Universidad Privada Boliviana ICPC Team Notebook (2025-26)

Contents

1 Data structures

1.1 Binary Indexed Tree

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

#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}

// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

1.2 Union-find set

```
#include <iostream>
#include <vector>
using namespace std;
struct UnionFind {
    vector<int> C;
    UnionFind(int n) : C(n) { for (int i = 0; i < n; i++) C[i] = i; }
    int find(int x) { return (C[x] == x) ? x : C[x] = find(C[x]); }
    void merge(int x, int y) { C[find(x)] = find(y); }
};
int main()
{
    int n = 5;
    UnionFind uf(n);
    uf.merge(0, 2);
    uf.merge(1, 0);
    uf.merge(3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << uf.find(i) << endl;
    return 0;
}
```

1.3 KD-tree

```
// -----------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well
//    distributed
//  - worst case for nearest-neighbor may be linear in pathological
//    case
//
// Sonny Chan, Stanford University, April 2009
// -----------------------------------------------------------------

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b)
{
    return a.x == b.x && a.y == b.y;
}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b)
{
    return a.x < b.x;
}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b)
{
    return a.y < b.y;
}

// squared distance between points
ntype pdist2(const point &a, const point &b)
{
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox
{
    ntype x0, x1, y0, y1;

    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);   x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);   y1 = max(y1, v[i].y);
        }
    }

    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0)       return pdist2(point(x0, y0), p);
            else if (p.y > y1)  return pdist2(point(x0, y1), p);
            else                return pdist2(point(x0, p.y), p);
        }
        else if (p.x > x1) {
            if (p.y < y0)       return pdist2(point(x1, y0), p);
            else if (p.y > y1)  return pdist2(point(x1, y1), p);
            else                return pdist2(point(x1, p.y), p);
        }
        else {
            if (p.y < y0)       return pdist2(point(p.x, y0), p);
```

```
            else if (p.y > y1)  return pdist2(point(p.x, y1), p);
            else                return 0;
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kdnode
{
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

    kdnode *first, *second; // two children of this kd-node

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp)
    {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode();   first->construct(vl);
            second = new kdnode();  second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree
{
    kdnode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

    // recursive search method returns squared distance to nearest point
    ntype search(kdnode *node, const point &p)
    {
        if (node->leaf) {
            // commented special case tells a point not to find itself
//            if (p == node->pt) return sentry;
//            else
                return pdist2(p, node->pt);
        }

        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);

        // choose the side with the closest bounding box to search first
        // (note that the other side is also searched if needed)
        if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
                best = min(best, search(node->second, p));
            return best;
        }
        else {
            ntype best = search(node->second, p);
            if (bfirst < best)
                best = min(best, search(node->first, p));
            return best;
        }
    }

    // squared distance to the nearest
    ntype nearest(const point &p) {
        return search(root, p);
    }
};

// --------------------------------------------------------------------------
// some basic test code here

int main()
{
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand()%100000, rand()%100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand()%100000, rand()%100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"
             << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

// --------------------------------------------------------------------------
```

1.4 Splay tree

```
#include <cstdio>
#include <algorithm>
using namespace std;

const int N_MAX = 130010;
const int oo = 0x3f3f3f3f;
struct Node
{
  Node *ch[2], *pre;
  int val, size;
  bool isTurned;
} nodePool[N_MAX], *null, *root;

Node *allocNode(int val)
{
  static int freePos = 0;
  Node *x = &nodePool[freePos ++];
  x->val = val, x->isTurned = false;
  x->ch[0] = x->ch[1] = x->pre = null;
  x->size = 1;
  return x;
}

inline void update(Node *x)
{
  x->size = x->ch[0]->size + x->ch[1]->size + 1;
}

inline void makeTurned(Node *x)
{
  if(x == null)
    return;
  swap(x->ch[0], x->ch[1]);
  x->isTurned ^= 1;
}

```

```
inline void pushDown(Node *x)
{
  if(x->isTurned)
  {
    makeTurned(x->ch[0]);
    makeTurned(x->ch[1]);
    x->isTurned ^= 1;
  }
}

inline void rotate(Node *x, int c)
{
  Node *y = x->pre;
  x->pre = y->pre;
  if(y->pre != null)
    y->pre->ch[y == y->pre->ch[1]] = x;
  y->ch[!c] = x->ch[c];
  if(x->ch[c] != null)
    x->ch[c]->pre = y;
  x->ch[c] = y, y->pre = x;
  update(y);
  if(y == root)
    root = x;
}

void splay(Node *x, Node *p)
{
  while(x->pre != p)
  {
    if(x->pre->pre == p)
      rotate(x, x == x->pre->ch[0]);
    else
    {
      Node *y = x->pre, *z = y->pre;
      if(y == z->ch[0])
      {
        if(x == y->ch[0])
          rotate(y, 1), rotate(x, 1);
        else
          rotate(x, 0), rotate(x, 1);
      }
      else
      {
        if(x == y->ch[1])
          rotate(y, 0), rotate(x, 0);
        else
          rotate(x, 1), rotate(x, 0);
      }
    }
  }
  update(x);
}

void select(int k, Node *fa)
{
  Node *now = root;
  while(1)
  {
    pushDown(now);
    int tmp = now->ch[0]->size + 1;
    if(tmp == k)
      break;
    else if(tmp < k)
      now = now->ch[1], k -= tmp;
    else
      now = now->ch[0];
  }
  splay(now, fa);
}

Node *makeTree(Node *p, int l, int r)
{
  if(l > r)
    return null;
  int mid = (l + r) / 2;
  Node *x = allocNode(mid);
  x->pre = p;
  x->ch[0] = makeTree(x, l, mid - 1);
  x->ch[1] = makeTree(x, mid + 1, r);
  update(x);
  return x;
}

int main()
{
  int n, m;
  null = allocNode(0);
  null->size = 0;
  root = allocNode(0);
  root->ch[1] = allocNode(oo);
  root->ch[1]->pre = root;
  update(root);

  scanf("%d%d", &n, &m);
  root->ch[1]->ch[0] = makeTree(root->ch[1], 1, n);
  splay(root->ch[1]->ch[0], null);

  while(m --)
  {
    int a, b;
    scanf("%d%d", &a, &b);
    a ++, b ++;
    select(a - 1, null);
    select(b + 1, root);
    makeTurned(root->ch[1]->ch[0]);
  }

  for(int i = 1; i <= n; i ++)
  {
    select(i + 1, null);
    printf("%d ", root->val);
  }
}
```

1.5 Lazy segment tree

```
public class SegmentTreeRangeUpdate {
  public long[] leaf;
  public long[] update;
  public int origSize;
  public SegmentTreeRangeUpdate(int[] list) {
    origSize = list.length;
    leaf = new long[4*list.length];
    update = new long[4*list.length];
    build(1,0,list.length-1,list);
  }
  public void build(int curr, int begin, int end, int[] list) {
    if(begin == end)
      leaf[curr] = list[begin];
    else  {
      int mid = (begin+end)/2;
      build(2 * curr, begin, mid, list);
      build(2 * curr + 1, mid+1, end, list);
      leaf[curr] = leaf[2*curr] + leaf[2*curr+1];
    }
  }
  public void update(int begin, int end, int val) {
    update(1,0,origSize-1,begin,end,val);
  }
  public void update(int curr,  int tBegin, int tEnd, int begin, int end, int val)  {
    if(tBegin >= begin && tEnd <= end)
      update[curr] += val;
    else  {
      leaf[curr] += (Math.min(end,tEnd)-Math.max(begin,tBegin)+1) * val;
      int mid = (tBegin+tEnd)/2;
      if(mid >= begin && tBegin <= end)
        update(2*curr, tBegin, mid, begin, end, val);
      if(tEnd >= begin && mid+1 <= end)
        update(2*curr+1, mid+1, tEnd, begin, end, val);
    }
  }
  public long query(int begin, int end) {
    return query(1,0,origSize-1,begin,end);
  }
  public long query(int curr, int tBegin, int tEnd, int begin, int end) {
    if(tBegin >= begin && tEnd <= end)  {
      if(update[curr] != 0) {
        leaf[curr] += (tEnd-tBegin+1) * update[curr];
        if(2*curr < update.length){
          update[2*curr] += update[curr];
          update[2*curr+1] += update[curr];
        }
        update[curr] = 0;
```

```
      }
      return leaf[curr];
    }
    else  {
      leaf[curr] += (tEnd-tBegin+1) * update[curr];
      if(2*curr < update.length){
        update[2*curr] += update[curr];
        update[2*curr+1] += update[curr];
      }
      update[curr] = 0;
      int mid = (tBegin+tEnd)/2;
      long ret = 0;
      if(mid >= begin && tBegin <= end)
        ret += query(2*curr, tBegin, mid, begin, end);
      if(tEnd >= begin && mid+1 <= end)
        ret += query(2*curr+1, mid+1, tEnd, begin, end);
      return ret;
    }
  }
}
```

1.6 Segment tree (range flip + sum)

```
/*
  Segment tree with lazy propagation
  Operation: range flip (0 <-> 1) and range sum query
  - update(l, r): flip bits in [l, r) (half-open)
  - query(l, r): sum of ones in [l, r)

  Notes:
  - All operations are O(log N)
  - Uses lazy XOR flag; flipping a segment of length L: sum = L - sum
*/

#include <bits/stdc++.h>
using namespace std;

struct SegTreeFlip {
  int n;                   // array size
  vector<int> seg;         // sums
  vector<unsigned char> lazy; // lazy flip flags (0/1)

  SegTreeFlip(int n = 0) { init(n); }

  void init(int n_) {
    n = n_;
    seg.assign(4 * max(1, n), 0);
    lazy.assign(4 * max(1, n), 0);
  }

  // apply flip to node covering [l, r)
  inline void apply(int id, int l, int r) {
    seg[id] = (r - l) - seg[id];
    lazy[id] ^= 1;
  }

  inline void push(int id, int l, int r) {
    if (!lazy[id] || r - l == 1) return;
    int m = (l + r) >> 1;
    int lc = id << 1, rc = lc | 1;
    apply(lc, l, m);
    apply(rc, m, r);
    lazy[id] = 0;
  }

  // build from initial array a (0/1 values)
  void build(const vector<int>& a, int id, int l, int r) {
    if (r - l == 1) { seg[id] = (l < (int)a.size() ? a[l] : 0); return; }
    int m = (l + r) >> 1, lc = id << 1, rc = lc | 1;
    build(a, lc, l, m);
    build(a, rc, m, r);
    seg[id] = seg[lc] + seg[rc];
  }
  void build(const vector<int>& a) { init((int)a.size()); build(a, 1, 0, n); }

  // flip [ql, qr)
  void update(int ql, int qr, int id, int l, int r) {
    if (qr <= l || r <= ql) return;
    if (ql <= l && r <= qr) { apply(id, l, r); return; }
    push(id, l, r);
    int m = (l + r) >> 1, lc = id << 1, rc = lc | 1;
    update(ql, qr, lc, l, m);
    update(ql, qr, rc, m, r);
    seg[id] = seg[lc] + seg[rc];
  }
  void update(int l, int r) { update(l, r, 1, 0, n); }

  // sum [ql, qr)
  int query(int ql, int qr, int id, int l, int r) {
    if (qr <= l || r <= ql) return 0;
    if (ql <= l && r <= qr) return seg[id];
    push(id, l, r);
    int m = (l + r) >> 1;
    return query(ql, qr, id << 1, l, m) + query(ql, qr, id << 1 | 1, m, r);
  }
  int query(int l, int r) { return query(l, r, 1, 0, n); }
};

// Example usage:
// int main(){
//   vector<int> a = {0,1,0,1,1,0};
//   SegTreeFlip st; st.build(a);
//   cout << st.query(0, 6) << "\n"; // 3
//   st.update(1, 5); // flip indices [1..4]
//   cout << st.query(0, 6) << "\n"; // updated sum
// }
```

