i},$$
$$\cos x = \cosh ix,$$
$$\tan x = \frac{\tanh ix}{i}.$$

Theoretical Computer Science Cheat Sheet

Number Theory

The Chinese remainder theorem: There exists a number C such that:

$$C \equiv r_1 \bmod m_1$$
$$\vdots \quad \vdots \quad \vdots$$
$$C \equiv r_n \bmod m_n$$

if m_i and m_j are relatively prime for $i \neq j$.

Euler's function: $\phi(x)$ is the number of positive integers less than x relatively prime to x. If $\prod_{i=1}^n p_i^{e_i}$ is the prime factorization of x then

$$\phi(x) = \prod_{i=1}^n p_i^{e_i-1}(p_i - 1).$$

Euler's theorem: If a and b are relatively prime then

$$1 \equiv a^{\phi(b)} \bmod b.$$

Fermat's theorem:

$$1 \equiv a^{p-1} \bmod p.$$

The Euclidean algorithm: if $a > b$ are integers then

$$\gcd(a, b) = \gcd(a \bmod b, b).$$

If $\prod_{i=1}^n p_i^{e_i}$ is the prime factorization of x then

$$S(x) = \sum_{d|x} d = \prod_{i=1}^n \frac{p_i^{e_i+1} - 1}{p_i - 1}.$$

Perfect Numbers: x is an even perfect number iff $x = 2^{n-1}(2^n - 1)$ and $2^n - 1$ is prime.

Wilson's theorem: n is a prime iff

$$(n-1)! \equiv -1 \bmod n.$$

Möbius inversion:

$$\mu(i) = \begin{cases} 1 & \text{if } i = 1. \\ 0 & \text{if } i \text{ is not square-free.} \\ (-1)^r & \text{if } i \text{ is the product of} \\ & r \text{ distinct primes.} \end{cases}$$

If

$$G(a) = \sum_{d|a} F(d),$$

then

$$F(a) = \sum_{d|a} \mu(d) G\Big(\frac{a}{d}\Big).$$

Prime numbers:

$$p_n = n \ln n + n \ln\ln n - n + n \frac{\ln\ln n}{\ln n} + O\Big(\frac{n}{\ln n}\Big),$$

$$\pi(n) = \frac{n}{\ln n} + \frac{n}{(\ln n)^2} + \frac{2!n}{(\ln n)^3} + O\Big(\frac{n}{(\ln n)^4}\Big).$$

Graph Theory

Definitions:

Loop	An edge connecting a vertex to itself.
Directed	Each edge has a direction.
Simple	Graph with no loops or multi-edges.
Walk	A sequence $v_0 e_1 v_1 \ldots e_\ell v_\ell$.
Trail	A walk with distinct edges.
Path	A trail with distinct vertices.
Connected	A graph where there exists a path between any two vertices.
Component	A maximal connected subgraph.
Tree	A connected acyclic graph.
Free tree	A tree with no root.
DAG	Directed acyclic graph.
Eulerian	Graph with a trail visiting each edge exactly once.
Hamiltonian	Graph with a cycle visiting each vertex exactly once.
Cut	A set of edges whose removal increases the number of components.
Cut-set	A minimal cut.
Cut edge	A size 1 cut.
k-Connected	A graph connected with the removal of any $k-1$ vertices.
k-Tough	$\forall S \subseteq V, S \neq \emptyset$ we have $k \cdot c(G - S) \leq \lvert S \rvert$.
k-Regular	A graph where all vertices have degree k.
k-Factor	A k-regular spanning subgraph.
Matching	A set of edges, no two of which are adjacent.
Clique	A set of vertices, all of which are adjacent.
Ind. set	A set of vertices, none of which are adjacent.
Vertex cover	A set of vertices which cover all edges.
Planar graph	A graph which can be embeded in the plane.
Plane graph	An embedding of a planar graph.

$$\sum_{v \in V} \deg(v) = 2m.$$

If G is planar then $n - m + f = 2$, so

$$f \leq 2n - 4, \quad m \leq 3n - 6.$$

Any planar graph has a vertex with degree ≤ 5.

Notation:

$E(G)$	Edge set
$V(G)$	Vertex set
$c(G)$	Number of components
$G[S]$	Induced subgraph
$\deg(v)$	Degree of v
$\Delta(G)$	Maximum degree
$\delta(G)$	Minimum degree
$\chi(G)$	Chromatic number
$\chi_E(G)$	Edge chromatic number
G^c	Complement graph
K_n	Complete graph
K_{n_1,n_2}	Complete bipartite graph
$\mathrm{r}(k,\ell)$	Ramsey number

Geometry

Projective coordinates: triples (x, y, z), not all x, y and z zero.

$$(x, y, z) = (cx, cy, cz) \quad \forall c \neq 0.$$

Cartesian	Projective
(x, y)	$(x, y, 1)$
$y = mx + b$	$(m, -1, b)$
$x = c$	$(1, 0, -c)$

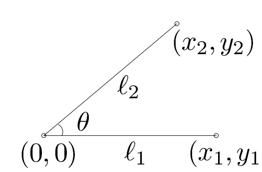
Distance formula, L_p and L_∞ metric:

$$\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2},$$
$$\big[|x_1 - x_0|^p + |y_1 - y_0|^p\big]^{1/p},$$
$$\lim_{p \to \infty} \big[|x_1 - x_0|^p + |y_1 - y_0|^p\big]^{1/p}.$$

Area of triangle (x_0, y_0), (x_1, y_1) and (x_2, y_2):

$$\tfrac{1}{2} \operatorname{abs} \begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix}.$$

Angle formed by three points:



$$\cos\theta = \frac{(x_1, y_1) \cdot (x_2, y_2)}{\ell_1 \ell_2}.$$

Line through two points (x_0, y_0) and (x_1, y_1):

$$\begin{vmatrix} x & y & 1 \\ x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \end{vmatrix} = 0.$$

Area of circle, volume of sphere:

$$A = \pi r^2, \qquad V = \tfrac{4}{3}\pi r^3.$$

If I have seen farther than others, it is because I have stood on the shoulders of giants.
– Issac Newton

π

Wallis' identity:
$$\pi = 2 \cdot \frac{2 \cdot 2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdots}{1 \cdot 3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdots}$$

Brouncker's continued fraction expansion:
$$\tfrac{\pi}{4} = 1 + \cfrac{1^2}{2 + \cfrac{3^2}{2 + \cfrac{5^2}{2 + \cfrac{7^2}{2 + \cdots}}}}$$

Gregrory's series:
$$\tfrac{\pi}{4} = 1 - \tfrac{1}{3} + \tfrac{1}{5} - \tfrac{1}{7} + \tfrac{1}{9} - \cdots$$

Newton's series:
$$\tfrac{\pi}{6} = \frac{1}{2} + \frac{1}{2 \cdot 3 \cdot 2^3} + \frac{1 \cdot 3}{2 \cdot 4 \cdot 5 \cdot 2^5} + \cdots$$

Sharp's series:
$$\tfrac{\pi}{6} = \frac{1}{\sqrt{3}}\left(1 - \frac{1}{3^1 \cdot 3} + \frac{1}{3^2 \cdot 5} - \frac{1}{3^3 \cdot 7} + \cdots\right)$$

Euler's series:
$$\tfrac{\pi^2}{6} = \tfrac{1}{1^2} + \tfrac{1}{2^2} + \tfrac{1}{3^2} + \tfrac{1}{4^2} + \tfrac{1}{5^2} + \cdots$$
$$\tfrac{\pi^2}{8} = \tfrac{1}{1^2} + \tfrac{1}{3^2} + \tfrac{1}{5^2} + \tfrac{1}{7^2} + \tfrac{1}{9^2} + \cdots$$
$$\tfrac{\pi^2}{12} = \tfrac{1}{1^2} - \tfrac{1}{2^2} + \tfrac{1}{3^2} - \tfrac{1}{4^2} + \tfrac{1}{5^2} - \cdots$$

Partial Fractions

Let $N(x)$ and $D(x)$ be polynomial functions of x. We can break down $N(x)/D(x)$ using partial fraction expansion. First, if the degree of N is greater than or equal to the degree of D, divide N by D, obtaining
$$\frac{N(x)}{D(x)} = Q(x) + \frac{N'(x)}{D(x)},$$
where the degree of N' is less than that of D. Second, factor $D(x)$. Use the following rules: For a non-repeated factor:
$$\frac{N(x)}{(x-a)D(x)} = \frac{A}{x-a} + \frac{N'(x)}{D(x)},$$
where
$$A = \left[\frac{N(x)}{D(x)}\right]_{x=a}.$$
For a repeated factor:
$$\frac{N(x)}{(x-a)^m D(x)} = \sum_{k=0}^{m-1} \frac{A_k}{(x-a)^{m-k}} + \frac{N'(x)}{D(x)},$$
where
$$A_k = \frac{1}{k!}\left[\frac{d^k}{dx^k}\left(\frac{N(x)}{D(x)}\right)\right]_{x=a}.$$

The reasonable man adapts himself to the world; the unreasonable persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable.
– George Bernard Shaw

Calculus

Derivatives:

1. $\dfrac{d(cu)}{dx} = c\dfrac{du}{dx}$, **2.** $\dfrac{d(u+v)}{dx} = \dfrac{du}{dx} + \dfrac{dv}{dx}$, **3.** $\dfrac{d(uv)}{dx} = u\dfrac{dv}{dx} + v\dfrac{du}{dx}$,

4. $\dfrac{d(u^n)}{dx} = nu^{n-1}\dfrac{du}{dx}$, **5.** $\dfrac{d(u/v)}{dx} = \dfrac{v\left(\frac{du}{dx}\right) - u\left(\frac{dv}{dx}\right)}{v^2}$, **6.** $\dfrac{d(e^{cu})}{dx} = ce^{cu}\dfrac{du}{dx}$,

7. $\dfrac{d(c^u)}{dx} = (\ln c)c^u\dfrac{du}{dx}$, **8.** $\dfrac{d(\ln u)}{dx} = \dfrac{1}{u}\dfrac{du}{dx}$,

9. $\dfrac{d(\sin u)}{dx} = \cos u\dfrac{du}{dx}$, **10.** $\dfrac{d(\cos u)}{dx} = -\sin u\dfrac{du}{dx}$,

11. $\dfrac{d(\tan u)}{dx} = \sec^2 u\dfrac{du}{dx}$, **12.** $\dfrac{d(\cot u)}{dx} = \csc^2 u\dfrac{du}{dx}$,

13. $\dfrac{d(\sec u)}{dx} = \tan u \sec u\dfrac{du}{dx}$, **14.** $\dfrac{d(\csc u)}{dx} = -\cot u \csc u\dfrac{du}{dx}$,

15. $\dfrac{d(\arcsin u)}{dx} = \dfrac{1}{\sqrt{1-u^2}}\dfrac{du}{dx}$, **16.** $\dfrac{d(\arccos u)}{dx} = \dfrac{-1}{\sqrt{1-u^2}}\dfrac{du}{dx}$,

17. $\dfrac{d(\arctan u)}{dx} = \dfrac{1}{1+u^2}\dfrac{du}{dx}$, **18.** $\dfrac{d(\operatorname{arccot} u)}{dx} = \dfrac{-1}{1+u^2}\dfrac{du}{dx}$,

19. $\dfrac{d(\operatorname{arcsec} u)}{dx} = \dfrac{1}{u\sqrt{1-u^2}}\dfrac{du}{dx}$, **20.** $\dfrac{d(\operatorname{arccsc} u)}{dx} = \dfrac{-1}{u\sqrt{1-u^2}}\dfrac{du}{dx}$,

21. $\dfrac{d(\sinh u)}{dx} = \cosh u\dfrac{du}{dx}$, **22.** $\dfrac{d(\cosh u)}{dx} = \sinh u\dfrac{du}{dx}$,

23. $\dfrac{d(\tanh u)}{dx} = \operatorname{sech}^2 u\dfrac{du}{dx}$, **24.** $\dfrac{d(\coth u)}{dx} = -\operatorname{csch}^2 u\dfrac{du}{dx}$,

25. $\dfrac{d(\operatorname{sech} u)}{dx} = -\operatorname{sech} u \tanh u\dfrac{du}{dx}$, **26.** $\dfrac{d(\operatorname{csch} u)}{dx} = -\operatorname{csch} u \coth u\dfrac{du}{dx}$,

27. $\dfrac{d(\operatorname{arcsinh} u)}{dx} = \dfrac{1}{\sqrt{1+u^2}}\dfrac{du}{dx}$, **28.** $\dfrac{d(\operatorname{arccosh} u)}{dx} = \dfrac{1}{\sqrt{u^2-1}}\dfrac{du}{dx}$,

29. $\dfrac{d(\operatorname{arctanh} u)}{dx} = \dfrac{1}{1-u^2}\dfrac{du}{dx}$, **30.** $\dfrac{d(\operatorname{arccoth} u)}{dx} = \dfrac{1}{u^2-1}\dfrac{du}{dx}$,

31. $\dfrac{d(\operatorname{arcsech} u)}{dx} = \dfrac{-1}{u\sqrt{1-u^2}}\dfrac{du}{dx}$, **32.** $\dfrac{d(\operatorname{arccsch} u)}{dx} = \dfrac{-1}{|u|\sqrt{1+u^2}}\dfrac{du}{dx}$.

