UC Berkeley ICPC Team Notebook (2016-17)

Contents

1 Combinatorial optimization

1.1 Dinitz's Algorithm

```
#include <vector>
#include <queue>
#include <iostream>
#include <cstdio>

using namespace std;

typedef long long LL;

#define pb push_back

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

// Indexes of nodes are 0-indexed.
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 add_edge(int u, int v, LL cap) {
        if (u != v) {
            E.pb(Edge(u, v, cap));
            g[u].pb((int)E.size() - 1);
            E.pb(Edge(v, u, 0));
            g[v].pb((int)E.size() - 1);
        }
    }

    bool bfs(int S, int T) {
        queue<int> q; q.push(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 i = 0; i < (int)g[u].size(); i++) {
                int k = g[u][i];
                Edge &e = E[k];
                if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
                    d[e.v] = d[e.u] + 1;
                    q.push(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 < (int)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;
    }
};

// Solves SPOJ 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.add_edge(u - 1, v - 1, cap);
        dinic.add_edge(v - 1, u - 1, cap);
    }
    printf("%lld\n", dinic.maxflow(0, N - 1));
    return 0;
}
```

1.2 Min-cost Max-flow

```
// Implementation of min cost max flow algorithm using adjacency
```

```
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <deque>

using namespace std;

typedef long long F;
typedef long long C;

#define F_INF 1e+9
#define C_INF 1e+9
#define NUM 10005
#define SIZE(x) ((int)x.size())

#define pb push_back
#define mp make_pair
#define fi first
#define se second

vector<F> cap;
vector<C> cost;
vector<int> to, prv;
C dist[NUM];
int last[NUM], path[NUM];

struct MinCostFlow {
  int V;

  MinCostFlow(int n) {
    cap.clear();
    cost.clear();
    to.clear();
    prv.clear();
    V = n;
    fill(last + 1, last + 1 + V, -1);
  }

  void add_edge(int x, int y, F w, C c) {
    cap.pb(w); cost.pb(c); to.pb(y); prv.pb(last[x]); last[x] = SIZE(cap) - 1;
    cap.pb(0); cost.pb(-c); to.pb(x); prv.pb(last[y]); last[y] = SIZE(cap) - 1;
  }

  pair<F, C> SPFA(int s, int t) {
    F ansf = 0;
    C ansc = 0;
    fill(dist + 1, dist + 1 + V, C_INF);
    fill(path + 1, path + 1 + V, -1);

    deque<pair<C, int> > pq;
    dist[s] = 0;
    path[s] = -1;
    pq.push_front(mp(0, s));

    while (!pq.empty()) {
      C d = pq.front().fi;
      int p = pq.front().se;
      pq.pop_front();
      if (dist[p] == d) {
        int e = last[p];
        while (e != -1) {
          if (cap[e] > 0) {
            C nd = dist[p] + cost[e];
            if (nd < dist[to[e]]) {
              dist[to[e]] = nd;
              path[to[e]] = e;
              if (cost[e] <= 0) {
                pq.push_front(mp(nd, to[e]));
              } else {
                pq.push_back(mp(nd, to[e]));
              }
            }
          }
          e = prv[e];
        }
      }
    }
    if (path[t] != -1) {
      ansf = F_INF;
      int e = path[t];
      while (e != -1) {
        ansf = min(ansf, cap[e]);
        e = path[to[e^1]];
      }
      e = path[t];
      while (e != -1) {
        ansc += cost[e] * ansf;
        cap[e^1] += ansf;
        cap[e] -= ansf;
        e = path[to[e^1]];
      }
    }
    return mp(ansf, ansc);
  }

  pair<F, C> calc(int s, int t) {
    F ansf = 0;
    C ansc = 0;
    while (true) {
      pair<F, C> p = SPFA(s, t);
      if (path[t] == -1)
        break;
      ansf += p.fi;
      ansc += p.se;
    }
    return mp(ansf, ansc);
  }
};

int main() {
    return 0;
}
```

1.3 Min-cost Matching

```
///////////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

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

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
```

```
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
    Lmate[i] = j;
    Rmate[j] = i;
    mated++;
    break;
      }
    }
  }

  VD dist(n);
  VI dad(n);
  VI seen(n);

  // repeat until primal solution is feasible
  while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

      // find closest
      j = -1;
      for (int k = 0; k < n; k++) {
    if (seen[k]) continue;
    if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
    if (seen[k]) continue;
    const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
    if (dist[k] > new_dist) {
      dist[k] = new_dist;
      dad[k] = j;
    }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;

    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}

int main() {
    return 0;
}
```