1.7 Sparse table (RMQ/idempotent)

```
/*
  Sparse Table for idempotent, associative operations (e.g., min, max, gcd)
  - Build O(n log n), query O(1)
*/

#include <bits/stdc++.h>
using namespace std;

struct SparseTable {
  int n, K; vector<int> lg; vector<vector<long long>> st;
  function<long long(long long,long long)> f;
  SparseTable() {}
  SparseTable(const vector<long long>& a, function<long long(long long,long long)> op) { build(a, op); }
  void build(const vector<long long>& a, function<long long(long long,long long)> op) {
    f = op; n = (int)a.size(); K = 32 - __builtin_clz(n);
    lg.assign(n + 1, 0); for (int i = 2; i <= n; ++i) lg[i] = lg[i/2] + 1;
    st.assign(K, vector<long long>(n)); st[0].assign(a.begin(), a.end());
    for (int k = 1; k < K; ++k) {
      int len = 1 << k;
      for (int i = 0; i + len <= n; ++i) st[k][i] = f(st[k-1][i], st[k-1][i + (len>>1)]);
    }
  }
  long long query(int l, int r) const { // [l, r)
    int k = lg[r - l];
    return f(st[k][l], st[k][r - (1 << k)]);
  }
};
```

1.8 Monotonic queue (sliding window)

```
/*
  Monotonic Queue for sliding window min/max in O(n)
  - window_min(a, k): minima of every subarray of length k
  - window_max(a, k): maxima of every subarray of length k
*/

#include <bits/stdc++.h>
using namespace std;
```

```cpp

vector<long long> window_min(const vector<long long>& a, int k){
  deque<int> dq; vector<long long> res; int n=a.size();
  for(int i=0;i<n;++i){
    while(!dq.empty() && dq.front() <= i-k) dq.pop_front();
    while(!dq.empty() && a[dq.back()] >= a[i]) dq.pop_back();
    dq.push_back(i);
    if(i>=k-1) res.push_back(a[dq.front()]);
  }
  return res;
}

vector<long long> window_max(const vector<long long>& a, int k){
  deque<int> dq; vector<long long> res; int n=a.size();
  for(int i=0;i<n;++i){
    while(!dq.empty() && dq.front() <= i-k) dq.pop_front();
    while(!dq.empty() && a[dq.back()] <= a[i]) dq.pop_back();
    dq.push_back(i);
    if(i>=k-1) res.push_back(a[dq.front()]);
  }
  return res;
}
```

1.9 Lowest common ancestor

```cpp
const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes];  // children[i] contains the children of node i
int A[max_nodes][log_max_nodes+1];  // A[i][j] is the 2^j-th ancestor of node i, or -1
      if that ancestor does not exist
int L[max_nodes];     // L[i] is the distance between node i and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
  return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>=  8; p +=  8; }
    if (n >= 1<< 4) { n >>=  4; p +=  4; }
    if (n >= 1<< 2) { n >>=  2; p +=  2; }
    if (n >= 1<< 1) {           p +=  1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
  DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
  swap(p, q);

    // "binary search" for the ancestor of node p situated on the same level as q
    for(int i = log_num_nodes; i >= 0; i--)
  if(L[p] - (1<<i) >= L[q])
      p = A[p][i];

    if(p == q)
  return p;

    // "binary search" for the LCA
    for(int i = log_num_nodes; i >= 0; i--)
  if(A[p][i] != -1 && A[p][i] != A[q][i])
  {
      p = A[p][i];
      q = A[q][i];
  }

    return A[p][0];
}

int main(int argc,char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes=lb(num_nodes);

    for(int i = 0; i < num_nodes; i++)
    {
  int p;
  // read p, the parent of node i or -1 if node i is the root

  A[i][0] = p;
  if(p != -1)
      children[p].push_back(i);
  else
      root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
  for(int i = 0; i < num_nodes; i++)
      if(A[i][j-1] != -1)
    A[i][j] = A[A[i][j-1]][j-1];
      else
    A[i][j] = -1;

    // precompute L
    DFS(root, 0);


    return 0;
}
```

2 Strings

2.1 Aho-Corasick (múltiples patrones)

```cpp
/*
  Aho-Corasick automaton (multiple pattern matching)
  - Builds a trie of patterns with failure links (BFS)
  - search(text) returns, for each pattern, all starting indices where it occurs

  Usage:
    vector<string> patterns = {"he", "she", "his", "hers"};
    AhoCorasick ac(patterns);
    auto occ = ac.search("ahishers");
  // occ[i] are the starting indices of patterns[i] in the text
*/

#include <bits/stdc++.h>
using namespace std;

struct Node {
  // trie edges by character
  unordered_map<char, Node*> next;
  // suffix link (failure link)
  Node* link = nullptr;
  // output link to next terminal node on the suffix chain
  Node* out = nullptr;
  // parent and incoming char (useful for building links)
  Node* parent = nullptr;
  char ch = 0;
  // if terminal: id of the pattern ending here, else -1
  int patId = -1;
};

struct AhoCorasick {
  Node* root;
  vector<string> pats;
  vector<Node*> nodes; // to free

  AhoCorasick(const vector<string>& patterns) : root(new Node()), pats(patterns) {
    nodes.push_back(root);
    for (int i = 0; i < (int)pats.size(); ++i) insert(pats[i], i);
    build();
  }

  ~AhoCorasick() { // simple deletion
    for (Node* n : nodes) delete n;
  }

  void insert(const string& s, int id) {
    Node* cur = root;
    for (char c : s) {
      auto it = cur->next.find(c);
      if (it == cur->next.end()) {
        Node* nx = new Node();
        nx->parent = cur; nx->ch = c;
        nodes.push_back(nx);
        cur->next[c] = nx;
        cur = nx;
      } else cur = it->second;
    }
    cur->patId = id;
  }

  void build() {
    queue<Node*> q;
    // root's direct children: link to root
    root->link = root; root->out = nullptr;
    for (auto& kv : root->next) {
      kv.second->link = root;
      kv.second->out = (kv.second->patId != -1) ? kv.second : nullptr;
      q.push(kv.second);
    }
    // BFS
    while (!q.empty()) {
      Node* v = q.front(); q.pop();
      for (auto& kv : v->next) {
        char c = kv.first; Node* u = kv.second; q.push(u);
        // compute failure link
        Node* f = v->link;
        while (f != root && !hasEdge(f, c)) f = f->link;
        if (hasEdge(f, c) && f->next[c] != u) f = f->next[c];
        u->link = (hasEdge(f, c) ? f : root);
        // compute output link: nearest terminal on suffix chain
        if (u->patId != -1) u->out = u; else u->out = u->link->out;
      }
    }
  }

  static inline bool hasEdge(Node* n, char c) { return n->next.find(c) != n->next.end(); }

  vector<vector<int>> search(const string& text) const {
    vector<vector<int>> res(pats.size());
    Node* cur = root;
    for (int i = 0; i < (int)text.size(); ++i) {
      char c = text[i];
      while (cur != root && !hasEdge(cur, c)) cur = cur->link;
      if (hasEdge(cur, c)) cur = cur->next.at(c);
      // report all matches at this state via out links
      for (Node* t = (cur->patId != -1 ? cur : cur->out); t; t = (t->out && t->out != t ? t->out : nullptr)) {
        int id = t->patId;
        if (id != -1) res[id].push_back(i - (int)pats[id].size() + 1);
      }
    }
    return res;
  }
};

// Example usage
// int main() {
//   vector<string> patterns = {"he", "she", "his", "hers"};
//   string text = "ahishers";
//   AhoCorasick ac(patterns);
//   auto occ = ac.search(text);
//   for (int i = 0; i < (int)patterns.size(); ++i) {
//     cout << patterns[i] << ": ";
//     for (int p : occ[i]) cout << p << ' ';
//     cout << '\n';
//   }
// }
```

2.2 Knuth-Morris-Pratt

```cpp
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respectively.
*/

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

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p)
{
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
```

```cpp
      cout << "matched at index " << i-k << ": ";
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main()
{
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
  return 0;
}
```

2.3 Z-function (linear)

```cpp
/*
  Z-Function (linear time)
  - z[i] = length of longest substring starting at i which is also a prefix of s
  - z[0] = 0 by convention
  Applications: pattern matching on s = pattern#text, string properties
*/

#include <bits/stdc++.h>
using namespace std;

vector<int> z_function(const string& s) {
  int n = (int)s.size();
  vector<int> z(n, 0);
  int l = 0, r = 0;
  for (int i = 1; i < n; ++i) {
    if (i <= r) z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > r) { l = i; r = i + z[i] - 1; }
  }
  return z;
}
```

2.4 Manacher (longest palindromic substring)

```cpp
/*
  Manacher's algorithm - longest palindromic substring in O(n)
  - manacher_odd(s): d1[i] = radius of odd-length palindrome centered at i
  - manacher_even(s): d2[i] = radius of even-length palindrome centered between i-1
      and i
*/

#include <bits/stdc++.h>
using namespace std;

vector<int> manacher_odd(const string& s) {
  int n = (int)s.size();
  vector<int> d1(n);
  int l = 0, r = -1;
  for (int i = 0; i < n; ++i) {
    int k = 1;
    if (i <= r) k = min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) ++k;
    d1[i] = k;
    if (i + k - 1 > r) { l = i - k + 1; r = i + k - 1; }
  }
  return d1;
}

vector<int> manacher_even(const string& s) {
  int n = (int)s.size();
  vector<int> d2(n);
  int l = 0, r = -1;
  for (int i = 0; i < n; ++i) {
    int k = 0;
    if (i <= r) k = min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) ++k;
    d2[i] = k;
    if (i + k - 1 > r) { l = i - k; r = i + k - 1; }
  }
  return d2;
}
```