Integrals:

1. $\displaystyle\int cu\,dx = c\int u\,dx$, **2.** $\displaystyle\int (u+v)\,dx = \int u\,dx + \int v\,dx$,

3. $\displaystyle\int x^n\,dx = \frac{1}{n+1}x^{n+1}, \quad n \neq -1$, **4.** $\displaystyle\int \frac{1}{x}dx = \ln x$, **5.** $\displaystyle\int e^x\,dx = e^x$,

6. $\displaystyle\int \frac{dx}{1+x^2} = \arctan x$, **7.** $\displaystyle\int u\frac{dv}{dx}dx = uv - \int v\frac{du}{dx}dx$,

8. $\displaystyle\int \sin x\,dx = -\cos x$, **9.** $\displaystyle\int \cos x\,dx = \sin x$,

10. $\displaystyle\int \tan x\,dx = -\ln|\cos x|$, **11.** $\displaystyle\int \cot x\,dx = \ln|\cos x|$,

12. $\displaystyle\int \sec x\,dx = \ln|\sec x + \tan x|$, **13.** $\displaystyle\int \csc x\,dx = \ln|\csc x + \cot x|$,

14. $\displaystyle\int \arcsin \tfrac{x}{a} dx = \arcsin \tfrac{x}{a} + \sqrt{a^2 - x^2}, \quad a > 0$,

Calculus Cont.

15. $\int \arccos\frac{x}{a}dx = \arccos\frac{x}{a} - \sqrt{a^2-x^2}, \quad a>0,$

16. $\int \arctan\frac{x}{a}dx = x\arctan\frac{x}{a} - \frac{a}{2}\ln(a^2+x^2), \quad a>0,$

17. $\int \sin^2(ax)dx = \frac{1}{2a}\big(ax - \sin(ax)\cos(ax)\big),$

18. $\int \cos^2(ax)dx = \frac{1}{2a}\big(ax + \sin(ax)\cos(ax)\big),$

19. $\int \sec^2 x\,dx = \tan x,$

20. $\int \csc^2 x\,dx = -\cot x,$

21. $\int \sin^n x\,dx = -\frac{\sin^{n-1}x\cos x}{n} + \frac{n-1}{n}\int \sin^{n-2}x\,dx,$

22. $\int \cos^n x\,dx = \frac{\cos^{n-1}x\sin x}{n} + \frac{n-1}{n}\int \cos^{n-2}x\,dx,$

23. $\int \tan^n x\,dx = \frac{\tan^{n-1}x}{n-1} - \int \tan^{n-2}x\,dx, \quad n\neq 1,$

24. $\int \cot^n x\,dx = -\frac{\cot^{n-1}x}{n-1} - \int \cot^{n-2}x\,dx, \quad n\neq 1,$

25. $\int \sec^n x\,dx = \frac{\tan x\sec^{n-1}x}{n-1} + \frac{n-2}{n-1}\int \sec^{n-2}x\,dx, \quad n\neq 1,$

26. $\int \csc^n x\,dx = -\frac{\cot x\csc^{n-1}x}{n-1} + \frac{n-2}{n-1}\int \csc^{n-2}x\,dx, \quad n\neq 1,$

27. $\int \sinh x\,dx = \cosh x,$

28. $\int \cosh x\,dx = \sinh x,$

29. $\int \tanh x\,dx = \ln|\cosh x|,$

30. $\int \coth x\,dx = \ln|\sinh x|,$

31. $\int \operatorname{sech} x\,dx = \arctan\sinh x,$

32. $\int \operatorname{csch} x\,dx = \ln\left|\tanh\frac{x}{2}\right|,$

33. $\int \sinh^2 x\,dx = \frac{1}{4}\sinh(2x) - \frac{1}{2}x,$

34. $\int \cosh^2 x\,dx = \frac{1}{4}\sinh(2x) + \frac{1}{2}x,$

35. $\int \operatorname{sech}^2 x\,dx = \tanh x,$

36. $\int \operatorname{arcsinh}\frac{x}{a}dx = x\operatorname{arcsinh}\frac{x}{a} - \sqrt{x^2+a^2}, \quad a>0,$

37. $\int \operatorname{arctanh}\frac{x}{a}dx = x\operatorname{arctanh}\frac{x}{a} + \frac{a}{2}\ln|a^2-x^2|,$

38. $$\int \operatorname{arccosh}\frac{x}{a}dx = \begin{cases} x\operatorname{arccosh}\dfrac{x}{a} - \sqrt{x^2+a^2}, & \text{if } \operatorname{arccosh}\frac{x}{a}>0 \text{ and } a>0,\\ x\operatorname{arccosh}\dfrac{x}{a} + \sqrt{x^2+a^2}, & \text{if } \operatorname{arccosh}\frac{x}{a}<0 \text{ and } a>0,\end{cases}$$

39. $\int \frac{dx}{\sqrt{a^2+x^2}} = \ln\left(x+\sqrt{a^2+x^2}\right), \quad a>0,$

40. $\int \frac{dx}{a^2+x^2} = \frac{1}{a}\arctan\frac{x}{a}, \quad a>0,$

41. $\int \sqrt{a^2-x^2}\,dx = \frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin\frac{x}{a}, \quad a>0,$

42. $\int (a^2-x^2)^{3/2}dx = \frac{x}{8}(5a^2-2x^2)\sqrt{a^2-x^2} + \frac{3a^4}{8}\arcsin\frac{x}{a}, \quad a>0,$

43. $\int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin\frac{x}{a}, \quad a>0,$

44. $\int \frac{dx}{a^2-x^2} = \frac{1}{2a}\ln\left|\frac{a+x}{a-x}\right|,$

45. $\int \frac{dx}{(a^2-x^2)^{3/2}} = \frac{x}{a^2\sqrt{a^2-x^2}},$

46. $\int \sqrt{a^2\pm x^2}\,dx = \frac{x}{2}\sqrt{a^2\pm x^2} \pm \frac{a^2}{2}\ln\left|x+\sqrt{a^2\pm x^2}\right|,$

47. $\int \frac{dx}{\sqrt{x^2-a^2}} = \ln\left|x+\sqrt{x^2-a^2}\right|, \quad a>0,$

48. $\int \frac{dx}{ax^2+bx} = \frac{1}{a}\ln\left|\frac{x}{a+bx}\right|,$

49. $\int x\sqrt{a+bx}\,dx = \frac{2(3bx-2a)(a+bx)^{3/2}}{15b^2},$

50. $\int \frac{\sqrt{a+bx}}{x}dx = 2\sqrt{a+bx} + a\int \frac{1}{x\sqrt{a+bx}}dx,$

51. $\int \frac{x}{\sqrt{a+bx}}dx = \frac{1}{\sqrt{2}}\ln\left|\frac{\sqrt{a+bx}-\sqrt{a}}{\sqrt{a+bx}+\sqrt{a}}\right|, \quad a>0,$

52. $\int \frac{\sqrt{a^2-x^2}}{x}dx = \sqrt{a^2-x^2} - a\ln\left|\frac{a+\sqrt{a^2-x^2}}{x}\right|,$

53. $\int x\sqrt{a^2-x^2}\,dx = -\frac{1}{3}(a^2-x^2)^{3/2},$

54. $\int x^2\sqrt{a^2-x^2}\,dx = \frac{x}{8}(2x^2-a^2)\sqrt{a^2-x^2} + \frac{a^4}{8}\arcsin\frac{x}{a}, \quad a>0,$

55. $\int \frac{dx}{\sqrt{a^2-x^2}} = -\frac{1}{a}\ln\left|\frac{a+\sqrt{a^2-x^2}}{x}\right|,$

56. $\int \frac{x\,dx}{\sqrt{a^2-x^2}} = -\sqrt{a^2-x^2},$

57. $\int \frac{x^2\,dx}{\sqrt{a^2-x^2}} = -\frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin\frac{x}{a}, \quad a>0,$

58. $\int \frac{\sqrt{a^2+x^2}}{x}dx = \sqrt{a^2+x^2} - a\ln\left|\frac{a+\sqrt{a^2+x^2}}{x}\right|,$

59. $\int \frac{\sqrt{x^2-a^2}}{x}dx = \sqrt{x^2-a^2} - a\arccos\frac{a}{|x|}, \quad a>0,$

60. $\int x\sqrt{x^2\pm a^2}\,dx = \frac{1}{3}(x^2\pm a^2)^{3/2},$

61. $\int \frac{dx}{x\sqrt{x^2+a^2}} = \frac{1}{a}\ln\left|\frac{x}{a+\sqrt{a^2+x^2}}\right|,$

Theoretical Computer Science Cheat Sheet

Calculus Cont.

62. $\displaystyle\int \frac{dx}{x\sqrt{x^2-a^2}} = \tfrac{1}{a}\arccos\tfrac{a}{|x|}, \quad a>0,$

63. $\displaystyle\int \frac{dx}{x^2\sqrt{x^2\pm a^2}} = \mp\frac{\sqrt{x^2\pm a^2}}{a^2x},$

64. $\displaystyle\int \frac{x\,dx}{\sqrt{x^2\pm a^2}} = \sqrt{x^2\pm a^2},$

65. $\displaystyle\int \frac{\sqrt{x^2\pm a^2}}{x^4}\,dx = \mp\frac{(x^2+a^2)^{3/2}}{3a^2x^3},$

66. $$\int \frac{dx}{ax^2+bx+c} = \begin{cases} \dfrac{1}{\sqrt{b^2-4ac}}\ln\left|\dfrac{2ax+b-\sqrt{b^2-4ac}}{2ax+b+\sqrt{b^2-4ac}}\right|, & \text{if } b^2>4ac,\\ \dfrac{2}{\sqrt{4ac-b^2}}\arctan\dfrac{2ax+b}{\sqrt{4ac-b^2}}, & \text{if } b^2<4ac, \end{cases}$$

67. $$\int \frac{dx}{\sqrt{ax^2+bx+c}} = \begin{cases} \dfrac{1}{\sqrt{a}}\ln\left|2ax+b+2\sqrt{a}\sqrt{ax^2+bx+c}\right|, & \text{if } a>0,\\ \dfrac{1}{\sqrt{-a}}\arcsin\dfrac{-2ax-b}{\sqrt{b^2-4ac}}, & \text{if } a<0, \end{cases}$$

68. $$\int \sqrt{ax^2+bx+c}\,dx = \frac{2ax+b}{4a}\sqrt{ax^2+bx+c} + \frac{4ax-b^2}{8a}\int\frac{dx}{\sqrt{ax^2+bx+c}},$$

69. $$\int \frac{x\,dx}{\sqrt{ax^2+bx+c}} = \frac{\sqrt{ax^2+bx+c}}{a} - \frac{b}{2a}\int\frac{dx}{\sqrt{ax^2+bx+c}},$$

70. $$\int \frac{dx}{x\sqrt{ax^2+bx+c}} = \begin{cases} \dfrac{-1}{\sqrt{c}}\ln\left|\dfrac{2\sqrt{c}\sqrt{ax^2+bx+c}+bx+2c}{x}\right|, & \text{if } c>0,\\ \dfrac{1}{\sqrt{-c}}\arcsin\dfrac{bx+2c}{|x|\sqrt{b^2-4ac}}, & \text{if } c<0, \end{cases}$$