1.4 Max bipartite Matching

```
// Solves the Maximum Matching problem on a Bipartite Graph.

#include <algorithm>
#include <iostream>

using namespace std;

const int MAXN1 = 50000;
const int MAXN2 = 50000;
const int MAXM = 150000;

int n1, n2, edges, last[MAXN1], prv[MAXM], head[MAXM];
int matching[MAXN2], dist[MAXN1], Q[MAXN1];
bool used[MAXN1], vis[MAXN1];

void init(int _n1, int _n2) {
        n1 = _n1;
        n2 = _n2;
        edges = 0;
        fill(last, last + n1, -1);
}

// Nodes are 0-indexed
void addEdge(int u, int v) {
        head[edges] = v;
        prv[edges] = last[u];
        last[u] = edges++;
}

void bfs() {
        fill(dist, dist + n1, -1);
        int sizeQ = 0;
        for (int u = 0; u < n1; u++) {
                if (!used[u]) {
                        Q[sizeQ++] = u;
                        dist[u] = 0;
                }
        }
        for (int i = 0; i < sizeQ; i++) {
                int u1 = Q[i];
                for (int e = last[u1]; e >= 0; e = prv[e]) {
                        int u2 = matching[head[e]];
                        if (u2 >= 0 && dist[u2] < 0) {
                                dist[u2] = dist[u1] + 1;
                                Q[sizeQ++] = u2;
                        }
                }
        }
}

bool dfs(int u1) {
        vis[u1] = true;
        for (int e = last[u1]; e >= 0; e = prv[e]) {
                int v = head[e];
                int u2 = matching[v];
                if (u2 < 0 || (!vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2))) {
                        matching[v] = u1;
                        used[u1] = true;
                        return true;
                }
        }
        return false;
}

int maxMatching() {
        fill(used, used + n1, false);
        fill(matching, matching + n2, -1);
        for (int res = 0;;) {
                bfs();
                fill(vis, vis + n1, false);
                int f = 0;
                for (int u = 0; u < n1; u++) {
                        if (!used[u] && dfs(u))
                                ++f;
                }
                if (!f)
                        return res;
                res += f;
        }
}

int main() {
    return 0;
}
```

1.5 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
```

2 Geometry

2.1 Convex Hull

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

#define mp make_pair
#define pb push_back

typedef double T;
const T EPS = 1e-7;

// uncomment to remove redundant points
#define REDUNDANT

struct PT {
        T x, y;
        PT() {}
        PT(T x, T y) : x(x), y(y) {}
        bool operator<(const PT &r) const { return mp(y, x) < mp(r.y, r.x); }
        bool operator==(const PT &r) const { return mp(y,x) == mp(r.y, r.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 REDUNDANT
bool bt(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

// takes in a vector of points and returns the convex hull
void CHull(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.pb(pts[i]);
                dn.pb(pts[i]);
        }
        pts = dn;
        for (int i = (int)up.size() - 2; i > 0; i--) pts.pb(up[i]);

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

// SOLVE SPOJ #26

#include <map>

double dist(PT a, PT b) {
        double dx = a.x - b.x;
        double dy = a.y - b.y;
        return sqrt(dx*dx + dy*dy);
}

int main() {
        int t;
        scanf("%d", &t);
        for (int c = 0; c < t; c++) {
                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;
                CHull(h);

                double len = 0;
                for (int i = 0; i < h.size()- 1; i++) {
                        len += dist(h[i], h[i+1]);
                }
```

```
            len += dist(h[0], h[h.size() - 1]);

            if (c > 0) puts("");

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

2.2 Graham Scan

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

typedef pair<double, double> point;

bool cw(const point &a, const point &b, const point &c) {
    return (b.first - a.first) * (c.second - a.second) - (b.second - a.second) * (c.first - a.first) <
         0;
}

vector<point> convexHull(vector<point> p) {
    int n = p.size();
    if (n <= 1)
        return p;
    int k = 0;
    sort(p.begin(), p.end());
    vector<point> q(n * 2);
    for (int i = 0; i < n; q[k++] = p[i++]) {
        for (; k >= 2 && !cw(q[k - 2], q[k - 1], p[i]); --k) {
            continue;
        }
    }
    for (int i = n - 2, t = k; i >= 0; q[k++] = p[i--]) {
        for (; k > t && !cw(q[k - 2], q[k - 1], p[i]); --k) {
            continue;
        }
    }
    q.resize(k - 1 - (q[0] == q[1]));
    return q;
}

int main() {
    vector<point> points(4);
    points[0] = point(0, 0);
    points[1] = point(3, 0);
    points[2] = point(0, 3);
    points[3] = point(1, 1);
    vector<point> hull = convexHull(points);
    cout << (3 == hull.size()) << endl;
}
```