2.5 Rolling hash (double mod)

```cpp
/*
  Rolling Hash (polynomial) with double mod
  - Precompute prefix hashes and powers; query substring hashes in O(1)
  - Suitable for equality checks, palindrome checks (with reverse), etc.
*/

#include <bits/stdc++.h>
using namespace std;

struct RollingHash {
  using ull = unsigned long long;
  static const ull M1 = 1000000007ULL, M2 = 1000000009ULL;
  static const ull B = 911382323ULL; // base
  int n;
  vector<ull> p1, p2, h1, h2;
  RollingHash() {}
  RollingHash(const string& s) { build(s); }
  void build(const string& s) {
    n = (int)s.size();
    p1.assign(n + 1, 1); p2.assign(n + 1, 1);
    h1.assign(n + 1, 0); h2.assign(n + 1, 0);
    for (int i = 0; i < n; ++i) {
      p1[i + 1] = (p1[i] * B) % M1;
      p2[i + 1] = (p2[i] * B) % M2;
      h1[i + 1] = (h1[i] * B + (unsigned char)s[i] + 1) % M1;
      h2[i + 1] = (h2[i] * B + (unsigned char)s[i] + 1) % M2;
    }
  }
  pair<ull, ull> get(int l, int r) const { // [l, r)
    ull x1 = (h1[r] + M1 - (h1[l] * p1[r - l]) % M1) % M1;
    ull x2 = (h2[r] + M2 - (h2[l] * p2[r - l]) % M2) % M2;
    return {x1, x2};
  }
};
```

2.6 Suffix array + LCP (Kasai)

```cpp
/*
  Suffix Array + LCP (Kasai) - O(n log n) build, O(n) LCP
  - computeSA(): builds suffix array for string s (cyclic-safe variant as given)
  - computeLCP(): builds LCP array between adjacent suffixes in SA order

  This implementation mirrors a common contest template and includes:
    - macros for brevity (forn, forsn, etc.)
    - arrays r (SA order), p (rank), lcp (Kasai)

  Usage:
    s = "banana"; n = (int)s.size();
    computeSA();
    computeLCP();
*/

#include <bits/stdc++.h>
using namespace std;

#define forn(i,n) for (int i = 0; i < (int)(n); i++)
#define forsn(i,s,n) for (int i = (int)(s); i < (int)(n); i++)
#define si(c) ((int)(c).size())

const int MAXN = 100000 + 100;

string s;
int n, t;
int r[MAXN], p[MAXN], lcp[MAXN], np_[MAXN];
```

```cpp

static inline bool bcomp(int i, int j) { return s[i] < s[j]; }
static inline bool beq(int i, int j) { return s[i] == s[j]; }
static inline bool comp(int i, int j) {
  return make_pair(p[i], p[(i + t) % n]) < make_pair(p[j], p[(j + t) % n]);
}
static inline bool eq(int i, int j) {
  return make_pair(p[i], p[(i + t) % n]) == make_pair(p[j], p[(j + t) % n]);
}

void refine(bool (*eqf)(int,int)) {
  np_[r[0]] = 0;
  forsn(i, 1, n) {
    int ra = r[i], rp = r[i - 1];
    np_[ra] = np_[rp];
    if (!eqf(ra, rp)) np_[ra]++;
  }
  copy(np_, np_ + n, p);
}

void computeSA() {
  forn(i, n) r[i] = i;
  sort(r, r + n, bcomp);
  refine(beq);
  for (t = 1; t < 2 * n; t *= 2) {
    sort(r, r + n, comp);
    refine(eq);
  }
}

void computeLCP() {
  int L = 0;
  forn(i, n) if (p[i]) {
    int j = r[p[i] - 1];
    while (s[(i + L) % n] == s[(j + L) % n]) L++;
    lcp[p[i]] = L ? L-- : L;
  }
  lcp[0] = 0;
}

// Example (comment out in production):
// int main() {
//   s = "banana"; n = si(s);
//   computeSA();
//   computeLCP();
//   // r[k] is the k-th suffix index in sorted order
//   // p[i] is the rank of suffix starting at i
//   // lcp[k] = LCP between suffixes r[k-1] and r[k]
//   return 0;
// }
```

3 Graphs

3.1 Fast Dijkstra's algorithm

```cpp
// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

#include <queue>
#include <cstdio>

using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;

int main() {

  int N, s, t;
  scanf("%d%d%d", &N, &s, &t);
  vector<vector<PII> > edges(N);
  for (int i = 0; i < N; i++) {
    int M;
    scanf("%d", &M);
    for (int j = 0; j < M; j++) {
      int vertex, dist;
      scanf("%d%d", &vertex, &dist);
      edges[i].push_back(make_pair(dist, vertex)); // note order of arguments here
    }
  }

  // use priority queue in which top element has the "smallest" priority
  priority_queue<PII, vector<PII>, greater<PII> > Q;
  vector<int> dist(N, INF), dad(N, -1);
  Q.push(make_pair(0, s));
  dist[s] = 0;
  while (!Q.empty()) {
    PII p = Q.top();
    Q.pop();
    int here = p.second;
    if (here == t) break;
    if (dist[here] != p.first) continue;

    for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it
        ++) {
      if (dist[here] + it->first < dist[it->second]) {
        dist[it->second] = dist[here] + it->first;
        dad[it->second] = here;
        Q.push(make_pair(dist[it->second], it->second));
      }
    }
  }

  printf("%d\n", dist[t]);
  if (dist[t] < INF)
    for (int i = t; i != -1; i = dad[i])
      printf("%d%c", i, (i == s ? '\n' : ' '));
  return 0;
}

/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1

Expected:
5
4 2 3 0
*/
```

3.2 0-1 BFS (shortest paths 0/1 weights)

```cpp
/*
  0-1 BFS on graph with edge weights in {0,1}
  - Computes shortest distances in O(V + E)
*/

#include <bits/stdc++.h>
using namespace std;

vector<int> bfs01(int n, const vector<vector<pair<int,int>>>& adj, int s) {
  const int INF = 1e9; deque<int> dq; vector<int> dist(n, INF); dist[s]=0; dq.
      push_front(s);
  while(!dq.empty()){
    int u=dq.front(); dq.pop_front();
    for(auto [v,w]: adj[u]){
      int nd = dist[u] + (w?1:0);
      if(nd < dist[v]){ dist[v]=nd; if(w) dq.push_back(v); else dq.push_front(v);}
    }
  }
  return dist;
}
```

3.3 Strongly connected components

```cpp
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
  int i;
  v[x]=true;
  for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
  stk[++stk[0]]=x;
}
void fill_backward(int x)
{
  int i;
  v[x]=false;
  group_num[x]=group_cnt;
  for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2
{
  e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
  er[  E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC()
{
  int i;
  stk[0]=0;
  memset(v, false, sizeof(v));
  for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
  group_cnt=0;
  for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}
```

3.4 Eulerian path

```cpp
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
  int next_vertex;
  iter reverse_edge;

  Edge(int next_vertex)
```

```cpp
    :next_vertex(next_vertex)
    { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices];   // adjacency list

vector<int> path;

void find_path(int v)
{
  while(adj[v].size() > 0)
  {
    int vn = adj[v].front().next_vertex;
    adj[vn].erase(adj[v].front().reverse_edge);
    adj[v].pop_front();
    find_path(vn);
  }
  path.push_back(v);
}

void add_edge(int a, int b)
{
  adj[a].push_front(Edge(b));
  iter ita = adj[a].begin();
  adj[b].push_front(Edge(a));
  iter itb = adj[b].begin();
  ita->reverse_edge = itb;
  itb->reverse_edge = ita;
}
```

4 Graphs++

4.1 Sparse max-flow

```cpp
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).

#include<cstdio>
#include<vector>
#include<queue>
using namespace std;
typedef long long LL;

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

struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;

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

  void AddEdge(int u, int v, LL cap) {
    if (u != v) {
      E.emplace_back(u, v, cap);
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(v, u, 0);
      g[v].emplace_back(E.size() - 1);
    }
  }

  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k: g[u]) {
        Edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.emplace(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }

  LL DFS(int u, int T, LL flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      Edge &e = E[g[u][i]];
      Edge &oe = E[g[u][i]^1];
      if (d[e.v] == d[e.u] + 1) {
        LL amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (LL pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  LL MaxFlow(int S, int T) {
    LL total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (LL flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main()
{
  int N, E;
  scanf("%d%d", &N, &E);
  Dinic dinic(N);
  for(int i = 0; i < E; i++)
  {
    int u, v;
    LL cap;
    scanf("%d%d%lld", &u, &v, &cap);
    dinic.AddEdge(u - 1, v - 1, cap);
    dinic.AddEdge(v - 1, u - 1, cap);
  }
  printf("%lld\n", dinic.MaxFlow(0, N - 1));
  return 0;
}

// END CUT
```

4.2 Push-relabel max-flow

```cpp
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
```

```
  queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
  }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
  dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }

  void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)
  Gap(dist[v]);
      else
  Relabel(v);
    }
  }

  LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  PushRelabel pr(n);
  for (int i = 0; i < m; i++) {
   int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    pr.AddEdge(a-1, b-1, c);
    pr.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", pr.GetMaxFlow(0, n-1));
  return 0;
}

// END CUT
```

4.3 Global min-cut

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
  if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
  for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
  for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
  used[last] = true;
  cut.push_back(last);
  if (best_weight == -1 || w[last] < best_weight) {
    best_cut = cut;
    best_weight = w[last];
  }
      } else {
  for (int j = 0; j < N; j++)
    w[j] += weights[last][j];
  added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
  int N;
  cin >> N;
  for (int i = 0; i < N; i++) {
    int n, m;
    cin >> n >> m;
```

```
    VVI weights(n, VI(n));
    for (int j = 0; j < m; j++) {
      int a, b, c;
      cin >> a >> b >> c;
      weights[a-1][b-1] = weights[b-1][a-1] = c;
    }
    pair<int, VI> res = GetMinCut(weights);
    cout << "Case #" << i+1 << ": " << res.first << endl;
  }
}
// END CUT
```

4.4 Graph cut inference

```
// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//        minimize         sum_i  psi_i(x[i])
//  x[1]...x[n] in {0,1}      + sum_{i < j}  phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)  (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method.  To perform maximization instead of minimization,
// ensure that #define MAXIMIZATION is enabled.