71. $\displaystyle\int x^3\sqrt{x^2+a^2}\,dx = (\tfrac{1}{3}x^2-\tfrac{2}{15}a^2)(x^2+a^2)^{3/2},$

72. $\displaystyle\int x^n\sin(ax)\,dx = -\tfrac{1}{a}x^n\cos(ax)+\tfrac{n}{a}\int x^{n-1}\cos(ax)\,dx,$

73. $\displaystyle\int x^n\cos(ax)\,dx = \tfrac{1}{a}x^n\sin(ax)-\tfrac{n}{a}\int x^{n-1}\sin(ax)\,dx,$

74. $\displaystyle\int x^ne^{ax}\,dx = \frac{x^ne^{ax}}{a}-\tfrac{n}{a}\int x^{n-1}e^{ax}\,dx,$

75. $\displaystyle\int x^n\ln(ax)\,dx = x^{n+1}\left(\frac{\ln(ax)}{n+1}-\frac{1}{(n+1)^2}\right),$

76. $\displaystyle\int x^n(\ln ax)^m\,dx = \frac{x^{n+1}}{n+1}(\ln ax)^m - \frac{m}{n+1}\int x^n(\ln ax)^{m-1}\,dx.$

$$\begin{aligned}
x^1 &= x^{\underline{1}} &&= x^{\overline{1}}\\
x^2 &= x^{\underline{2}}+x^{\underline{1}} &&= x^{\overline{2}}-x^{\overline{1}}\\
x^3 &= x^{\underline{3}}+3x^{\underline{2}}+x^{\underline{1}} &&= x^{\overline{3}}-3x^{\overline{2}}+x^{\overline{1}}\\
x^4 &= x^{\underline{4}}+6x^{\underline{3}}+7x^{\underline{2}}+x^{\underline{1}} &&= x^{\overline{4}}-6x^{\overline{3}}+7x^{\overline{2}}-x^{\overline{1}}\\
x^5 &= x^{\underline{5}}+15x^{\underline{4}}+25x^{\underline{3}}+10x^{\underline{2}}+x^{\underline{1}} &&= x^{\underline{5}}-15x^{\overline{4}}+25x^{\overline{3}}-10x^{\overline{2}}+x^{\overline{1}}
\end{aligned}$$

$$\begin{aligned}
x^{\overline{1}} &= x^1 & x^{\underline{1}} &= x^1\\
x^{\overline{2}} &= x^2+x^1 & x^{\underline{2}} &= x^2-x^1\\
x^{\overline{3}} &= x^3+3x^2+2x^1 & x^{\underline{3}} &= x^3-3x^2+2x^1\\
x^{\overline{4}} &= x^4+6x^3+11x^2+6x^1 & x^{\underline{4}} &= x^4-6x^3+11x^2-6x^1\\
x^{\overline{5}} &= x^5+10x^4+35x^3+50x^2+24x^1 & x^{\underline{5}} &= x^5-10x^4+35x^3-50x^2+24x^1
\end{aligned}$$

Finite Calculus

Difference, shift operators:

$$\Delta f(x) = f(x+1)-f(x),$$
$$\mathrm{E}\,f(x) = f(x+1).$$

Fundamental Theorem:

$$f(x)=\Delta F(x) \Leftrightarrow \sum f(x)\delta x = F(x)+C.$$
$$\sum_a^b f(x)\delta x = \sum_{i=a}^{b-1} f(i).$$

Differences:

$\Delta(cu) = c\Delta u, \qquad \Delta(u+v) = \Delta u+\Delta v,$

$\Delta(uv) = u\Delta v + \mathrm{E}\,v\Delta u,$

$\Delta(x^{\underline{n}}) = nx^{\underline{n-1}},$

$\Delta(H_x) = x^{\underline{-1}}, \qquad \Delta(2^x) = 2^x,$

$\Delta(c^x) = (c-1)c^x, \qquad \Delta\binom{x}{m} = \binom{x}{m-1}.$

Sums:

$\sum cu\,\delta x = c\sum u\,\delta x,$

$\sum(u+v)\,\delta x = \sum u\,\delta x + \sum v\,\delta x,$

$\sum u\Delta v\,\delta x = uv - \sum \mathrm{E}\,v\Delta u\,\delta x,$

$\sum x^{\underline{n}}\,\delta x = \frac{x^{\underline{n+1}}}{m+1}, \qquad \sum x^{\underline{-1}}\,\delta x = H_x,$

$\sum c^x\,\delta x = \frac{c^x}{c-1}, \qquad \sum\binom{x}{m}\,\delta x = \binom{x}{m+1}.$

Falling Factorial Powers:

$x^{\underline{n}} = x(x-1)\cdots(x-n+1), \quad n>0,$

$x^{\underline{0}} = 1,$

$x^{\underline{n}} = \dfrac{1}{(x+1)\cdots(x+|n|)}, \quad n<0,$

$x^{\underline{n+m}} = x^{\underline{m}}(x-m)^{\underline{n}}.$

Rising Factorial Powers:

$x^{\overline{n}} = x(x+1)\cdots(x+n-1), \quad n>0,$

$x^{\overline{0}} = 1,$

$x^{\overline{n}} = \dfrac{1}{(x-1)\cdots(x-|n|)}, \quad n<0,$

$x^{\overline{n+m}} = x^{\overline{m}}(x+m)^{\overline{n}}.$

Conversion:

$x^{\underline{n}} = (-1)^n(-x)^{\overline{n}} = (x-n+1)^{\overline{n}}$

$\quad = 1/(x+1)^{\overline{-n}},$

$x^{\overline{n}} = (-1)^n(-x)^{\underline{n}} = (x+n-1)^{\underline{n}}$

$\quad = 1/(x-1)^{\underline{-n}},$

$$x^n = \sum_{k=1}^n \left\{ n \atop k \right\} x^{\underline{k}} = \sum_{k=1}^n \left\{ n \atop k \right\}(-1)^{n-k}x^{\overline{k}},$$

$$x^{\underline{n}} = \sum_{k=1}^n \left[n \atop k \right](-1)^{n-k}x^k,$$

$$x^{\overline{n}} = \sum_{k=1}^n \left[n \atop k \right]x^k.$$

Theoretical Computer Science Cheat Sheet

Series

Taylor's series:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + \cdots = \sum_{i=0}^{\infty} \frac{(x-a)^i}{i!} f^{(i)}(a).$$

Expansions:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \cdots = \sum_{i=0}^{\infty} x^i,$$

$$\frac{1}{1-cx} = 1 + cx + c^2x^2 + c^3x^3 + \cdots = \sum_{i=0}^{\infty} c^i x^i,$$

$$\frac{1}{1-x^n} = 1 + x^n + x^{2n} + x^{3n} + \cdots = \sum_{i=0}^{\infty} x^{ni},$$

$$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \cdots = \sum_{i=0}^{\infty} ix^i,$$

$$x^k \frac{d^n}{dx^n}\left(\frac{1}{1-x}\right) = x + 2^n x^2 + 3^n x^3 + 4^n x^4 + \cdots = \sum_{i=0}^{\infty} i^n x^i,$$

$$e^x = 1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{6}x^3 + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!},$$

$$\ln(1+x) = x - \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 - \tfrac{1}{4}x^4 - \cdots = \sum_{i=1}^{\infty} (-1)^{i+1}\frac{x^i}{i},$$

$$\ln\frac{1}{1-x} = x + \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 + \tfrac{1}{4}x^4 + \cdots = \sum_{i=1}^{\infty} \frac{x^i}{i},$$

$$\sin x = x - \tfrac{1}{3!}x^3 + \tfrac{1}{5!}x^5 - \tfrac{1}{7!}x^7 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!},$$

$$\cos x = 1 - \tfrac{1}{2!}x^2 + \tfrac{1}{4!}x^4 - \tfrac{1}{6!}x^6 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!},$$

$$\tan^{-1} x = x - \tfrac{1}{3}x^3 + \tfrac{1}{5}x^5 - \tfrac{1}{7}x^7 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)},$$

$$(1+x)^n = 1 + nx + \tfrac{n(n-1)}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{n}{i} x^i,$$

$$\frac{1}{(1-x)^{n+1}} = 1 + (n+1)x + \tbinom{n+2}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{i+n}{i} x^i,$$

$$\frac{x}{e^x - 1} = 1 - \tfrac{1}{2}x + \tfrac{1}{12}x^2 - \tfrac{1}{720}x^4 + \cdots = \sum_{i=0}^{\infty} \frac{B_i x^i}{i!},$$

$$\frac{1}{2x}(1 - \sqrt{1-4x}) = 1 + x + 2x^2 + 5x^3 + \cdots = \sum_{i=0}^{\infty} \frac{1}{i+1}\binom{2i}{i} x^i,$$

$$\frac{1}{\sqrt{1-4x}} = 1 + x + 2x^2 + 6x^3 + \cdots = \sum_{i=0}^{\infty} \binom{2i}{i} x^i,$$

$$\frac{1}{\sqrt{1-4x}}\left(\frac{1-\sqrt{1-4x}}{2x}\right)^n = 1 + (2+n)x + \tbinom{4+n}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{2i+n}{i} x^i,$$

$$\frac{1}{1-x}\ln\frac{1}{1-x} = x + \tfrac{3}{2}x^2 + \tfrac{11}{6}x^3 + \tfrac{25}{12}x^4 + \cdots = \sum_{i=1}^{\infty} H_i x^i,$$

$$\frac{1}{2}\left(\ln\frac{1}{1-x}\right)^2 = \tfrac{1}{2}x^2 + \tfrac{3}{4}x^3 + \tfrac{11}{24}x^4 + \cdots = \sum_{i=2}^{\infty} \frac{H_{i-1}x^i}{i},$$

$$\frac{x}{1-x-x^2} = x + x^2 + 2x^3 + 3x^4 + \cdots = \sum_{i=0}^{\infty} F_i x^i,$$

$$\frac{F_n x}{1 - (F_{n-1} + F_{n+1})x - (-1)^n x^2} = F_n x + F_{2n}x^2 + F_{3n}x^3 + \cdots = \sum_{i=0}^{\infty} F_{ni} x^i.$$

Ordinary power series:

$$A(x) = \sum_{i=0}^{\infty} a_i x^i.$$

Exponential power series:

$$A(x) = \sum_{i=0}^{\infty} a_i \frac{x^i}{i!}.$$

Dirichlet power series:

$$A(x) = \sum_{i=1}^{\infty} \frac{a_i}{i^x}.$$

Binomial theorem:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

Difference of like powers:

$$x^n - y^n = (x-y)\sum_{k=0}^{n-1} x^{n-1-k} y^k.$$

For ordinary power series:

$$\alpha A(x) + \beta B(x) = \sum_{i=0}^{\infty} (\alpha a_i + \beta b_i) x^i,$$

$$x^k A(x) = \sum_{i=k}^{\infty} a_{i-k} x^i,$$

$$\frac{A(x) - \sum_{i=0}^{k-1} a_i x^i}{x^k} = \sum_{i=0}^{\infty} a_{i+k} x^i,$$

$$A(cx) = \sum_{i=0}^{\infty} c^i a_i x^i,$$

$$A'(x) = \sum_{i=0}^{\infty} (i+1) a_{i+1} x^i,$$

$$xA'(x) = \sum_{i=1}^{\infty} i a_i x^i,$$

$$\int A(x)\,dx = \sum_{i=1}^{\infty} \frac{a_{i-1}}{i} x^i,$$

$$\frac{A(x) + A(-x)}{2} = \sum_{i=0}^{\infty} a_{2i} x^{2i},$$

$$\frac{A(x) - A(-x)}{2} = \sum_{i=0}^{\infty} a_{2i+1} x^{2i+1}.$$

Summation: If $b_i = \sum_{j=0}^{i} a_i$ then

$$B(x) = \frac{1}{1-x} A(x).$$

Convolution:

$$A(x)B(x) = \sum_{i=0}^{\infty} \left(\sum_{j=0}^{i} a_j b_{i-j}\right) x^i.$$

God made the natural numbers;
all the rest is the work of man.
– Leopold Kronecker