#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
typedef vector<VVVI> VVVVI;

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
  int N;
  VVI cap, flow;
  VI reached;

  int Augment(int s, int t, int a) {
    reached[s] = 1;
    if (s == t) return a;
    for (int k = 0; k < N; k++) {
      if (reached[k]) continue;
      if (int aa = min(a, cap[s][k] - flow[s][k])) {
  if (int b = Augment(k, t, aa)) {
    flow[s][k] += b;
    flow[k][s] -= b;
    return b;
  }
      }
    }
    return 0;
  }

  int GetMaxFlow(int s, int t) {
    N = cap.size();
    flow = VVI(N, VI(N));
    reached = VI(N);

    int totflow = 0;
    while (int amt = Augment(s, t, INF)) {
      totflow += amt;
      fill(reached.begin(), reached.end(), 0);
    }
    return totflow;
  }

  int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
      b[i] += psi[i][1] - psi[i][0];
      c += psi[i][0];
      for (int j = 0; j < i; j++)
  b[i] += phi[i][j][1][1] - phi[i][j][0][1];
      for (int j = i+1; j < M; j++) {
  cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] - phi[i][j][1][1];
  b[i] += phi[i][j][1][0] - phi[i][j][0][0];
  c += phi[i][j][0][0];
      }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
      for (int j = i+1; j < M; j++)
  cap[i][j] *= -1;
      b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
      if (b[i] >= 0) {
  cap[M][i] = b[i];
      } else {
  cap[i][M+1] = -b[i];
  c += b[i];
      }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

    return score;
  }

};

int main() {

  // solver for "Cat vs. Dog" from NWERC 2008

  int numcases;
  cin >> numcases;
  for (int caseno = 0; caseno < numcases; caseno++) {
    int c, d, v;
    cin >> c >> d >> v;

    VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));
    VVI psi(c+d, VI(2));
    for (int i = 0; i < v; i++) {
      char p, q;
      int u, v;
      cin >> p >> u >> q >> v;
      u--; v--;
      if (p == 'C') {
  phi[u][c+v][0][0]++;
  phi[c+v][u][0][0]++;
      } else {
  phi[v][c+u][1][1]++;
  phi[c+u][v][1][1]++;
```

```
      }
    }

    GraphCutInference graph;
    VI x;
    cout << graph.DoInference(phi, psi, x) << endl;
  }

  return 0;
}
```

4.5 2-SAT (implication graph + SCC)

```
/*
  2-SAT using implication graph + SCC (Kosaraju)
  - n variables x in {0,1}. Index each literal as var*2 (false), var*2^1 (true)
  - add_imp(a, b): add implication a -> b (a,b are literal indices)
  - add_or(a, b): clause (a or b)
  - add_xor(a, b): (a xor b)
  - add_eq(a, b): a == b
  - solve(): returns {sat, assignment}
*/

#include <bits/stdc++.h>
using namespace std;

struct TwoSAT {
  int n; vector<vector<int>> g, gr; vector<int> comp, order, val;
  TwoSAT(int n=0){init(n);} void init(int n_){n=n_; g.assign(2*n,{}); gr.assign(2*n,{});}
  static inline int var(int x, bool truth){ return x<<1 | (truth?1:0); }
  void add_imp(int a, int b){ g[a].push_back(b); gr[b].push_back(a);}
  void add_or(int a, int b){ // (a or b) => (-a -> b) and (-b -> a)
    add_imp(a^1, b); add_imp(b^1, a);
  }
  void add_xor(int a, int b){ // (a xor b) => (a or b) and (-a or -b)
    add_or(a, b); add_or(a^1, b^1);
  }
  void add_eq(int a, int b){ // a == b => (a->b) and (b->a) and (-a->-b) and (-b->-a)
    add_imp(a, b); add_imp(b, a); add_imp(a^1, b^1); add_imp(b^1, a^1);
  }
  pair<bool, vector<int>> solve(){
    int N=2*n; vector<char> used(N,0); order.clear(); comp.assign(N,-1);
    function<void(int)> dfs1=[&](int v){ used[v]=1; for(int to:g[v]) if(!used[to]) dfs1(to); order.push_back(v); };
    function<void(int,int)> dfs2=[&](int v,int c){ comp[v]=c; for(int to:gr[v]) if(comp[to]==-1) dfs2(to,c); };
    for(int i=0;i<N;++i) if(!used[i]) dfs1(i);
    int j=0; for(int i=N-1;i>=0;--i){ int v=order[i]; if(comp[v]==-1) dfs2(v,j++); }
    val.assign(n,0);
    for(int i=0;i<n;++i){ if(comp[2*i]==comp[2*i+1]) return {false,{}}; val[i]= comp[2*i] < comp[2*i+1]; }
    return {true, val};
  }
};
```

4.6 Hopcroft-Karp (matching bipartito)

```
/*
  Hopcroft-Karp (Maximum Bipartite Matching)
  - Left part: 0..nL-1, Right part: 0..nR-1
  - addEdge(u, v): u in [0, nL), v in [0, nR)
  - maxMatching(): returns maximum matching size
  Complexity: O(E * sqrt(V)).
*/

#include <bits/stdc++.h>
using namespace std;

struct HopcroftKarp {
  int nL, nR;
  vector<vector<int>> adj; // adj[u] -> neighbors v on right
  vector<int> matchL, matchR, dist;

  HopcroftKarp(int nL, int nR) : nL(nL), nR(nR), adj(nL), matchL(nL, -1), matchR(nR, -1), dist(nL, -1) {}

  void addEdge(int u, int v) { // 0 <= u < nL, 0 <= v < nR
    adj[u].push_back(v);
  }

  bool bfs() {
    queue<int> q;
    fill(dist.begin(), dist.end(), -1);
    bool reachableFreeRight = false;
    for (int u = 0; u < nL; ++u) if (matchL[u] == -1) { dist[u] = 0; q.push(u); }
    while (!q.empty()) {
      int u = q.front(); q.pop();
      for (int v : adj[u]) {
        int u2 = matchR[v];
        if (u2 == -1) reachableFreeRight = true; // we can end on a free right node
        else if (dist[u2] == -1) { dist[u2] = dist[u] + 1; q.push(u2); }
      }
    }
    return reachableFreeRight;
  }

  bool dfs(int u) {
    for (int v : adj[u]) {
      int u2 = matchR[v];
      if (u2 == -1 || (dist[u2] == dist[u] + 1 && dfs(u2))) {
        matchL[u] = v; matchR[v] = u; return true;
      }
    }
    dist[u] = -1; // mark as dead end in this layering
    return false;
  }

  int maxMatching() {
    int res = 0;
    while (bfs()) {
      for (int u = 0; u < nL; ++u)
        if (matchL[u] == -1 && dfs(u)) ++res;
    }
    return res;
  }
};

// Example usage:
// int main() {
//   int nL = 3, nR = 3; HopcroftKarp HK(nL, nR);
//   HK.addEdge(0, 0); HK.addEdge(0, 1); HK.addEdge(1, 1); HK.addEdge(2, 2);
//   cout << HK.maxMatching() << "\n"; // expected 3
// }
```

4.7 Hungarian (assignment problem)

```
/*
  Hungarian algorithm (assignment problem, min-cost perfect matching on bipartite graph)
  Complexity: O(n^2 m) for n x m cost matrix; O(n^3) for square matrix.

  Interface:
    - Hungarian hung(n, m);            // n rows (left), m cols (right)
    - hung.addCost(i, j, cost);        // 0 <= i < n, 0 <= j < m
    - auto [minCost, matchR] = hung.solve();
        * minCost: minimal total cost (long long)
        * matchR: size m, matchR[j] = i matched to column j, or -1

  Notes:
    - If m < n, pad with dummy columns of zero cost or swap sides.
    - This implementation works for any rectangular matrix.
*/

#include <bits/stdc++.h>
using namespace std;

struct Hungarian {
  int n, m;                        // rows (left), cols (right)
  const long long INF = (1LL<<62);
```

```cpp
  vector<vector<long long>> a;      // costs

  Hungarian(int n, int m) : n(n), m(m), a(n, vector<long long>(m, 0)) {}
  void addCost(int i, int j, long long c) { a[i][j] = c; }

  pair<long long, vector<int>> solve() {
    // Implementation with potentials (u, v) and matching p/way (cols indexed 1..m)
    // Converts to 1-indexed per classic formulation
    int n1 = n, m1 = m;
    int N = max(n1, m1);
    vector<long long> u(N + 1, 0), v(N + 1, 0);
    vector<int> p(N + 1, 0), way(N + 1, 0);

    // Build square matrix by padding with zeros if needed
    vector<vector<long long>> cost(N + 1, vector<long long>(N + 1, 0));
    for (int i = 1; i <= n1; ++i)
      for (int j = 1; j <= m1; ++j)
        cost[i][j] = a[i-1][j-1];

    for (int i = 1; i <= N; ++i) {
      p[0] = i; int j0 = 0; vector<long long> minv(N + 1, INF); vector<char> used(N + 1, false);
      do {
        used[j0] = true; int i0 = p[j0]; long long delta = INF; int j1 = 0;
        for (int j = 1; j <= N; ++j) if (!used[j]) {
          long long cur = cost[i0][j] - u[i0] - v[j];
          if (cur < minv[j]) { minv[j] = cur; way[j] = j0; }
          if (minv[j] < delta) { delta = minv[j]; j1 = j; }
        }
        for (int j = 0; j <= N; ++j) {
          if (used[j]) { u[p[j]] += delta; v[j] -= delta; }
          else minv[j] -= delta;
        }
        j0 = j1;
      } while (p[j0] != 0);
      do { int j1 = way[j0]; p[j0] = p[j1]; j0 = j1; } while (j0);
    }

    vector<int> matchR(m, -1); // column j matched to row i
    vector<int> matchL(n, -1);
    for (int j = 1; j <= N; ++j) if (p[j] != 0) {
      int i = p[j];
      if (j <= m && i <= n) { matchR[j-1] = i-1; matchL[i-1] = j-1; }
    }
    long long minCost = 0;
    for (int j = 0; j < m; ++j) if (matchR[j] != -1) minCost += a[matchR[j]][j];
    return {minCost, matchR};
  }
};

// Example usage:
// int main() {
//   int n = 3, m = 3; Hungarian H(n, m);
//   long long C[3][3] = {{4,1,3},{2,0,5},{3,2,2}};
//   for (int i=0;i<n;++i) for (int j=0;j<m;++j) H.addCost(i,j,C[i][j]);
//   auto [cost, matchR] = H.solve();
//   cout << cost << "\n"; // expected 5
//   // matchR[j] = i
// }
```

4.8 Min-cost max-flow (potenciales)

```cpp
/*
  Min-Cost Max-Flow (successive shortest augmenting paths)
  - Dijkstra on residual graph with Johnson potentials to handle costs >= any value
  - Interface:
      MinCostMaxFlow mcmf(n);
      mcmf.add_edge(u, v, cap, cost);  // directed edge
      auto [flow, cost] = mcmf.min_cost_max_flow(s, t);
  - Complexity: roughly O(F * E log V), fast in practice.
*/

#include <bits/stdc++.h>
using namespace std;

struct MinCostMaxFlow {
  struct Edge { int to, rev; long long cap, cost; };
  int n; vector<vector<Edge>> g;
  MinCostMaxFlow(int n = 0) { init(n); }
  void init(int n_) { n = n_; g.assign(n, {}); }

  void add_edge(int u, int v, long long cap, long long cost) {
    Edge a{v, (int)g[v].size(), cap, cost};
    Edge b{u, (int)g[u].size(), 0,  -cost};
    g[u].push_back(a); g[v].push_back(b);
  }

  pair<long long,long long> min_cost_max_flow(int s, int t) {
    const long long INF = (1LL<<62);
    long long flow = 0, cost = 0;
    vector<long long> dist(n), pot(n, 0), add(n);
    vector<int> pv_v(n), pv_e(n);

    auto dijkstra = [&]()->bool {
      fill(dist.begin(), dist.end(), INF);
      dist[s] = 0;
      priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<pair<long long,int>>> pq;
      pq.push({0, s});
      while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d != dist[u]) continue;
        for (int i = 0; i < (int)g[u].size(); ++i) {
          const Edge &e = g[u][i]; if (e.cap <= 0) continue;
          long long w = e.cost + pot[u] - pot[e.to];
          if (dist[e.to] > d + w) {
            dist[e.to] = d + w; pv_v[e.to] = u; pv_e[e.to] = i;
            pq.push({dist[e.to], e.to});
          }
        }
      }
      if (dist[t] == INF) return false;
      for (int v = 0; v < n; ++v) if (dist[v] < INF) pot[v] += dist[v];
      return true;
    };

    while (dijkstra()) {
      long long push = INF;
      for (int v = t; v != s; v = pv_v[v]) {
        Edge &e = g[pv_v[v]][pv_e[v]]; push = min(push, e.cap);
      }
      for (int v = t; v != s; v = pv_v[v]) {
        Edge &e = g[pv_v[v]][pv_e[v]]; Edge &er = g[v][e.rev];
        e.cap -= push; er.cap += push; cost += push * e.cost;
      }
      flow += push;
    }
    return {flow, cost};
  }
};

// Example usage:
// int main() {
//   MinCostMaxFlow mf(4);
//   mf.add_edge(0,1,1,1);
//   mf.add_edge(0,2,1,2);
//   mf.add_edge(1,3,1,1);
//   mf.add_edge(2,3,1,1);
//   auto [f, c] = mf.min_cost_max_flow(0,3); // f=2, c=3
//   cout << f << " " << c << "\n";
// }
```

5 Dynamic programming

5.1 Longest increasing subsequence (strict and non-strict)

```cpp
// Longest Increasing Subsequence (LIS) with reconstruction
// - Time: O(n log n)
// - Provides strict and non-strict variants
//   * LongestIncreasingSubsequence(v): strictly increasing
//   * LongestIncreasingSubsequenceNonStrict(v): non-decreasing

#include <bits/stdc++.h>

using namespace std;

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

static VI LIS_impl(const VI& v, bool strict) {
  VPII best;              // stores pairs (value, index)
  VI dad(v.size(), -1);   // predecessor to reconstruct sequence

  for (int i = 0; i < (int)v.size(); i++) {
    PII key;
    if (strict) {
      // lower_bound on (value, -inf) for strictly increasing
      key = make_pair(v[i], INT_MIN);
      VPII::iterator it = lower_bound(best.begin(), best.end(), key);
      PII item = make_pair(v[i], i);
      if (it == best.end()) {
        dad[i] = best.empty() ? -1 : best.back().second;
        best.push_back(item);
      } else {
        dad[i] = (it == best.begin()) ? -1 : prev(it)->second;
        *it = item;
      }
    } else {
      // upper_bound on (value, +inf) for non-decreasing
      key = make_pair(v[i], INT_MAX);
      VPII::iterator it = upper_bound(best.begin(), best.end(), key);
      PII item = make_pair(v[i], i);
      if (it == best.end()) {
        dad[i] = best.empty() ? -1 : best.back().second;
        best.push_back(item);
      } else {
        dad[i] = (it == best.begin()) ? -1 : prev(it)->second;
        *it = item;
      }
    }
  }

  VI ret;
  if (best.empty()) return ret;
  for (int i = best.back().second; i >= 0; i = dad[i]) ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}

VI LongestIncreasingSubsequence(VI v) {
  return LIS_impl(v, true);
}

VI LongestIncreasingSubsequenceNonStrict(VI v) {
  return LIS_impl(v, false);
}
```

5.2 Maximum subarray sum (Kadane)

```cpp
/*
  Kadane's Algorithm - Maximum Subarray Sum (O(n))
  - kadane(a): returns maximum subarray sum (works when all numbers are negative)
  - kadane_with_indices(a): returns {best_sum, [l, r]} for subarray a[l..r] inclusive
*/

#include <bits/stdc++.h>
using namespace std;

long long kadane(const vector<long long>& a) {
  long long best = LLONG_MIN, cur = 0;
  for (long long x : a) {
    cur = max(x, cur + x);
    best = max(best, cur);
  }
  return best;
}

pair<long long, pair<int,int>> kadane_with_indices(const vector<long long>& a) {
  long long best = LLONG_MIN, cur = 0;
  int bestL = 0, bestR = -1, curL = 0;
  for (int i = 0; i < (int)a.size(); ++i) {
    if (cur + a[i] < a[i]) { cur = a[i]; curL = i; }
    else { cur += a[i]; }
    if (cur > best) { best = cur; bestL = curL; bestR = i; }
  }
  return {best, {bestL, bestR}};
}

// Example usage:
// int main(){
//   vector<long long> a = {-2,1,-3,4,-1,2,1,-5,4};
//   auto res = kadane_with_indices(a);
//   cout << res.first << " [" << res.second.first << "," << res.second.second << "]\n"; // 6 [3,6]
// }
```

5.3 Subset sum (0/1 decision)

```cpp
/*
  Subset Sum (0/1)
  - can_sum(v, target): returns true if some subset sums to target
  - Optional reconstruction is easy by tracking choices; here we return bool only.
*/

#include <bits/stdc++.h>
using namespace std;

bool can_sum(const vector<int>& v, int target) {
  vector<char> dp(target + 1, 0);
  dp[0] = 1;
  for (int a : v) {
    for (int s = target; s >= a; --s) dp[s] = dp[s] || dp[s - a];
  }
  return dp[target];
}

// Example usage:
// int main(){
//   vector<int> v = {3, 34, 4, 12, 5, 2};
//   cout << can_sum(v, 9) << "\n"; // 1
// }
```

5.4 0/1 knapsack (max value)

```cpp
/*
  0/1 Knapsack
  - knap01(weights, values, W): maximum value with total weight <= W
  - Complexity: O(n * W)
*/

#include <bits/stdc++.h>
using namespace std;

long long knap01(const vector<int>& w, const vector<int>& v, int W) {
  int n = (int)w.size();
  vector<long long> dp(W + 1, 0);
  for (int i = 0; i < n; ++i) {
    for (int c = W; c >= w[i]; --c) dp[c] = max(dp[c], dp[c - w[i]] + v[i]);
```

```
  }
  return dp[W];
}

// Example usage:
// int main(){
//   vector<int> w = {3,2,1}, v = {5,3,4}; int W = 5;
//   cout << knap01(w, v, W) << "\n"; // 9
// }
```

5.5 Coin change (ways and min coins)

```
/*
  Coin Change problems
  - ways_unbounded(amount, coins): number of ways to make amount with unlimited coins (order-insensitive)
  - min_coins_unbounded(amount, coins): minimum number of coins to make amount (INF if impossible)
*/

#include <bits/stdc++.h>
using namespace std;

static const long long INF64 = (1LL<<60);

long long ways_unbounded(int amount, const vector<int>& coins) {
  vector<long long> dp(amount + 1, 0);
  dp[0] = 1;
  for (int c : coins) {
    for (int x = c; x <= amount; ++x) dp[x] += dp[x - c];
  }
  return dp[amount];
}

long long min_coins_unbounded(int amount, const vector<int>& coins) {
  vector<long long> dp(amount + 1, INF64);
  dp[0] = 0;
  for (int x = 1; x <= amount; ++x) {
    for (int c : coins) if (c <= x) dp[x] = min(dp[x], dp[x - c] + 1);
  }
  return dp[amount] >= INF64 ? -1 : dp[amount];
}

// Example usage:
// int main(){
//   vector<int> coins = {1, 2, 5};
//   cout << ways_unbounded(10, coins) << "\n"; // 10
//   cout << min_coins_unbounded(11, coins) << "\n"; // 3 (5+5+1)
// }
```

5.6 Digit DP (sum of digits divisible by m)

```
/*
  Digit DP templates
  - count_sum_divisible(R, mod): count x in [0, R] such that sum of digits(x) % mod == 0
  - count_sum_divisible(L, R, mod): count x in [L, R] with same property

  Notes:
  - Works for 0 <= R up to 10^18 (or longer if you pass a longer string)
  - Complexity: O(len * mod * 10)
*/