Theoretical Computer Science Cheat Sheet

Series

Expansions:

$$\frac{1}{(1-x)^{n+1}}\ln\frac{1}{1-x} = \sum_{i=0}^{\infty}(H_{n+i}-H_n)\binom{n+i}{i}x^i,$$

$$x^{\overline{n}} = \sum_{i=0}^{\infty}\left[{n \atop i}\right]x^i,$$

$$\left(\ln\frac{1}{1-x}\right)^n = \sum_{i=0}^{\infty}\left[{i \atop n}\right]\frac{n!x^i}{i!},$$

$$\tan x = \sum_{i=1}^{\infty}(-1)^{i-1}\frac{2^{2i}(2^{2i}-1)B_{2i}x^{2i-1}}{(2i)!},$$

$$\frac{1}{\zeta(x)} = \sum_{i=1}^{\infty}\frac{\mu(i)}{i^x},$$

$$\zeta(x) = \prod_p \frac{1}{1-p^{-x}},$$

$$\zeta^2(x) = \sum_{i=1}^{\infty}\frac{d(i)}{x^i} \quad \text{where } d(n)=\textstyle\sum_{d|n}1,$$

$$\zeta(x)\zeta(x-1) = \sum_{i=1}^{\infty}\frac{S(i)}{x^i} \quad \text{where } S(n)=\textstyle\sum_{d|n}d,$$

$$\zeta(2n) = \frac{2^{2n-1}|B_{2n}|}{(2n)!}\pi^{2n}, \quad n\in\mathbb{N},$$

$$\frac{x}{\sin x} = \sum_{i=0}^{\infty}(-1)^{i-1}\frac{(4^i-2)B_{2i}x^{2i}}{(2i)!},$$

$$\left(\frac{1-\sqrt{1-4x}}{2x}\right)^n = \sum_{i=0}^{\infty}\frac{n(2i+n-1)!}{i!(n+i)!}x^i,$$

$$e^x\sin x = \sum_{i=1}^{\infty}\frac{2^{i/2}\sin\frac{i\pi}{4}}{i!}x^i,$$

$$\sqrt{\frac{1-\sqrt{1-x}}{x}} = \sum_{i=0}^{\infty}\frac{(4i)!}{16^i\sqrt{2}(2i)!(2i+1)!}x^i,$$

$$\left(\frac{\arcsin x}{x}\right)^2 = \sum_{i=0}^{\infty}\frac{4^i i!^2}{(i+1)(2i+1)!}x^{2i}.$$

$$\left(\frac{1}{x}\right)^{\overline{-n}} = \sum_{i=0}^{\infty}\left\{{i \atop n}\right\}x^i,$$

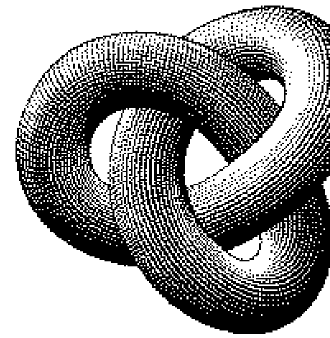
$$(e^x-1)^n = \sum_{i=0}^{\infty}\left\{{i \atop n}\right\}\frac{n!x^i}{i!},$$

$$x\cot x = \sum_{i=0}^{\infty}\frac{(-4)^iB_{2i}x^{2i}}{(2i)!},$$

$$\zeta(x) = \sum_{i=1}^{\infty}\frac{1}{i^x},$$

$$\frac{\zeta(x-1)}{\zeta(x)} = \sum_{i=1}^{\infty}\frac{\phi(i)}{i^x},$$

Escher's Knot



Stieltjes Integration

If G is continuous in the interval $[a,b]$ and F is nondecreasing then

$$\int_a^b G(x)\,dF(x)$$

exists. If $a\le b\le c$ then

$$\int_a^c G(x)\,dF(x) = \int_a^b G(x)\,dF(x) + \int_b^c G(x)\,dF(x).$$

If the integrals involved exist

$$\int_a^b \big(G(x)+H(x)\big)\,dF(x) = \int_a^b G(x)\,dF(x) + \int_a^b H(x)\,dF(x),$$

$$\int_a^b G(x)\,d\big(F(x)+H(x)\big) = \int_a^b G(x)\,dF(x) + \int_a^b G(x)\,dH(x),$$

$$\int_a^b c\cdot G(x)\,dF(x) = \int_a^b G(x)\,d\big(c\cdot F(x)\big) = c\int_a^b G(x)\,dF(x),$$

$$\int_a^b G(x)\,dF(x) = G(b)F(b) - G(a)F(a) - \int_a^b F(x)\,dG(x).$$

If the integrals involved exist, and F possesses a derivative F' at every point in $[a,b]$ then

$$\int_a^b G(x)\,dF(x) = \int_a^b G(x)F'(x)\,dx.$$

Cramer's Rule

If we have equations:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1\\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2\\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots &\\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

Let $A=(a_{i,j})$ and B be the column matrix (b_i). Then there is a unique solution iff $\det A\ne 0$. Let A_i be A with column i replaced by B. Then

$$x_i = \frac{\det A_i}{\det A}.$$

Improvement makes strait roads, but the crooked roads without Improvement, are roads of Genius.
– William Blake (The Marriage of Heaven and Hell)

00	47	18	76	29	93	85	34	61	52
86	11	57	28	70	39	94	45	02	63
95	80	22	67	38	71	49	56	13	04
59	96	81	33	07	48	72	60	24	15
73	69	90	82	44	17	58	01	35	26
68	74	09	91	83	55	27	12	46	30
37	08	75	19	92	84	66	23	50	41
14	25	36	40	51	62	03	77	88	99
21	32	43	54	65	06	10	89	97	78
42	53	64	05	16	20	31	98	79	87

The Fibonacci number system: Every integer n has a unique representation

$$n = F_{k_1} + F_{k_2} + \cdots + F_{k_m},$$

where $k_i \ge k_{i+1}+2$ for all i, $1\le i<m$ and $k_m\ge 2$.

Fibonacci Numbers

$1,1,2,3,5,8,13,21,34,55,89,\ldots$

Definitions:

$$F_i = F_{i-1}+F_{i-2}, \quad F_0=F_1=1,$$

$$F_{-i} = (-1)^{i-1}F_i,$$

$$F_i = \frac{1}{\sqrt{5}}\left(\phi^i - \hat{\phi}^i\right),$$

Cassini's identity: for $i>0$:

$$F_{i+1}F_{i-1} - F_i^2 = (-1)^i.$$

Additive rule:

$$F_{n+k} = F_kF_{n+1} + F_{k-1}F_n,$$

$$F_{2n} = F_nF_{n+1} + F_{n-1}F_n.$$

Calculation by matrices:

$$\begin{pmatrix} F_{n-2} & F_{n-1}\\ F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} 0 & 1\\ 1 & 1 \end{pmatrix}^n.$$

1 Notations

- The symbol const for `const`.
- The symbol ↶ for *function returned value.*
- Template class parameters lead by outlined character. For example: $\mathbb{T}$, $\mathbb{K}$ey, $\mathbb{C}$ompare. Interpreted in `template` definition context.
- Sometimes `class`, `typename` dropped.
- Template class parameters dropped, thus C sometimes used instead of C⟨$\mathbb{T}$⟩.
- "See example" by ☞, its output by ✇➠.

2 Containers

2.1 Pair

#include <utility>

```
template⟨class 𝕋1, class 𝕋2⟩
struct pair {
      𝕋1 first;   𝕋2 second;
      pair() {}
      pair(const 𝕋1& a, const 𝕋2& b):
          first(a), second(b) {}   };
```

2.1.1 Types

pair::**first_type**
pair::**second_type**

2.1.2 Functions & Operators

See also 2.2.3.
pair⟨$\mathbb{T}$1,$\mathbb{T}$2⟩
make_pair(const $\mathbb{T}$1&, const $\mathbb{T}$2&);

2.2 Containers — Common

Here X is any of
{**vector**, **deque**, **list**,
set, **multiset**, **map**, **multimap**}

2.2.1 Types

X::**value_type**
X::**reference**
X::**const_reference**
X::**iterator**
X::**const_iterator**
X::**reverse_iterator**
X::**const_reverse_iterator**
X::**difference_type**
X::**size_type**

Iterators reference value_type (See 6).

2.2.2 Members & Operators

X::X();
X::X(const X&);
X::~X();
X& X::**operator=**(const X&);

X::iterator	X::**begin**();	
X::const_iterator	X::**begin**()	const;
X::iterator	X::**end**();	
X::const_iterator	X::**end**()	const;
X::reverse_iterator	X::**rbegin**();	
X::const_reverse_iterator	X::**rbegin**()	const;
X::reverse_iterator	X::**rend**();	
X::const_reverse_iterator	X::**rend**()	const;

X::size_type	X::**size**() const;
X::size_type	X::**max_size**() const;
bool	X::**empty**() const;
void	X::**swap**(X& x);

void X::**clear**();

2.2.3 Comparison Operators

Let, X v, w. X may also be **pair** (2.1).

v `==` w	v `!=` w
v `<` w	v `>` w
v `<=` w	v `>=` w

All done lexicographically and ↶`bool`.

2.3 Sequence Containers

S is any of {**vector**, **deque**, **list**}

2.3.1 Constructors

S::S(S::size_type *n*,
 const S::value_type& *t*);
S::S(S::const_iterator *first*,
 S::const_iterator *last*); ☞7.2, 7.3

2.3.2 Members

S::iterator // *inserted copy*
S::**insert**(S::iterator *before*,
 const S::value_type& *val*);

S::iterator // *inserted copy*
S::**insert**(S::iterator *before*,
 S::size_type *nVal*,
 const S::value_type& *val*);

S::iterator // *inserted copy*
S::**insert**(S::iterator *before*,
 S::const_iterator *first*,
 S::const_iterator *last*);

S:iterator S::**erase**(S::iterator *position*);

S:iterator S::**erase**(S::const_iterator *first*,
↶ *post erased* S::const_iterator *last*);

void S::**push_back**(const S::value_type& x);
void S::**pop_back**();
S::reference S::**front**();
S::const_reference S::**front**() const;
S::reference S::**back**();
S::const_reference S::**back**() const;

2.4 Vector

#include <vector>

```
template⟨class 𝕋,
        class 𝔸lloc=allocator⟩
class vector;
```

See also 2.2 and 2.3.
size_type vector::**capacity**() const;
void vector::**reserve**(size_type n);
vector::reference
vector::**operator[]**(size_type i);
vector::const_reference
vector::**operator[]**(size_type i) const;
☞ 7.1.

2.5 Deque

#include <deque>

```
template⟨class 𝕋,
        class 𝔸lloc=allocator⟩
class deque;
```

Has all of **vector** functionality (see 2.4).
void deque::**push_front**(const $\mathbb{T}$& x);
void deque::**pop_front**();

2.6 List

#include <list>

```
template⟨class 𝕋,
        class 𝔸lloc=allocator⟩
class list;
```

See also 2.2 and 2.3.
void list::**pop_front**();
void list::**push_front**(const $\mathbb{T}$& x);

void // *move all x (&x ≠ `this`) before pos*
list::**splice**(iterator *pos*, list⟨$\mathbb{T}$⟩& x); ☞7.2

void // *move x's xElemPos before pos*
list::**splice** (iterator *pos*,
 list⟨$\mathbb{T}$⟩& x,
 iterator *xElemPos*); ☞7.2

void // *move x's [xFirst,xLast) before pos*
list::**splice** (iterator *pos*,
 list⟨$\mathbb{T}$⟩& x,
 iterator *xFirst*,
 iterator *xLast*); ☞7.2

void list::**remove**(const $\mathbb{T}$& *value*);
void list::**remove_if**($\mathbb{P}$redicate *pred*);
// *after call:* ∀ `this` *iterator* $p, *p \neq *(p+1)$
void list::**unique**(); // *remove repeats*
void // *as before but,* ¬*binPred*$(*p, *(p+1))$
list::**unique**($\mathbb{B}$inaryPredicate *binPred*);
// *Assuming both* `this` *and x sorted*
void list::**merge**(list⟨$\mathbb{T}$⟩& x);
// *merge and assume sorted by cmp*
void list::**merge**(list⟨$\mathbb{T}$⟩& x, $\mathbb{C}$ompare *cmp*);
void list::**reverse**();
void list::**sort**();
void list::**sort**($\mathbb{C}$ompare *cmp*);

2.7 Sorted Associative

Here A any of
{**set**, **multiset**, **map**, **multimap**}.

2.7.1 Types

For A=[multi]set, columns are the same

A::**key_type**	A::**value_type**
A::**keycompare**	A::**value_compare**

2.7.2 Constructors

A::A($\mathbb{C}$ompare c=$\mathbb{C}$ompare())
A::A(A::const_iterator *first*,
 A::const_iterator *last*,
 $\mathbb{C}$ompare c=$\mathbb{C}$ompare());

2.7.3 Members

A::keycompare A::**key_comp**() const;
A::value_compare A::**value_comp**() const;

A::iterator
A::**insert**(A::iterator *hint*,
 const A::value_type& *val*);

void A::**insert**(A::iterator *first*,
 A::iterator *last*);

A::size_type // *# erased*
A::**erase**(const A::key_type& k);
void A::**erase**(A::iterator p);
void A::**erase**(A::iterator *first*,
 A::iterator *last*);

A::size_type
A::**count**(const A::key_type& k) const;
A::iterator A::**find**(const A::key_type& k) const;

A::iterator
A::**lower_bound**(const A::key_type& *k*) const ;

A::iterator
A::**upper_bound**(const A::key_type& *k*) const ;

pair⟨A::iterator, A::iterator⟩ // *see 4.3.1*
A::**equal_range**(const A::key_type& *k*) const ;

2.8 Set

#include <set>

```
template⟨class 𝕂ey,
         class ℂompare=less⟨𝕂ey⟩,
         class 𝔸lloc=allocator⟩
class set;
```

See also 2.2 and 2.7.

set::**set**(const ℂompare& *cmp*=ℂompare());

pair⟨set::iterator, bool⟩ // `bool` = *if new*
set::**insert**(const set::value_type& *x*);

2.9 Multiset

#include <multiset.h>

```
template⟨class 𝕂ey,
         class ℂompare=less⟨𝕂ey⟩,
         class 𝔸lloc=allocator⟩
class multiset;
```

See also 2.2 and 2.7.

multiset::**multiset**(
    const ℂompare& *cmp*=ℂompare());

multiset::**multiset**(
    𝕀nputIterator *first*,
    𝕀nputIterator *last*,
    const ℂompare& *cmp*=ℂompare());

multiset::iterator // *inserted copy*
multiset::**insert**(const multiset::value_type& *x*);

2.10 Map

#include <map>

```
template⟨class 𝕂ey, class 𝕋,
         class ℂompare=less⟨𝕂ey⟩,
         class 𝔸lloc=allocator⟩
class map;
```

See also 2.2 and 2.7.

2.10.1 Types

map::**value_type** // pair⟨const 𝕂ey,𝕋⟩

2.10.2 Members

map::**map**(
    const ℂompare& *cmp*=ℂompare());

pair⟨map::iterator, bool⟩ // `bool` = *if new*
map::**insert**(const map::value_type& *x*);

𝕋& map:**operator**[](const map::key_type&);

map::const_iterator
map::**lower_bound**(
    const map::key_type& *k*) const ;

map::const_iterator
map::**upper_bound**(
    const map::key_type& *k*) const ;

pair⟨map::const_iterator, map::const_iterator⟩
map::**equal_range**(
const map::key_type& *k*) const ;

Example

```
typedef map<string, int> MSI;
MSI  nam2num;
nam2num.insert(MSI::value_type("one", 1));
nam2num.insert(MSI::value_type("two", 2));
nam2num.insert(MSI::value_type("three", 3));
int n3 = nam2num["one"] + nam2num["two"];
cout << n3 << " called ";
for (MSI::const_iterator i = nam2num.begin();
     i != nam2num.end();   ++i)
  if ((*i).second == n3)
    {cout << (*i).first << endl;}
```

➠ `3 called three`

2.11 Multimap

#include <multimap.h>

```
template⟨class 𝕂ey, class 𝕋,
         class ℂompare=less⟨𝕂ey⟩,
         class 𝔸lloc=allocator⟩
class multimap;
```

See also 2.2 and 2.7.

2.11.1 Types

multimap::**value_type** // pair⟨const 𝕂ey,𝕋⟩

2.11.2 Members

multimap::**multimap**(
    const ℂompare& *cmp*=ℂompare());

multimap::**multimap**(
    𝕀nputIterator *first*,
    𝕀nputIterator *last*,
    const ℂompare& *cmp*=ℂompare());

multimap::const_iterator
multimap::**lower_bound**(
    const multimap::key_type& *k*) const ;

multimap::const_iterator
multimap::**upper_bound**(
    const multimap::key_type& *k*) const ;

pair⟨multimap::const_iterator,
    multimap::const_iterator⟩
multimap::**equal_range**(
    const multimap::key_type& *k*) const ;

3 Container Adaptors

3.1 Stack Adaptor

#include <stack>

```
template⟨class 𝕋,
         class ℂontainer=deque⟨𝕋⟩ ⟩
class stack;
```

Default constructor. ℂontainer must have `back()`, `push_back()`, `pop_back()`. So **vector**, **list** and **deque** can be used.

bool stack::**empty**() const ;

ℂontainer::size_type stack::**size**() const ;

void
stack::**push**(const ℂontainer::value_type& x);

void stack::**pop**();

const ℂontainer::value_type&
stack::**top**() const ;

void ℂontainer::value_type& stack::**top**();

Comparision Operators

bool **operator==**(const stack& *s0*,
    const stack& *s1*);

bool **operator<**(const stack& *s0*,
    const stack& *s1*);

3.2 Queue Adaptor

#include <queue>

```
template⟨class 𝕋,
         class ℂontainer=deque⟨𝕋⟩ ⟩
class queue;
```

Default constructor. ℂontainer must have `empty()`, `size()`, `back()`, `front()`, `push_back()` and `pop_front()`. So **list** and **deque** can be used.

bool queue::**empty**() const ;

ℂontainer::size_type queue::**size**() const ;

void
queue::**push**(const ℂontainer::value_type& x);

void queue::**pop**();

const ℂontainer::value_type&
queue::**front**() const ;

ℂontainer::value_type& queue::**front**();

const ℂontainer::value_type&
queue::**back**() const ;

ℂontainer::value_type& queue::**back**();

Comparision Operators

bool **operator==**(const queue& *q0*,
    const queue& *q1*);

bool **operator<**(const queue& *q0*,
    const queue& *q1*);

3.3 Priority Queue

#include <queue>

```
template⟨class 𝕋,
         class ℂontainer=vector⟨𝕋⟩,
         class ℂompare=less⟨𝕋⟩ ⟩
class priority_queue;
```

ℂontainer must provide random access iterator and have `empty()`, `size()`, `front()`, `push_back()` and `pop_back()`. So **vector** and **deque** can be used.

Mostly implemented as *heap*.

3.3.1 Constructors

explicit **priority_queue::priority_queue**(
    const ℂompare& comp=ℂompare());

priority_queue::priority_queue(
    𝕀nputIterator *first*,
    𝕀nputIterator *last*,
    const ℂompare& *comp*=ℂompare());

3.3.2 Members

bool priority_queue::**empty**() const ;

ℂontainer::size_type
priority_queue::**size**() const ;

const ℂontainer::value_type&
priority_queue::**top**() const ;

ℂontainer::value_type& priority_queue::**top**();

void priority_queue::**push**(
    const ℂontainer::value_type& x);

void priority_queue::**pop**();

No comparision operators.

4 Algorithms

#include <algorithm>

STL algorithms use iterator type parameters. Their *names* suggest their category (See 6.1).

For abbreviation, the clause — `template ⟨class Foo, ...⟩` is dropped. The outlined leading character can suggest the `template` context.

Note: When looking at two sequences: $S_1 = [first_1, last_1)$ and $S_2 = [first_2, ?)$ or $S_2 = [?, last_2)$ — caller is responsible that function will not overflow S_2.

4.1 Query Algorithms

```
Function // f not changing [first, last)
for_each(InputIterator first,
         InputIterator last,
         Function      f);   ☞7.4
```

```
InputIterator // first i so i==last or *i==val
find(InputIterator first,
     InputIterator last,
     const T       val);   ☞7.2
```

```
InputIterator // first i so i==last or pred(i)
find_if(InputIterator first,
        InputIterator last,
        Predicate     pred);   ☞7.7
```

```
ForwardIterator // first duplicate
adjacent_find(ForwardIterator first,
              ForwardIterator last);
```

```
ForwardIterator // first binPred-duplicate
adjacent_find(ForwardIterator first,
              ForwardIterator last,
              BinaryPredicate binPred);
```

```
void // n = # equal val
count(ForwardIterator first,
      ForwardIterator last,
      const T         val,
      Size&           n);
```

```
void // n = # satisfying pred
count_if(ForwardIterator first,
         ForwardIterator last,
         Predicate       pred,
         Size&           n);
```

```
// ↶ bi-pointing to first !=
pair⟨InputIterator1, InputIterator2⟩
mismatch(InputIterator1 first1,
         InputIterator1 last1,
         InputIterator2 first2);
```

```
// ↶ bi-pointing to first binPred-mismatch
pair⟨InputIterator1, InputIterator2⟩
mismatch(InputIterator1  first1,
         InputIterator1  last1,
         InputIterator2  first2,
         BinaryPredicate binPred);
```

```
bool
equal(InputIterator1 first1,
      InputIterator1 last1,
      InputIterator2 first2);
```

```
bool
equal(InputIterator1  first1,
      InputIterator1  last1,
      InputIterator2  first2,
      BinaryPredicate binPred);
```

```
// [first2, last2) ⊑ [first1, last1)
ForwardIterator1
search(ForwardIterator1 first1,
       ForwardIterator1 last1,
       ForwardIterator2 first2,
       ForwardIterator2 last2);
```

```
// [first2, last2) ⊑binPred [first1, last1)
ForwardIterator1
search(ForwardIterator1 first1,
       ForwardIterator1 last1,
       ForwardIterator2 first2,
       ForwardIterator2 last2,
       BinaryPredicate  binPred);
```

4.2 Mutating Algorithms

```
OutputIterator // ↶ first2 + (last1 − first1)
copy(InputIterator  first1,
     InputIterator  last1,
     OutputIterator first2);
```

```
// ↶ last2 − (last1 − first1)
BidirectionalIterator2
copy_backward(
    BidirectionalIterator1 first1,
    BidirectionalIterator1 last1,
    BidirectionalIterator2 last2);
```

```
void swap(T& x, T& y);
```

```
ForwardIterator2 // ↶ first2 + #[first1, last1)
swap_ranges(ForwardIterator1 first1,
            ForwardIterator1 last1,
            ForwardIterator2 first2);
```

```
OutputIterator // ↶ result + (last1 − first1)
transform(InputIterator  first,
          InputIterator  last,
          OutputIterator result,
          UnaryOperation op);   ☞7.6
```

```
OutputIterator // ∀s_i^k ∈ S_k  r_i = bop(s_i^1, s_i^2)
transform(InputIterator1  first1,
          InputIterator1  last1,
          InputIterator2  first2,
          OutputIterator  result,
          BinaryOperation bop);
```

```
void replace(ForwardIterator first,
             ForwardIterator last,
             const T&        oldVal,
             const T&        newVal);
```

```
void
replace_if(ForwardIterator first,
           ForwardIterator last,
           Predicate&      pred,
           const T&        newVal);
```

```
OutputIterator // ↶ result2 + #[first, last)
replace_copy(InputIterator  first,
             InputIterator  last,
             OutputIterator result,
             const T&       oldVal,
             const T&       newVal);
```

```
OutputIterator // as above but using pred
replace_copy_if(InputIterator  first,
                InputIterator  last,
                OutputIterator result,
                Predicate&     pred,
                const T&       newVal);
```

```
void fill(ForwardIterator first,
          ForwardIterator last,
          const T&        value);
```

```
void fill_n(ForwardIterator first,
            Size            n,
            const T&        value);
```

```
void // by calling gen()
generate(ForwardIterator first,
         ForwardIterator last,
         Generator       gen);
```

```
void // n calls to gen()
generate_n(ForwardIterator first,
           Size            n,
           Generator       gen);
```