#include <bits/stdc++.h>
using namespace std;

static long long solve_sum_divisible(const string &s, int mod) {
  int n = (int)s.size();
  // dp[pos][sum_mod][tight][started]
  vector dp(n + 1, vector(mod, array<array<long long, 2>, 2>{}));
  vector vis(n + 1, vector(mod, array<array<char, 2>, 2>{}));

  function<long long(int,int,int,int)> dfs = [&](int pos, int sum, int tight, int started) -> long long {
    if (pos == n) return started ? (sum % mod == 0) : 1; // treat 0 as valid (sum 0)
    if (vis[pos][sum][tight][started]) return dp[pos][sum][tight][started];
    vis[pos][sum][tight][started] = 1;
    int limit = tight ? (s[pos] - '0') : 9;
    long long res = 0;
    for (int d = 0; d <= limit; ++d) {
      int ntight = tight && (d == limit);
      int nstarted = started || (d != 0);
      int nsum = sum;
      if (nstarted) nsum = (nsum + d) % mod;
      else nsum = 0; // still leading zeros -> sum stays 0
      res += dfs(pos + 1, nsum, ntight, nstarted);
    }
    return dp[pos][sum][tight][started] = res;
  };

  return dfs(0, 0, 1, 0);
}

long long count_sum_divisible(long long R, int mod) {
  if (R < 0) return 0;
  string s = to_string(R);
  return solve_sum_divisible(s, mod);
}

long long count_sum_divisible(long long L, long long R, int mod) {
  if (L > R) return 0;
  return count_sum_divisible(R, mod) - count_sum_divisible(L - 1, mod);
}

// Example usage:
// int main(){
//   // Count numbers in [1, 1000] whose sum of digits is divisible by 3
//   cout << count_sum_divisible(1, 1000, 3) << "\n";
// }
```

5.7 Dice sums probabilities (DP)

```
/*
  Probability DP: Dice sums
  - dice_sum_pmf(n, faces): returns vector p where p[s] = P(sum == s) for n fair dice with 'faces' faces (values 1..faces)
  - prob_sum_at_least(n, faces, thr): returns P(sum >= thr)
  Complexity: O(n * faces * max_sum)
*/

#include <bits/stdc++.h>
using namespace std;

vector<double> dice_sum_pmf(int n, int faces = 6) {
  if (n <= 0) return {1.0};
  int minS = n, maxS = n * faces;
  vector<double> dp_prev(maxS + 1, 0.0), dp_cur(maxS + 1, 0.0);
  // one die
  for (int d = 1; d <= faces; ++d) dp_prev[d] = 1.0 / faces;
  for (int k = 2; k <= n; ++k) {
    fill(dp_cur.begin(), dp_cur.end(), 0.0);
    for (int s = (k - 1); s <= (k - 1) * faces; ++s) if (dp_prev[s] > 0) {
      for (int d = 1; d <= faces; ++d) dp_cur[s + d] += dp_prev[s] / faces;
    }
    dp_prev.swap(dp_cur);
  }
  return vector<double>(dp_prev.begin(), dp_prev.end());
}

double prob_sum_at_least(int n, int faces, int thr) {
  auto p = dice_sum_pmf(n, faces);
  int maxS = n * faces;
  thr = max(thr, 0);
  if (thr > maxS) return 0.0;
  double ans = 0.0;
  for (int s = thr; s < (int)p.size(); ++s) ans += p[s];
  return ans;
}

// Example usage:
```

```
// int main(){
//   cout.setf(std::ios::fixed); cout<<setprecision(6);
//   cout << prob_sum_at_least(3, 6, 10) << "\n";
// }
```

5.8 Expected rolls to reach target

```
/*
  Expected Value DP: expected number of die rolls to reach or exceed N
  - expected_rolls_to_reach(N, faces): E[pos] satisfies
      E[N] = 0, and for 0 <= x < N:
      E[x] = 1 + (1/faces) * sum_{d=1..faces} E[min(x + d, N)]
  Returns E[0].
  Complexity: O(N * faces)
*/

#include <bits/stdc++.h>
using namespace std;

double expected_rolls_to_reach(int N, int faces = 6) {
  vector<double> E(N + 1, 0.0);
  for (int x = N - 1; x >= 0; --x) {
    double sum = 0.0;
    for (int d = 1; d <= faces; ++d) {
      int nx = x + d; if (nx > N) nx = N; // cap at N
      sum += E[nx];
    }
    E[x] = 1.0 + sum / faces;
  }
  return E[0];
}

// Example usage:
// int main(){ cout.setf(std::ios::fixed); cout<<setprecision(6);
//   cout << expected_rolls_to_reach(100) << "\n"; }
```

5.9 Markov chains (small state DP)

```
/*
  Markov Chains (small state) utilities
  - step_distribution(pi0, T, steps): apply transition matrix T 'steps' times to initial distribution pi0
  - stationary_distribution(T): power iteration to approximate stationary vector for an ergodic chain
  Assumes T is row-stochastic (rows sum to 1). Vectors are probabilities.
*/

#include <bits/stdc++.h>
using namespace std;

using Vec = vector<double>;
using Mat = vector<vector<double>>;

static Vec mul(const Vec& v, const Mat& T) {
  int n = (int)T.size();
  Vec r(n, 0.0);
  for (int i = 0; i < n; ++i) if (v[i] != 0.0) {
    for (int j = 0; j < n; ++j) r[j] += v[i] * T[i][j];
  }
  return r;
}

Vec step_distribution(const Vec& pi0, const Mat& T, long long steps) {
  Vec pi = pi0; long long k = max(0LL, steps);
  while (k--) pi = mul(pi, T);
  return pi;
}

Vec stationary_distribution(const Mat& T, int iters = 10000, double tol = 1e-12) {
  int n = (int)T.size();
  Vec pi(n, 1.0 / n);
  for (int it = 0; it < iters; ++it) {
    Vec nxt = mul(pi, T);
    double diff = 0.0; for (int i = 0; i < n; ++i) diff = max(diff, fabs(nxt[i] - pi[i]));
    pi.swap(nxt);
    if (diff < tol) break;
  }
  return pi;
}

// Example usage:
// int main(){
//   Mat T = { {0.9, 0.1}, {0.5, 0.5} };
//   Vec pi0 = {1, 0};
//   auto pi = step_distribution(pi0, T, 10);
//   auto st = stationary_distribution(T);
//   cout.setf(std::ios::fixed); cout<<setprecision(6);
//   cout << pi[0] << " " << pi[1] << "\n";
//   cout << st[0] << " " << st[1] << "\n";
// }
```

6 Numerical algorithms

6.1 Number theory (modular, Chinese remainder, linear Diophantine)

```cpp
// This is a collection of useful code for solving problems that
// involve modular linear equations.  Note that all of the
// algorithms described here work on nonnegative integers.

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

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
  while (b) { int t = a%b; a = b; b = t; }
  return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
  return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
  int ret = 1;
  while (b)
  {
    if (b & 1) ret = mod(ret*a, m);
    a = mod(a*a, m);
    b >>= 1;
  }
  return ret;
}

// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a / b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x - q*xx; x = t;
    t = yy; yy = y - q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI ret;
  int g = extended_euclid(a, n, x, y);
  if (!(b%g)) {
    x = mod(x*(b / g), n);
    for (int i = 0; i < g; i++)
      ret.push_back(mod(x + i*(n / g), n));
  }
  return ret;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int g = extended_euclid(a, n, x, y);
  if (g > 1) return -1;
  return mod(x, n);
}

// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2.  Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M).  On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
  int s, t;
  int g = extended_euclid(m1, m2, s, t);
  if (r1%g != r2%g) return make_pair(0, -1);
  return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &m, const VI &r) {
  PII ret = make_pair(r[0], m[0]);
  for (int i = 1; i < m.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
  if (!a && !b)
  {
    if (c) return false;
    x = 0; y = 0;
    return true;
  }
  if (!a)
  {
    if (c % b) return false;
    x = 0; y = c / b;
    return true;
  }
  if (!b)
  {
    if (c % a) return false;
    x = c / a; y = 0;
    return true;
  }
  int g = gcd(a, b);
  if (c % g) return false;
  x = c / g * mod_inverse(a / g, b / g);
  y = (c - a*x) / b;
  return true;
}

int main() {
  // expected: 2
  cout << gcd(14, 30) << endl;

  // expected: 2 -2 1
  int x, y;
  int g = extended_euclid(14, 30, x, y);
  cout << g << " " << x << " " << y << endl;

  // expected: 95 451
  VI sols = modular_linear_equation_solver(14, 30, 100);
  for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
  cout << endl;

  // expected: 8
  cout << mod_inverse(8, 9) << endl;

  // expected: 23 105
  //           11 12
  PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2 }));
  cout << ret.first << " " << ret.second << endl;
  ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
  cout << ret.first << " " << ret.second << endl;

  // expected: 5 -15
  if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
  cout << x << " " << y << endl;
  return 0;
}
```

6.2 Systems of linear equations, matrix inverse, determinant

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
  if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

6.3 Reduced row echelon form, matrix rank

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
  int n = a.size();
  int m = a[0].size();
  int r = 0;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r + 1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);

    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}

int main() {
  const int n = 5, m = 4;
  double A[n][m] = {
```

```cpp
    {16,  2,  3, 13},
    { 5, 11, 10,  8},
    { 9,  7,  6, 12},
    { 4, 14, 15,  1},
    {13, 21, 21, 13}};
  VVT a(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + m);

  int rank = rref(a);

  // expected: 3
  cout << "Rank: " << rank << endl;

  // expected: 1 0 0 1
  //           0 1 0 3
  //           0 0 1 -3
  //           0 0 0 3.10862e-15
  //           0 0 0 2.22045e-15
  cout << "rref: " << endl;
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 4; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }
}
```

6.4 Fast Fourier transform

```cpp
#include <cassert>
#include <cstdio>
#include <cmath>

struct cpx
{
  cpx(){}
  cpx(double aa):a(aa),b(0){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const
  {
    return a * a + b * b;
  }
  cpx bar(void) const
  {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b)
{
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta)
{
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j * k / size)
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

int main(void)
{
  printf("If rows come in identical pairs, then everything works.\n");

  cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};
  cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};
  cpx A[8];
  cpx B[8];
  FFT(a, A, 1, 8, 1);
  FFT(b, B, 1, 8, 1);

  for(int i = 0 ; i < 8 ; i++)
  {
    printf("%7.2lf%7.2lf", A[i].a, A[i].b);
  }
  printf("\n");
  for(int i = 0 ; i < 8 ; i++)
  {
    cpx Ai(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
      Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
    }
    printf("%7.2lf%7.2lf", Ai.a, Ai.b);
  }
  printf("\n");

  cpx AB[8];
  for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
  cpx aconvb[8];
  FFT(AB, aconvb, 1, 8, -1);
  for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
  for(int i = 0 ; i < 8 ; i++)
  {
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
  }
  printf("\n");
  for(int i = 0 ; i < 8 ; i++)
  {
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
      aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    }
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
  }
  printf("\n");

  return 0;
}
```

6.5 Simplex algorithm

```cpp
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    double inv = 1.0 / D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] * inv;
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
          (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE>::
          infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j
              ;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);

  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

6.6 Sieve of Eratosthenes (linear, SPF)

```cpp
/*
  Sieve of Eratosthenes (linear) with smallest prime factor (SPF)
  - Time: O(n)
  - Memory: O(n)
  Provides:
    - primes: list of primes up to n
    - spf[x]: smallest prime factor of x (x>=2), or 0 if x<2
    - is_prime(x): quick primality for x<=n
    - factorize(x): factorization for x<=n using spf
*/

#include <bits/stdc++.h>
using namespace std;

struct Sieve {
  int n; vector<int> primes, spf; vector<char> comp;
  Sieve(int n=0) { init(n); }
  void init(int n_) {
    n = n_; primes.clear(); spf.assign(n+1, 0); comp.assign(n+1, 0);
    for (int i = 2; i <= n; ++i) {
      if (!comp[i]) { primes.push_back(i); spf[i] = i; }
      for (int p : primes) {
        long long x = 1LL * p * i; if (x > n) break;
        comp[x] = 1; spf[x] = p;
```

```
        if (i % p == 0) break;
      }
    }
  }
  bool is_prime(int x) const { return x >= 2 && x <= n && !comp[x]; }
  vector<pair<int,int>> factorize(int x) const {
    vector<pair<int,int>> f; if (x < 2) return f;
    while (x > 1) {
      int p = spf[x], c = 0; while (x % p == 0) { x /= p; ++c; }
      f.push_back({p, c});
    }
    return f;
  }
};

// Example usage:
// int main(){ Sieve sv(1000000); cout << sv.primes.size() << "\n"; }
```

6.7 Primality test (deterministic 64-bit)

```
/*
  Deterministic Miller-Rabin for 64-bit integers
  - Time: ~ O(k log^3 n) with fixed bases (k small)
  - For 0 <= n < 2^64, the set of bases {2,3,5,7,11,13,17} is deterministic
*/

#include <bits/stdc++.h>
using namespace std;

using u128 = unsigned __int128;
using u64 = unsigned long long;
using u32 = unsigned int;

static inline u64 mul_mod(u64 a, u64 b, u64 m) {
  return (u128)a * b % m;
}
static inline u64 pow_mod(u64 a, u64 e, u64 m) {
  u64 r = 1;
  while (e) { if (e & 1) r = mul_mod(r, a, m); a = mul_mod(a, a, m); e >>= 1; }
  return r;
}

bool isPrime64(u64 n) {
  if (n < 2) return false;
  for (u64 p : {2ULL,3ULL,5ULL,7ULL,11ULL,13ULL,17ULL}) {
    if (n % p == 0) return n == p;
  }
  u64 d = n - 1, s = 0;
  while ((d & 1) == 0) { d >>= 1; ++s; }
  auto check = [&](u64 a)->bool{
    u64 x = pow_mod(a, d, n);
    if (x == 1 || x == n - 1) return true;
    for (u64 r = 1; r < s; ++r) {
      x = mul_mod(x, x, n);
      if (x == n - 1) return true;
    }
    return false;
  };
  for (u64 a : {2ULL,3ULL,5ULL,7ULL,11ULL,13ULL,17ULL}) if (!check(a)) return false;
  return true;
}

// Example usage:
// int main(){ cout << isPrime64(1000000007ULL) << "\n"; }
```

6.8 Segmented sieve (primes in [L,R])

```
/*
  Segmented Sieve of Eratosthenes
  - Lists primes in an arbitrary interval [L, R] (0 <= L <= R <= 1e18 typical)
  - Uses a simple sieve up to sqrt(R), then marks composites in the segment
  - Memory: O(R-L+1)

  Interface:
    vector<long long> segmented_sieve(long long L, long long R)
*/

#include <bits/stdc++.h>
using namespace std;

static vector<int> simple_sieve(int n) {
  int m = max(2, n);
  vector<char> comp(m + 1, 0);
  vector<int> primes;
  for (int i = 2; 1LL * i * i <= m; ++i)
    if (!comp[i]) for (long long j = 1LL * i * i; j <= m; j += i) comp[(int)j] = 1;
  for (int i = 2; i <= m; ++i) if (!comp[i]) primes.push_back(i);
  return primes;
}

vector<long long> segmented_sieve(long long L, long long R) {
  if (R < 2 || L > R) return {};
  long long s = floor(sqrt((long double)R));
  vector<int> base = simple_sieve((int)s);
  long long len = R - L + 1;
  vector<char> comp(len, 0);

  for (int p : base) {
    long long start = max(1LL * p * p, ((L + p - 1) / p) * 1LL * p);
    for (long long x = start; x <= R; x += p) comp[(int)(x - L)] = 1;
  }

  vector<long long> res;
  for (long long x = max(2LL, L); x <= R; ++x)
    if (!comp[(int)(x - L)]) res.push_back(x);
  return res;
}

// Example usage:
// int main(){
//   auto ps = segmented_sieve(1, 100);
//   for (auto p: ps) cout << p << ' ';
//   cout << "\n";
// }
```

6.9 Matrix exponentiation (power)

```
/*
  Matrix exponentiation (NxN) in O(N^3 log e)
  - mult(A,B,mod), power(A, e, mod)
*/

#include <bits/stdc++.h>
using namespace std;

using Mat = vector<vector<long long>>;

Mat mult(const Mat& A, const Mat& B, long long mod){
  int n=A.size(), m=B[0].size(), p=B.size();
  Mat C(n, vector<long long>(m,0));
  for(int i=0;i<n;++i)
    for(int k=0;k<p;++k) if(A[i][k]){
      long long aik=A[i][k]%mod;
      for(int j=0;j<m;++j) C[i][j]=(C[i][j]+aik*(B[k][j]%mod))%mod;
    }
  return C;
}

Mat power(Mat A, long long e, long long mod){
  int n=A.size(); Mat R(n, vector<long long>(n,0)); for(int i=0;i<n;++i) R[i][i]=1%mod
      ;
  while(e){ if(e&1) R=mult(R,A,mod); A=mult(A,A,mod); e>>=1; }
  return R;
}
```

6.10 Combinatorics nCr mod prime (factorials)

```
/*
  Combinatorics modulo prime: factorials, inverse factorials, nCr
  - Precompute up to N in O(N)
*/

#include <bits/stdc++.h>
using namespace std;

struct CombMod {
  int N; long long MOD; vector<long long> fact, invfact;
  long long modpow(long long a, long long e){ long long r=1%MOD; while(e){ if(e&1) r=r*a%MOD; a=a*a%MOD; e>>=1;} return r; }
  CombMod(int N=0, long long MOD=1000000007LL){ init(N, MOD); }
  void init(int N_, long long MOD_){ N=N_; MOD=MOD_; fact.assign(N+1,1); invfact.assign(N+1,1); for(int i=1;i<=N;++i) fact[i]=fact[i-1]*i%MOD; invfact[N]=modpow(fact[N], MOD-2); for(int i=N;i>0;--i) invfact[i-1]=invfact[i]*i%MOD; }
  long long nCr(int n, int r){ if(r<0||r>n) return 0; return fact[n]*invfact[r]%MOD*invfact[n-r]%MOD; }
};
```

6.11 Simpson integration (composite + adaptive)

```
/*
  Numerical Integration (Simpson's rule)
  - simpson_composite(f, a, b, n): O(n) composite Simpson over n subintervals
    (no need for n even; uses midpoint per subinterval)
  - simpson_adaptive(f, a, b, eps, max_depth): adaptive Simpson with tolerance
    eps and recursion limit max_depth. Works well for smooth functions.
*/

#include <bits/stdc++.h>
using namespace std;

using F = function<double(double)>;

// Composite Simpson: sum over each subinterval using a midpoint
double simpson_composite(const F& f, double a, double b, int n = 10000) {
  if (n <= 0) return 0.0;
  double h = (b - a) / n;
  double area = 0.0;
  double fa = f(a);
  for (int i = 0; i < n; ++i) {
    double left = a + h * i;
    double right = left + h;
    double mid = (left + right) * 0.5;
    double fb = f(right);
    area += fa + 4.0 * f(mid) + fb;
    fa = fb;
  }
  return area * (h / 6.0);
}

// Helper: Simpson estimate on [a, b]
static inline double simpson_interval(const F& f, double a, double b) {
  double c = 0.5 * (a + b);
  return (f(a) + 4.0 * f(c) + f(b)) * (b - a) / 6.0;
}

// Adaptive Simpson recursion
static double adaptive_rec(const F& f, double a, double b, double eps, double S,
                           int depth, int max_depth) {
  double c = 0.5 * (a + b);
  double Sleft = simpson_interval(f, a, c);
  double Sright = simpson_interval(f, c, b);
  double delta = Sleft + Sright - S;
  if (depth >= max_depth || fabs(delta) <= 15.0 * eps) {
    // Richardson extrapolation
    return Sleft + Sright + delta / 15.0;
  }
  return adaptive_rec(f, a, c, eps * 0.5, Sleft, depth + 1, max_depth)
       + adaptive_rec(f, c, b, eps * 0.5, Sright, depth + 1, max_depth);
}

double simpson_adaptive(const F& f, double a, double b, double eps = 1e-9, int max_depth = 20) {
  double S = simpson_interval(f, a, b);
  return adaptive_rec(f, a, b, eps, S, 0, max_depth);
}

// Example usage:
// int main(){
//   auto f = [](double x){ return sin(x); };
//   cout.setf(std::ios::fixed); cout<<setprecision(12);
//   cout << simpson_composite(f, 0, M_PI, 10000) << "\n"; // ~2
//   cout << simpson_adaptive(f, 0, M_PI, 1e-10) << "\n";   // ~2
// }
```

7 Geometry

7.1 Convex hull

```
// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm.  Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//   INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull, counterclockwise, starting
//            with bottommost/leftmost point

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
// BEGIN CUT
#include <map>
// END CUT

using namespace std;

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)

int main() {
  int t;
  scanf("%d", &t);
  for (int caseno = 0; caseno < t; caseno++) {
    int n;
    scanf("%d", &n);
    vector<PT> v(n);
    for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
    vector<PT> h(v);
    map<PT,int> index;
    for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
    ConvexHull(h);

    double len = 0;
    for (int i = 0; i < h.size(); i++) {
      double dx = h[i].x - h[(i+1)%h.size()].x;
      double dy = h[i].y - h[(i+1)%h.size()].y;
      len += sqrt(dx*dx+dy*dy);
    }

    if (caseno > 0) printf("\n");
    printf("%.2f\n", len);
    for (int i = 0; i < h.size(); i++) {
      if (i > 0) printf(" ");
      printf("%d", index[h[i]]);
    }
    printf("\n");
  }
}

// END CUT
```

7.2 Miscellaneous geometry

```
// C++ routines for computational geometry.