All variants of **remove** and **unique** return iterator to *new end* or *past last copied.*

```
ForwardIterator // [↶,last) is all value
remove(ForwardIterator first,
       ForwardIterator last,
       const T&        value);
```

```
ForwardIterator // as above but using pred
remove_if(ForwardIterator first,
          ForwardIterator last,
          Predicate       pred);
```

```
OutputIterator // ↶ past last copied
remove_copy(InputIterator  first,
            InputIterator  last,
            OutputIterator result,
            const T&       value);
```

```
OutputIterator // as above but using pred
remove_copy_if(InputIterator  first,
               InputIterator  last,
               OutputIterator result,
               Predicate      pred);
```

All variants of **unique** template functions remove *consecutive* (*binPred*-) duplicates. Thus usefull after sort (See 4.3).

```
ForwardIterator // [↶,last) gets repetitions
unique(ForwardIterator first,
       ForwardIterator last);
```

```
ForwardIterator // as above but using binPred
unique(ForwardIterator first,
       ForwardIterator last,
       BinaryPredicate binPred);
```

```
OutputIterator // ↶ past last copied
unique_copy(InputIterator  first,
            InputIterator  last,
            OutputIterator result,
            const T&       result);
```

```
OutputIterator // as above but using binPred
unique_copy(InputIterator   first,
            InputIterator   last,
            OutputIterator  result,
            BinaryPredicate binPred);
```

```
void
reverse(BidirectionalIterator first,
        BidirectionalIterator last);
```

```
OutputIterator // ↶ past last copied
reverse_copy(BidirectionalIterator first,
             BidirectionalIterator last,
             OutputIterator        result);
```

```
void // with first moved to middle
rotate(ForwardIterator first,
       ForwardIterator middle,
       ForwardIterator last);
```

```
OutputIterator // first to middle position
rotate_copy(ForwardIterator first,
            ForwardIterator middle,
            ForwardIterator last,
            OutputIterator  result);
```

```
void
random_shuffle(
    RandomAccessIterator  first,
    RandomAccessIterator  result);
```

```
void  // rand() returns double in [0,1)
random_shuffle(
    RandomAccessIterator  first,
    RandomAccessIterator  last,
    RandomGenerator       rand);
```

```
BidirectionalIterator  // begin with true
partition(BidirectionalIterator  first,
          BidirectionalIterator  last,
          Predicate              pred);
```

```
BidirectionalIterator  // begin with true
stable_partition(
    BidirectionalIterator  first,
    BidirectionalIterator  last,
    Predicate              pred);
```

4.3 Sort and Application

```
void sort(RandomAccessIterator  first,
          RandomAccessIterator  last);
```

```
void sort(RandomAccessIterator  first,
          RandomAccessIterator  last,
  ☞7.3    Compare               comp);
```

```
void
stable_sort(RandomAccessIterator  first,
            RandomAccessIterator  last);
```

```
void
stable_sort(RandomAccessIterator  first,
            RandomAccessIterator  last,
            Compare               comp);
```

```
void             // [first,middle) sorted,
partial_sort(  // [middle,last) eq-greater
    RandomAccessIterator  first,
    RandomAccessIterator  middle,
    RandomAccessIterator  last);
```

```
void  // as above but using comp(e_i, e_j)
partial_sort(
    RandomAccessIterator  first,
    RandomAccessIterator  middle,
    RandomAccessIterator  last,
    Compare               comp);
```

```
RandomAccessIterator  // post last sorted
partial_sort_copy(
    InputIterator         first,
    InputIterator         last,
    RandomAccessIterator  resultFirst,
    RandomAccessIterator  resultLast);
```

```
RandomAccessIterator
partial_sort_copy(
    InputIterator         first,
    InputIterator         last,
    RandomAccessIterator  resultFirst,
    RandomAccessIterator  resultLast,
    Compare               comp);
```

Let $n = position - first$, **nth_element** partitions $[first, last)$ into: $L = [first, position)$, e_n, $R = [position + 1, last)$ such that $\forall l \in L, \forall r \in R \quad l \not> e_n \le r$.

```
void
nth_element(
    RandomAccessIterator  first,
    RandomAccessIterator  position,
    RandomAccessIterator  last);
```

```
void  // as above but using comp(e_i, e_j)
nth_element(
    RandomAccessIterator  first,
    RandomAccessIterator  position,
    RandomAccessIterator  last,
    Compare               comp);
```

4.3.1 Binary Search

```
bool
binary_search(ForwardIterator  first,
              ForwardIterator  last,
              const T&         value);
```

```
bool
binary_search(ForwardIterator  first,
              ForwardIterator  last,
              const T&         value,
              Compare          comp);
```

```
ForwardIterator
lower_bound(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value);
```

```
ForwardIterator
lower_bound(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value,
            Compare          comp);
```

```
ForwardIterator
upper_bound(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value);
```

```
ForwardIterator
upper_bound(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value,
            Compare          comp);
```

equal_range returns iterators pair that lower_bound and upper_bound return.

```
pair⟨ForwardIterator,ForwardIterator⟩
equal_range(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value);
```

```
pair⟨ForwardIterator,ForwardIterator⟩
equal_range(ForwardIterator  first,
            ForwardIterator  last,
            const T&         value,
            Compare          comp);
```

☞ 7.5

4.3.2 Merge

Assuming $S_1 = [first_1, last_1)$ and $S_2 = [first_2, last_2)$ are sorted, stably merge them into $[result, result + N)$ where $N = |S_1| + |S_2|$.

```
OutputIterator
merge(InputIterator1  first1,
      InputIterator1  last1,
      InputIterator2  first2,
      InputIterator2  last2,
      OutputIterator  result);
```

```
OutputIterator
merge(InputIterator1  first1,
      InputIterator1  last1,
      InputIterator2  first2,
      InputIterator2  last2,
      OutputIterator  result,
      Compare         comp);
```

```
void  // ranges [first,middle) [middle,last)
inplace_merge(  // into [first,last)
    BidirectionalIterator  first,
    BidirectionalIterator  middle,
    BidirectionalIterator  last);
```

```
void  // as above but using comp
inplace_merge(
    BidirectionalIterator  first,
    BidirectionalIterator  middle,
    BidirectionalIterator  last,
    Compare                comp);
```

4.3.3 Functions on Sets

Can work on *sorted associcative* containers (see 2.7). For **multiset** the interpretation of — *union*, *intersection* and *difference* is by: *maximum*, *minimum* and *substraction* of occurrences respectably.

Let $S_i = [first_i, last_i)$ for $i = 1, 2$.

```
bool  // S1 ⊇ S2
includes(InputIterator1  first1,
         InputIterator1  last1,
         InputIterator2  first2,
         InputIterator2  last2);
```

```
bool  // as above but using comp
includes(InputIterator1  first1,
         InputIterator1  last1,
         InputIterator2  first2,
         InputIterator2  last2,
         Compare         comp);
```

```
OutputIterator  // S1 ∪ S2, ↶past end
set_union(InputIterator1  first1,
          InputIterator1  last1,
          InputIterator2  first2,
          InputIterator2  last2,
          OutputIterator  result);
```

```
OutputIterator  // as above but using comp
set_union(InputIterator1  first1,
          InputIterator1  last1,
          InputIterator2  first2,
          InputIterator2  last2,
          OutputIterator  result,
          Compare         comp);
```

```
OutputIterator  // S1 ∩ S2, ↶past end
set_intersection(InputIterator1  first1,
                 InputIterator1  last1,
                 InputIterator2  first2,
                 InputIterator2  last2,
                 OutputIterator  result);
```

```
OutputIterator  // as above but using comp
set_intersection(InputIterator1  first1,
                 InputIterator1  last1,
                 InputIterator2  first2,
                 InputIterator2  last2,
                 OutputIterator  result,
                 Compare         comp);
```

```
OutputIterator  // S1 \ S2, ↶past end
set_difference(InputIterator1  first1,
               InputIterator1  last1,
               InputIterator2  first2,
               InputIterator2  last2,
               OutputIterator  result);
```

```
OutputIterator  // as above but using comp
set_difference(InputIterator1  first1,
               InputIterator1  last1,
               InputIterator2  first2,
               InputIterator2  last2,
               OutputIterator  result,
               Compare         comp);
```

OutputIterator // $S_1 \triangle S_2$, ↶past end
```
set_symmetric_difference(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2,
    OutputIterator   result);
```

OutputIterator // *as above but using comp*
```
set_symmetric_difference(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2,
    OutputIterator   result,
    Compare          comp);
```

4.3.4 Heap

void // $(last - 1)$ *is pushed*
```
push_heap(RandomAccessIterator   first,
          RandomAccessIterator   last);
```

void // *as above but using comp*
```
push_heap(RandomAccessIterator   first,
          RandomAccessIterator   last,
          Compare                comp);
```

void // *first is popped*
```
pop_heap(RandomAccessIterator   first,
         RandomAccessIterator   last);
```

void // *as above but using comp*
```
pop_heap(RandomAccessIterator   first,
         RandomAccessIterator   last,
         Compare                comp);
```

void // *[first,last) arbitrary ordered*
```
make_heap(RandomAccessIterator   first,
          RandomAccessIterator   last);
```

void // *as above but using comp*
```
make_heap(RandomAccessIterator   first,
          RandomAccessIterator   last,
          Compare                comp);
```

void // *sort the [first,last) heap*
```
sort_heap(RandomAccessIterator   first,
          RandomAccessIterator   last);
```

void // *as above but using comp*
```
sort_heap(RandomAccessIterator   first,
          RandomAccessIterator   last,
          Compare                comp);
```

4.3.5 Min and Max

```
const T& min(const T& x0, const T& x1);

const T& min(const T&   x0,
             const T&   x1,
             Compare    comp);

const T& max(const T& x0, const T& x1);

const T& max(const T&   x0,
             const T&   x1,
             Compare    comp);

ForwardIterator
min_element(ForwardIterator   first,
            ForwardIterator   last);

ForwardIterator
min_element(ForwardIterator   first,
            ForwardIterator   last,
            Compare           comp);

ForwardIterator
max_element(ForwardIterator   first,
            ForwardIterator   last);

ForwardIterator
max_element(ForwardIterator   first,
            ForwardIterator   last,
            Compare           comp);
```

4.3.6 Permutations

To get all permutations, start with ascending sequence end with descending.

bool // ↶ *iff available*
```
next_permutation(
    BidirectionalIterator   first,
    BidirectionalIterator   last);
```

bool // *as above but using comp*
```
next_permutation(
    BidirectionalIterator   first,
    BidirectionalIterator   last,
    Compare                 comp);
```

bool // ↶ *iff available*
```
prev_permutation(
    BidirectionalIterator   first,
    BidirectionalIterator   last);
```

bool // *as above but using comp*
```
prev_permutation(
    BidirectionalIterator   first,
    BidirectionalIterator   last,
    Compare                 comp);
```

4.3.7 Lexicographic Order

```
bool lexicographical_compare(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2);

bool lexicographical_compare(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    InputIterator2   last2,
    Compare          comp);
```

4.4 Computational

#include <numeric>

T // $\sum_{[first,last)}$ ☞7.6
```
accumulate(InputIterator   first,
           InputIterator   last,
           T               initVal);
```

T // *as above but using binop*
```
accumulate(InputIterator     first,
           InputIterator     last,
           T                 initVal,
           BinaryOperation   binop);
```

T // $\sum_i e_i^1 \times e_i^2$ *for* $e_i^k \in S_k, (k = 1, 2)$
```
inner_product(InputIterator1   first1,
              InputIterator1   last1,
              InputIterator2   first2,
              T                initVal);
```

T // *Similar, using* $\sum^{(sum)}$ *and* $\times_{mult}$
```
inner_product(InputIterator1    first1,
              InputIterator1    last1,
              InputIterator2    first2,
              T                 initVal,
              BinaryOperation   sum,
              BinaryOperation   mult);
```

OutputIterator // $r_k = \sum_{i=first}^{first+k} e_i$
```
partial_sum(InputIterator    first,
            InputIterator    last,
            OutputIterator   result);
```