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

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  return os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
```

```cpp

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

```

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

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;
```

```cpp

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
       << PointInPolygon(v, PT(2,0)) << " "
       << PointInPolygon(v, PT(0,2)) << " "
       << PointInPolygon(v, PT(5,2)) << " "
       << PointInPolygon(v, PT(2,5)) << endl;

  // expected: 0 1 1 1 1
  cerr << PointOnPolygon(v, PT(2,2)) << " "
       << PointOnPolygon(v, PT(2,0)) << " "
       << PointOnPolygon(v, PT(0,2)) << " "
       << PointOnPolygon(v, PT(5,2)) << " "
       << PointOnPolygon(v, PT(2,5)) << endl;

  // expected: (1,6)
  //           (5,4) (4,5)
  //           blank line
  //           (4,5) (5,4)
  //           blank line
  //           (4,5) (5,4)
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

7.3 Java geometry

```java
// In this example, we read an input file containing three lines, each
// containing an even number of doubles, separated by commas. The first two
// lines represent the coordinates of two polygons, given in counterclockwise
// (or clockwise) order, which we will call "A" and "B". The last line
// contains a list of points, p[1], p[2], ...
//
// Our goal is to determine:
//   (1) whether B - A is a single closed shape (as opposed to multiple shapes)
//   (2) the area of B - A
//   (3) whether each p[i] is in the interior of B - A
//
// INPUT:
//   0 0 10 0 0 10
//   0 0 10 10 10 0
//   8 6
//   5 1
//
// OUTPUT:
//   The area is singular.
//   The area is 25.0
//   Point belongs to the area.
//   Point does not belong to the area.

import java.util.*;
import java.awt.geom.*;
import java.io.*;

public class JavaGeometry {

    // make an array of doubles from a string
    static double[] readPoints(String s) {
        String[] arr = s.trim().split("\\s++");
        double[] ret = new double[arr.length];
        for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);
        return ret;
    }

    // make an Area object from the coordinates of a polygon
    static Area makeArea(double[] pts) {
        Path2D.Double p = new Path2D.Double();
        p.moveTo(pts[0], pts[1]);
        for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);
        p.closePath();
        return new Area(p);
    }

    // compute area of polygon
    static double computePolygonArea(ArrayList<Point2D.Double> points) {
        Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);
        double area = 0;
        for (int i = 0; i < pts.length; i++){
            int j = (i+1) % pts.length;
            area += pts[i].x * pts[j].y - pts[j].x * pts[i].y;
        }
        return Math.abs(area)/2;
    }

    // compute the area of an Area object containing several disjoint polygons
    static double computeArea(Area area) {
        double totArea = 0;
        PathIterator iter = area.getPathIterator(null);
        ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();

        while (!iter.isDone()) {
            double[] buffer = new double[6];
            switch (iter.currentSegment(buffer)) {
            case PathIterator.SEG_MOVETO:
            case PathIterator.SEG_LINETO:
                points.add(new Point2D.Double(buffer[0], buffer[1]));
                break;
            case PathIterator.SEG_CLOSE:
```

```java
                totArea += computePolygonArea(points);
                points.clear();
                break;
            }
            iter.next();
        }
        return totArea;
    }

    // notice that the main() throws an Exception -- necessary to
    // avoid wrapping the Scanner object for file reading in a
    // try { ... } catch block.
    public static void main(String args[]) throws Exception {

        Scanner scanner = new Scanner(new File("input.txt"));
        // also,
        //   Scanner scanner = new Scanner (System.in);

        double[] pointsA = readPoints(scanner.nextLine());
        double[] pointsB = readPoints(scanner.nextLine());
        Area areaA = makeArea(pointsA);
        Area areaB = makeArea(pointsB);
        areaB.subtract(areaA);
        // also,
        //   areaB.exclusiveOr (areaA);
        //   areaB.add (areaA);
        //   areaB.intersect (areaA);

        // (1) determine whether B - A is a single closed shape (as
        //     opposed to multiple shapes)
        boolean isSingle = areaB.isSingular();
        // also,
        //   areaB.isEmpty();

        if (isSingle)
            System.out.println("The area is singular.");
        else
            System.out.println("The area is not singular.");

        // (2) compute the area of B - A
        System.out.println("The area is " + computeArea(areaB) + ".");

        // (3) determine whether each p[i] is in the interior of B - A
        while (scanner.hasNextDouble()) {
            double x = scanner.nextDouble();
            assert(scanner.hasNextDouble());
            double y = scanner.nextDouble();

            if (areaB.contains(x,y)) {
                System.out.println ("Point belongs to the area.");
            } else {
                System.out.println ("Point does not belong to the area.");
            }
        }

        // Finally, some useful things we didn't use in this example:
        //
        //   Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,
        //                                                    double w, double h);
        //
        //     creates an ellipse inscribed in box with bottom-left corner (x,y)
        //     and upper-right corner (x+y,w+h)
        //
        //   Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,
        //                                                     double w, double h);
        //
        //     creates a box with bottom-left corner (x,y) and upper-right
        //     corner (x+y,w+h)
        //
        // Each of these can be embedded in an Area object (e.g., new Area (rect)).

    }
}
```

7.4 3D geometry

```java
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
  public static double ptPlaneDist(double x, double y, double z,
      double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
  }

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

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

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

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

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

7.5 Slow Delaunay triangulation

```cpp
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
//
// OUTPUT:   triples = a vector containing m triples of indices
//                     corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};
```

```

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
  int n = x.size();
  vector<T> z(n);
  vector<triple> ret;

  for (int i = 0; i < n; i++)
      z[i] = x[i] * x[i] + y[i] * y[i];

  for (int i = 0; i < n-2; i++) {
      for (int j = i+1; j < n; j++) {
    for (int k = i+1; k < n; k++) {
        if (j == k) continue;
        double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
        double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
        double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
        bool flag = zn < 0;
        for (int m = 0; flag && m < n; m++)
      flag = flag && ((x[m]-x[i])*xn +
          (y[m]-y[i])*yn +
          (z[m]-z[i])*zn <= 0);
        if (flag) ret.push_back(triple(i, j, k));
    }
      }
  }
  return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

8 Miscellaneous

8.1 Dates

```cpp
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

8.2 Regular expressions

```java
// Code which demonstrates the use of Java's regular expression libraries.
// This is a solution for
//
//   Loglan: a logical language
//   http://acm.uva.es/p/v1/134.html
//
// In this problem, we are given a regular language, whose rules can be
// inferred directly from the code.  For each sentence in the input, we must
// determine whether the sentence matches the regular expression or not.  The
// code consists of (1) building the regular expression (which is fairly
// complex) and (2) using the regex to match sentences.

import java.util.*;
import java.util.regex.*;

public class LogLan {

    public static String BuildRegex (){
  String space = " +";

  String A = "([aeiou])";
  String C = "([a-z&&[^aeiou]])";
  String MOD = "(g" + A + ")";
  String BA = "(b" + A + ")";
  String DA = "(d" + A + ")";
  String LA = "(l" + A + ")";
  String NAM = "([a-z]*" + C + ")";
  String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A + ")";

  String predstring = "(" + PREDA + "(" + space + PREDA + ")*)";
  String predname = "(" + LA + space + predstring + "|" + NAM + ")";
  String preds = "(" + predstring + "(" + space + A + space + predstring + ")*)";
  String predclaim = "(" + predname + space + BA + space + preds + "|" + DA + space +
          preds + ")";
  String verbpred = "(" + MOD + space + predstring + ")";
  String statement = "(" + predname + space + verbpred + space + predname + "|" +
          predname + space + verbpred + ")";
  String sentence = "(" + statement + "|" + predclaim + ")";

  return "^" + sentence + "$";
    }

    public static void main (String args[]){

  String regex = BuildRegex();
  Pattern pattern = Pattern.compile (regex);

  Scanner s = new Scanner(System.in);
  while (true) {

          // In this problem, each sentence consists of multiple lines, where the last
      // line is terminated by a period.  The code below reads lines until
      // encountering a line whose final character is a '.'.  Note the use of
          //
          //    s.length() to get length of string
          //    s.charAt() to extract characters from a Java string
          //    s.trim() to remove whitespace from the beginning and end of Java string
          //
          // Other useful String manipulation methods include
          //
          //    s.compareTo(t) < 0 if s < t, lexicographically
          //    s.indexOf("apple") returns index of first occurrence of "apple" in s
          //    s.lastIndexOf("apple") returns index of last occurrence of "apple" in s
          //    s.replace(c,d) replaces occurrences of character c with d
          //    s.startsWith("apple) returns (s.indexOf("apple") == 0)
          //    s.toLowerCase() / s.toUpperCase() returns a new lower/uppercased string
          //
          //    Integer.parseInt(s) converts s to an integer (32-bit)
          //    Long.parseLong(s) converts s to a long (64-bit)
          //    Double.parseDouble(s) converts s to a double

      String sentence = "";
      while (true){
    sentence = (sentence + " " + s.nextLine()).trim();
    if (sentence.equals("#")) return;
    if (sentence.charAt(sentence.length()-1) == '.') break;
      }

          // now, we remove the period, and match the regular expression

          String removed_period = sentence.substring(0, sentence.length()-1).trim();
      if (pattern.matcher (removed_period).find()){
    System.out.println ("Good");
      } else {
    System.out.println ("Bad!");
      }
  }
    }
}
```

8.3 C++ input/output

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

8.4 Latitude/longitude

```cpp
/*
Converts from rectangular coordinates to latitude/longitude and vice
versa. Uses degrees (not radians).
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << " " << B.lat << " " << B.lon << endl;

  A = convert(B);
  cout << A.x << " " << A.y << " " << A.z << endl;
}
```

8.5 Emacs settings

```lisp
;; Jack's .emacs file

(global-set-key "\C-z"      'scroll-down)
(global-set-key "\C-x\C-p"  '(lambda() (interactive) (other-window -1)) )
(global-set-key "\C-x\C-o"  'other-window)
(global-set-key "\C-x\C-n"  'other-window)
(global-set-key "\M-."      'end-of-buffer)
(global-set-key "\M-,"      'beginning-of-buffer)
(global-set-key "\M-g"      'goto-line)
(global-set-key "\C-c\C-w"  'compare-windows)

(tool-bar-mode 0)
(scroll-bar-mode -1)

(global-font-lock-mode 1)
(show-paren-mode 1)

(setq-default c-default-style "linux")

(custom-set-variables
 '(compare-ignore-whitespace t)
)
```