OutputIterator // *as above but using binop*
```
partial_sum(
    InputIterator     first,
    InputIterator     last,
    OutputIterator    result,
    BinaryOperation   binop);
```

OutputIterator // $r_k = s_k - s_{k-1}$ *for* $k > 0$
```
adjacent_difference(          // r0 = s0
    InputIterator    first,
    InputIterator    last,
    OutputIterator   result);
```

OutputIterator // *as above but using binop*
```
adjacent_difference(
    InputIterator     first,
    InputIterator     last,
    OutputIterator    result,
    BinaryOperation   binop);
```

5 Function Objects

#include <**functional**>

```
template<class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;}
```

Derived unary objects:
struct **negate**⟨T⟩;
struct **logical_not**⟨T⟩;
☞ 7.6

```
template<class Arg1, class Arg2,
         class Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;}
```

Following derived template objects accept two operands. Result obvious by the name.

struct **plus**⟨T⟩;
struct **minus**⟨T⟩;
struct **multiplies**⟨T⟩;
struct **divides**⟨T⟩;
struct **modulus**⟨T⟩;
struct **equal_to**⟨T⟩;
struct **not_equal_to**⟨T⟩;
struct **greater**⟨T⟩;
struct **less**⟨T⟩;
struct **greater_equal**⟨T⟩;
struct **less_equal**⟨T⟩;
struct **logical_and**⟨T⟩;
struct **logical_or**⟨T⟩;

5.1 Function Adaptors

5.1.1 Negators

```
template⟨class ℙredicate⟩
class unary_negate : public
  unary_function⟨ℙredicate::argument_type,
                 bool⟩;
```

unary_negate::**unary_negate**(
  ℙredicate *pred*);

bool // *negate pred*
unary_negate::**operator()**(
  ℙredicate::argument_type *x*);

unary_negate⟨ℙredicate⟩
not1(const ℙredicate *pred*);

```
template⟨class ℙredicate⟩
class binary_negate : public
  binary_function⟨
    ℙredicate::first_argument_type,
    ℙredicate::second_argument_type⟩;
    bool⟩;
```

binary_negate::**binary_negate**(
  ℙredicate *pred*);

bool // *negate pred*
binary_negate::**operator()**(
  ℙredicate::first_argument_type *x*
  ℙredicate::second_argument_type *y*);

binary_negate⟨ℙredicate⟩
not2(const ℙredicate *pred*);

5.1.2 Binders

```
template⟨class 𝕆peration⟩
class binder1st: public
  unary_function⟨
    𝕆peration::second_argument_type,
    𝕆peration::result_type⟩;
```

binder1st::**binder1st**(
const 𝕆peration& *op*,
const 𝕆peration::first_argument_type *y*);

// *argument_type from unary_function*
𝕆peration::result_type
binder1st::**operator()**(
  const binder1st::argument_type *x*);

binder1st⟨𝕆peration⟩
bind1st(const 𝕆peration& *op*, const 𝕋& *x*);

```
template⟨class 𝕆peration⟩
class binder2nd: public
  unary_function⟨
    𝕆peration::first_argument_type,
    𝕆peration::result_type⟩;
```

binder2nd::**binder2nd**(
const 𝕆peration& *op*,
const 𝕆peration::second_argument_type *y*);

// *argument_type from unary_function*
𝕆peration::result_type
binder2nd::**operator()**(
  const binder2nd::argument_type *x*);

binder2nd⟨𝕆peration⟩
bind2nd(const 𝕆peration& *op*, const 𝕋& *x*);
☞ 7.7.

5.1.3 Pointers to Functions

```
template⟨class 𝔸rg, class ℝesult⟩
class pointer_to_unary_function :
  public unary_function⟨𝔸rg, ℝesult⟩;
```

pointer_to_unary_function⟨𝔸rg, ℝesult⟩
ptr_fun(ℝesult(*x*)(𝔸rg));

```
template<class 𝔸rg1, class 𝔸rg2,
         class ℝesult>
class pointer_to_binary_function :
  public binary_function⟨𝔸rg1, 𝔸rg2,
                         ℝesult⟩;
```

pointer_to_binary_function⟨𝔸rg1, 𝔸rg2,
ℝesult⟩
ptr_fun(ℝesult(*x*)(𝔸rg1, 𝔸rg2));

6 Iterators

#include <**iterator**>

6.1 Iterators Categories

Here, we will use:

- `X` iterator type.
- `a, b` iterator values.
- `r` iterator reference (`X& r`).
- `t` a value type `T`.

Imposed by empty `struct` tags.

6.1.1 Input, Output, Forward

struct **input_iterator_tag** {}☞ 7.8
struct **output_iterator_tag** {}
struct **forward_iterator_tag** {}

In table follows requirements check list for **I**nput, **O**utput and **F**orward iterators.

Expression; Requirements		**I**	**O**	**F**
`X()` `X u`	might be singular			•
`X(a)`	⇒`X(a) == a`	•		•
	`*a=t` ⇔ `*X(a)=t`		•	
`X u(a)` `X u=a`	⇒ `u == a`	•		•
	`u` copy of `a`		•	
`a==b`	equivalence relation	•		•
`a!=b`	⇔`!(a==b)`	•		•
`r = a`	⇒ `r == a`			•
`*a`	convertible to `T`. `a==b` ⇔ `*a==*b`	•		•
`*a=t`	(for *forward*, if `X` mutable)		•	•
`++r`	result is dereferenceable or *past-the-end*. `&r == &++r`	•	•	•
	convertible to `const X&`	•	•	
	convertible to `X&` `r==s`⇔ `++r==++s`			•
`r++`	convertible to `X&` ⇔`{X x=r;++r;return x;}`	•	•	•
`*++r` `*r++`	convertible to `T`	•	•	•

☞ 7.7.

6.1.2 Bidirectional Iterators

struct **bidirectional_iterator_tag** {}

The **forward** requirements and:

`--r` Convertible to const `X&`. If ∃ `r=++s` then `--r` refers same as `s`. `&r==&--r`. `--(++r)==r`. (`--r == --s` ⇒ `r==s`.

`r--` ⇔ `{X x=r; --r; return x;}`.

6.1.3 Random Access Iterator

struct **random_access_iterator_tag** {}

The **bidirectional** requirements and (`m,n` iterator's *distance* (integral) value):

`r+=n` ⇔ `{for (m=n; m-->0; ++r);`
  `for (m=n; m++<0; --r);`
  `return r;}` //but time $= O(1)$.
`a+n` ⇔ `n+a` ⇔ `{X x=a; return a+=n]}`
`r-=n` ⇔ `r += -n`.
`a-n` ⇔ `a+(-n)`.
`b-a` Returns iterator's *distance* value n, such that `a+n == b`.
`a[n]` ⇔ `*(a+n)`.
`a<b` Convertible to `bool`, `<` total ordering.
`a<b` Convertible to `bool`, `>` opposite to `<`.
`a<=b` ⇔ `!(a>b)`.
`a>=b` ⇔ `!(a<b)`.

6.2 Stream Iterators

```
template⟨class 𝕋,
         class 𝔻istance=ptrdiff_t⟩
class istream_iterator :
    pubic iterator⟨input_iterator_tag, 𝕋, 𝔻istance⟩;
```

// *end of stream* ☞7.4
istream_iterator::**istream_iterator**();

istream_iterator::**istream_iterator**(
  istream& *s*); ☞7.4

istream_iterator::**istream_iterator**(
  const istream_iterator⟨𝕋, 𝔻istance⟩&);

istream_iterator::**~istream_iterator**();

const 𝕋& istream_iterator::**operator***() const;

istream_iterator& // *Read and store* 𝕋 *value*
istream_iterator::**operator++**() const;

bool // *all end-of-streams are equal*
operator==(const istream_iterator,
  const istream_iterator);

```
template⟨class 𝕋⟩
class ostream_iterator :
    public iterator⟨output_iterator_tag, void, ...⟩;
```

// *If delim* $\neq 0$ *add after each write*
ostream_iterator::**ostream_iterator**(
  ostream& *s*,
  const char* *delim=0);*

ostream_iterator::**ostream_iterator**(
  const ostream_iterator *s*);

ostream_iterator& // *Assign & write (*o=t)*
ostream_iterator::**operator***() const;

ostream_iterator&
ostream_iterator::**operator=**(
  const ostream_iterator *s*);

ostream_iterator& // *No-op*
ostream_iterator::**operator++**();

ostream_iterator& // *No-op*
ostream_iterator::**operator++**(int);

☞ 7.4.

6.3 Typedefs & Adaptors

template⟨$\mathbb{C}$ategory, $\mathbb{T}$,
$\mathbb{D}$istance=ptrdiff_t,
$\mathbb{P}$ointer=$\mathbb{T}$*, $\mathbb{R}$eference= $\mathbb{T}$&⟩
class **iterator** {
$\mathbb{C}$ategory **iterator_category**;
$\mathbb{T}$ **value_type**;
$\mathbb{D}$istance **difference_type**;
$\mathbb{P}$ointer **pointer**;
$\mathbb{R}$eference **reference**;}

6.3.1 Traits

template⟨$\mathbb{I}$⟩
class **iterator_traits** {
$\mathbb{I}$::iterator_category **iterator_category**;
$\mathbb{I}$::value_type **value_type**;
$\mathbb{I}$::difference_type **difference_type**;
$\mathbb{I}$::pointer **pointer**;
$\mathbb{I}$::reference **reference**;}

Pointer specilaizations: ☞ 7.8

template⟨$\mathbb{T}$⟩
class **iterator_traits**⟨$\mathbb{T}$*⟩ {
random_access_iterator_tag **iterator_category** ;
$\mathbb{T}$ **value_type**;
ptrdiff_t **difference_type**;
$\mathbb{T}$* **pointer**;
$\mathbb{T}$& **reference**;}

template⟨$\mathbb{T}$⟩
class **iterator_traits**⟨const $\mathbb{T}$*⟩ {
random_access_iterator_tag **iterator_category** ;
$\mathbb{T}$ **value_type**;
ptrdiff_t **difference_type**;
const $\mathbb{T}$* **pointer**;
const $\mathbb{T}$& **reference**;}

6.3.2 Reverse Iterator

Transform $[i \nearrow j) \mapsto [j-1 \searrow i-1)$.

template⟨$\mathbb{I}$ter⟩
class **reverse_iterator** : public iterator⟨
iterator_traits⟨$\mathbb{I}$ter⟩::iterator_category,
iterator_traits⟨$\mathbb{I}$ter⟩::value_type,
iterator_traits⟨$\mathbb{I}$ter⟩::difference_type,
iterator_traits⟨$\mathbb{I}$ter⟩::pointer,
iterator_traits⟨$\mathbb{I}$ter⟩::reference⟩;

Denote
RI = **reverse_iterator**
$\mathbb{AI}$ = $\mathbb{R}$andomAccessIterator.

Abbreviate:
typedef RI<$\mathbb{AI}$, $\mathbb{T}$,
$\mathbb{R}$eference, $\mathbb{D}$istance⟩ **self**;

// *Default constructor ⇒ singular value*
self::RI();

explicit // *Adaptor Constructor*
self::RI($\mathbb{AI}$*i*);

$\mathbb{AI}$ self::**base**(); // *adpatee's position*

// *so that:* `&*(RI(i)) == &*(i-1)` $\mathbb{R}$eference
self::**operator***();

self // *position to & return base()-1*
RI::**operator++**();

self& // *return old position and move*
RI::**operator++**(int); // *to base()-1*

self // *position to & return base()+1*
RI::**operator−−**();

self& // *return old position and move*
RI::**operator−−**(int); // *to base()+1*

bool // ⇔ *s0.base() == s1.base()*
operator==(const self& *s0*, const self& *s1*);

reverse_iterator Specific

self // *returned value positioned at base()-n*
reverse_iterator::**operator+**(
$\mathbb{D}$istance *n*) const;

self& // *change & return position to base()-n*
reverse_iterator::**operator+=**($\mathbb{D}$istance *n*);

self // *returned value positioned at base()+n*
reverse_iterator::**operator−**(
$\mathbb{D}$istance *n*) const;

self& // *change & return position to base()+n*
reverse_iterator::**operator−=**($\mathbb{D}$istance *n*);

$\mathbb{R}$eference // `*(*this + n)`
reverse_iterator::**operator**[]($\mathbb{D}$istance *n*);

$\mathbb{D}$istance // *r0.base() - r1.base()*
operator−(const self& *r0*, const self& *r1*);

self // *n + r.base()*
operator−($\mathbb{D}$istance *n*, const self& *r*);

bool // *r0.base() < r1.base()*
operator<(const self& *r0*, const self& *r1*);

6.3.3 Insert Iterators

template⟨class $\mathbb{C}$ontainer⟩
class **back_insert_iterator** :
public output_iterator;

template⟨class $\mathbb{C}$ontainer⟩
class **front_insert_iterator** :
public output_iterator;

template⟨class $\mathbb{C}$ontainer⟩
class **insert_iterator** :
public output_iterator;

Here $\mathbb{T}$ will denote the $\mathbb{C}$ontainer::value_type.

Constructors

explicit // ∃ $\mathbb{C}$ontainer::*push_back*(const $\mathbb{T}$&)
back_insert_iterator::back_insert_iterator(
$\mathbb{C}$ontainer& *x*);

explicit // ∃ $\mathbb{C}$ontainer::*push_front*(const $\mathbb{T}$&)
front_insert_iterator::front_insert_iterator(
$\mathbb{C}$ontainer& *x*);

// ∃ $\mathbb{C}$ontainer::*insert*(const $\mathbb{T}$&)
insert_iterator::insert_iterator(
$\mathbb{C}$ontainer *x*,
$\mathbb{C}$ontainer::iterator *i*);

Denote
InsIter = **back_insert_iterator**
insFunc = **push_back**
iterMaker = **back_inserter** ☞7.4
or
InsIter = **front_insert_iterator**
insFunc = **push_front**
iterMaker = **front_inserter**
or
InsIter = **insert_iterator**
insFunc = **insert**

Member Functions & Operators

InsIter& // *calls x.insFunc(val)*
InsIter::**operator=**(const $\mathbb{T}$& *val*);

InsIter& // *return* `*this`
InsIter::**operator***();

InsIter& // *no-op, just return* `*this`
InsIter::**operator++**();

InsIter& // *no-op, just return* `*this`
InsIter::**operator++**(int);

Template Function

InsIter // *return InsIter⟨$\mathbb{C}$ontainer⟩(x)*
iterMaker($\mathbb{C}$ontainer& *x*);

// *return insert_iterator⟨$\mathbb{C}$ontainer⟩(x, i)*
insert_iterator⟨$\mathbb{C}$ontainer⟩
inserter($\mathbb{C}$ontainer& *x*, $\mathbb{I}$terator *i*);

7 Examples

7.1 Vector

```
// safe get
int  vi(const vector<unsigned>& v, int i)
{ return(i < (int)v.size() ? (int)v[i] : -1);}

// safe set
void vin(vector<int>& v, unsigned i, int n) {
   int  nAdd = i - v.size() + 1;
   if (nAdd>0) v.insert(v.end(), nAdd, n);
   else v[i] = n;
}
```

7.2 List Splice

```
void lShow(ostream& os, const list<int>& l) {
 ostream_iterator<int> osi(os, " ");
 copy(l.begin(), l.end(), osi); os<<endl;}

void lmShow(ostream& os, const char* msg,
            const list<int>& l,
            const list<int>& m) {
 os << msg << (m.size() ? ":\n" : ": ");
 lShow(os, l);
 if (m.size()) lShow(os, m); } // lmShow

list<int>::iterator p(list<int>& l, int val)
{ return find(l.begin(), l.end(), val);}

 static int prim[] = {2, 3, 5, 7};
 static int perf[] = {6, 28, 496};
 const list<int> lPrimes(prim+0, prim+4);
 const list<int> lPerfects(perf+0, perf+3);
 list<int> l(lPrimes), m(lPerfects);
 lmShow(cout, "primes & perfects", l, m);
 l.splice(l.begin(), m);
 lmShow(cout, "splice(l.beg, m)", l, m);
 l = lPrimes; m = lPerfects;
 l.splice(l.begin(), m, p(m, 28));
 lmShow(cout, "splice(l.beg, m, ^28)", l, m);
 m.erase(m.begin(), m.end()); // <=>m.clear()
 l = lPrimes;
 l.splice(p(l, 3), l, p(l, 5));
 lmShow(cout, "5 before 3", l, m);
 l = lPrimes;
 l.splice(l.begin(), l, p(l, 7), l.end());
 lmShow(cout, "tail to head", l, m);
 l = lPrimes;
 l.splice(l.end(), l, l.begin(), p(l, 3));
 lmShow(cout, "head to tail", l, m);
```

✇ ➠

```
primes & perfects:
2 3 5 7
6 28 496
splice(l.beg, m): 6 28 496 2 3 5 7
splice(l.beg, m, ^28):
28 2 3 5 7
6 496
5 before 3: 2 5 3 7
tail to head: 7 2 3 5
head to tail: 3 5 7 2
```

7.3 Compare Object Sort

```
class ModN {
 public:
  ModN(unsigned m): _m(m) {}
  bool operator ()(const unsigned& u0,
                   const unsigned& u1)
       {return ((u0 % _m) < (u1 % _m));}
 private: unsigned _m;
}; // ModN

 ostream_iterator<unsigned> oi(cout, " ");
 unsigned  q[6];
 for (int n=6, i=n-1;  i>=0;  n=i--)
    q[i] = n*n*n*n;
 cout<<"four-powers:   ";
 copy(q + 0, q + 6, oi);
 for (unsigned b=10; b<=1000; b *= 10) {
  vector<unsigned>  sq(q + 0, q + 6);
  sort(sq.begin(), sq.end(), ModN(b));
  cout<<endl<<"sort mod "<<setw(4)<<b<<": ";
  copy(sq.begin(), sq.end(), oi);
 } cout << endl;
```

```
four-powers:   1 16 81 256 625 1296
sort mod   10: 1 81 625 16 256 1296
sort mod  100: 1 16 625 256 81 1296
sort mod 1000: 1 16 81 256 1296 625
```

7.4 Stream Iterators

```
void unitRoots(int n) {
 cout << "unit " << n << "-roots:" << endl;
 vector<complex<float> > roots;
 float  arg = 2.*M_PI/(float)n;
 complex<float> r, r1 = polar((float)1., arg);
 for (r = r1; --n;  r *= r1)
   roots.push_back(r);
 copy(roots.begin(), roots.end(),
      ostream_iterator<complex<float> >(cout,
                                       "\n"));
} // unitRoots

 {ofstream o("primes.txt"); o << "2 3 5";}
 ifstream pream("primes.txt");
 vector<int> p;
 istream_iterator<int>  priter(pream);
 istream_iterator<int>  eosi;
 copy(priter, eosi, back_inserter(p));
 for_each(p.begin(), p.end(), unitRoots);
```

```
unit 2-roots:
(-1.000,-0.000)
unit 3-roots:
(-0.500,0.866)
(-0.500,-0.866)
unit 5-roots:
(0.309,0.951)
(-0.809,0.588)
(-0.809,-0.588)
(0.309,-0.951)
```

7.5 Binary Search

```
 // first 5 Fibonacci
 static int fb5[] = {1, 1, 2, 3, 5};
 for (int n = 0; n <= 6; ++n) {
   pair<int*,int*> p =
      equal_range(fb5, fb5+5, n);
   cout<< n <<":["<< p.first-fb5 <<','
                   << p.second-fb5 <<") ";
   if (n==3 || n==6) cout << endl;
 }
```

```
0:[0,0) 1:[0,2) 2:[2,3) 3:[3,4)
4:[4,4) 5:[4,5) 6:[5,5)
```

7.6 Transform & Numeric

```
template <class T>
class AbsPwr : public unary_function<T, T> {
 public:
    AbsPwr(T p): _p(p) {}
    T operator()(const T& x) const
       { return pow(fabs(x), _p); }
 private: T _p;
}; // AbsPwr

template<typename InpIter> float
normNP(InpIter xb, InpIter xe, float p) {
  vector<float>  vf;
  transform(xb, xe, back_inserter(vf),
            AbsPwr<float>(p > 0. ? p : 1.));
  return( (p > 0.)
  ? pow(accumulate(vf.begin(), vf.end(), 0.),
        1./p)
  : *(max_element(vf.begin(), vf.end())));
} // normNP

float distNP(const float* x, const float* y,
             unsigned n, float p) {
  vector<float>  diff;
  transform(x, x + n, y, back_inserter(diff),
            minus<float>());
  return normNP(diff.begin(), diff.end(), p);
} // distNP

 float  x3y4[] = {3., 4.,  0.};
 float  z12[]  = {0., 0., 12.};
 float  p[] = {1., 2., M_PI, 0.};
 for (int i=0; i<4;  ++i) {
  float d = distNP(x3y4, z12, 3, p[i]);
  cout << "d_{" << p[i] << "}=" << d << endl;
 }
```

```
d_{1}=19
d_{2}=13
d_{3.14159}=12.1676
d_{0}=12
```

7.7 Iterator and Binder

```
// self-refering int
class Interator : public
  iterator<input_iterator_tag, int, size_t> {
  int  _n;
 public:
  Interator(int n=0) : _n(n) {}
  int operator*() const {return _n;}
  Interator& operator++() {
    ++_n;  return *this;  }
  Interator  operator++(int) {
    Interator t(*this);
    ++_n;   return t;}
}; // Interator
bool operator==(const Interator& i0,
                const Interator& i1)
{ return (*i0 == *i1); }
bool operator!=(const Interator& i0,
                const Interator& i1)
{ return !(i0 == i1); }

struct Fermat: public
    binary_function<int, int, bool> {
  Fermat(int p=2) : n(p) {}
  int n;
  int nPower(int t) const { // t^n
    int i=n, tn=1;
    while (i--) tn *= t;
    return tn; } // nPower
  int nRoot(int t) const {
    return (int)pow(t +.1, 1./n); }
  int xNyN(int x, int y) const   {
    return(nPower(x)+nPower(y)); }
  bool operator()(int x, int y) const {
    int zn = xNyN(x, y), z = nRoot(zn);
    return(zn == nPower(z));  }
}; // Fermat

 for (int n=2; n<=Mp; ++n) {
   Fermat fermat(n);
   for (int x=1; x<Mx; ++x) {
     binder1st<Fermat>
       fx = bind1st(fermat, x);
     Interator iy(x), iyEnd(My);
     while ((iy = find_if(++iy, iyEnd, fx))
            != iyEnd) {
       int  y = *iy,
         z = fermat.nRoot(fermat.xNyN(x, y));
       cout << x << '^' << n << " + "
            << y << '^' << n << " = "
            << z << '^' << n << endl;
       if (n>2)
         cout << "Fermat is wrong!" << endl;
     }
   }
 }
```

```
3^2 + 4^2 = 5^2
5^2 + 12^2 = 13^2
6^2 + 8^2 = 10^2
7^2 + 24^2 = 25^2
```

7.8 Iterator Traits

```
template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e, input_iterator_tag) {
  cout << "mid(general):\n";
  Itr bm(b);  bool  next = false;
  for ( ;  b != e;  ++b, next = !next) {
    if (next) { ++bm; }
  }
  return *bm;
} // mid<input>

template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e,
    random_access_iterator_tag) {
  cout << "mid(random):\n";
  Itr bm = b + (e - b)/2;
  return *bm;
} // mid<random>

template <class Itr>
typename iterator_traits<Itr>::value_type
mid(Itr b, Itr e) {
  typename
  iterator_traits<Itr>::iterator_category t;
  mid(b, e, t);
} // mid

template <class Ctr>
void fillmid(Ctr& ctr) {
  static int perfects[5] =
    {6, 14, 496, 8128, 33550336},
    *pb = &perfects[0];
  ctr.insert(ctr.end(), pb, pb + 5);
  int m = mid(ctr.begin(), ctr.end());
  cout << "mid=" << m << "\n";
} // fillmid

  list<int> l;  vector<int> v;
  fillmid(l);   fillmid(v);
```

```
mid(general):
mid=496
mid(random):
mid=496
```