2.3 Intersecting Line Segments

```
#include <algorithm>
#include <vector>
#include <set>
using namespace std;

typedef pair<int, int> pii;

int cross(int ax, int ay, int bx, int by, int cx, int cy) {
    return (bx - ax) * (cy - ay) - (by - ay) * (cx - ax);
}

int cross(pii a, pii b, pii c) {
    return cross(a.first, a.second, b.first, b.second, c.first, c.second);
}

class segment {
    public:
    pii a, b;
    int id;
    segment(pii a, pii b, int id) :
        a(a), b(b), id(id) {
    }
    bool operator<(const segment &o) const {
        if (a.first < o.a.first) {
            int s = cross(a, b, o.a);
            return ((s > 0) || (s == 0 && a.second < o.a.second));
        } else {
            int s = cross(o.a, o.b, a);
            return ((s < 0) || (s == 0 && a.second < o.a.second));
        }
        return a.second < o.a.second;
    }
};

bool intersect(segment s1, segment s2) {
    int x1 = s1.a.first, y1 = s1.a.second, x2 = s1.b.first, y2 = s1.b.second;
    int x3 = s2.a.first, y3 = s2.a.second, x4 = s2.b.first, y4 = s2.b.second;
    if (max(x1, x2) < min(x3, x4) || max(x3, x4) < min(x1, x2) || max(y1, y2) < min(y3, y4) || max(y3,
         y4) < min(y1, y2)) {
        return false;
    }
    int z1 = (x3 - x1) * (y2 - y1) - (y3 - y1) * (x2 - x1);
    int z2 = (x4 - x1) * (y2 - y1) - (y4 - y1) * (x2 - x1);
    if ((z1 < 0 && z2 < 0) || (z1 > 0 && z2 > 0)) {
        return false;
    }
    int z3 = (x1 - x3) * (y4 - y3) - (y1 - y3) * (x4 - x3);
    int z4 = (x2 - x3) * (y4 - y3) - (y2 - y3) * (x4 - x3);
    if ((z3 < 0 && z4 < 0) || (z3 > 0 && z4 > 0)) {
        return false;
    }
    return true;
}

class event {
    public:
    pii p;
    int id;
    int type;
    event(pii p, int id, int type) :
        p(p), id(id), type(type) {
    }
    bool operator<(const event &o) const {
        return (p.first < o.p.first) || (p.first == o.p.first && ((type > o.type || type == o.type) &&
                 p.second < o.p.second));
    }
};

pii findIntersection(vector&lt;segment&gt; a) {
    int n = a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        if (a[i].a > a[i].b)
            swap(a[i].a, a[i].b);
        e.push_back(event(a[i].a, i, 1));
        e.push_back(event(a[i].b, i, -1));
    }
    sort(e.begin(), e.end());

    set&lt;segment&gt; q;

    for (int i = 0; i < n * 2; ++i) {
        int id = e[i].id;
        if (e[i].type == 1) {
            set&lt;segment&gt;::iterator it = q.lower_bound(a[id]);
            if (it != q.end() && intersect(*it, a[id]))
                return make_pair(it->id, a[id].id);
            if (it != q.begin() && intersect(*--it, a[id]))
                return make_pair(it->id, a[id].id);
            q.insert(a[id]);
        } else {
            set&lt;segment&gt;::iterator it = q.lower_bound(a[id]), next = it, prev = it;
            if (it != q.begin() && it != --q.end()) {
                ++next, --prev;
                if (intersect(*next, *prev))
                    return make_pair(next->id, prev->id);
            }
            q.erase(it);
        }
    }
    return make_pair(-1, -1);
}

int main() {
}
```

2.4 Miscellaneous Geometry

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

```cpp
#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) {
  os << "(" << p.x << "," << p.y << ")";
  return os;
}

// 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);
}

// 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;
}

// 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;
}

// computes the reflection of a vector about a normal
PT reflect(PT d, PT n) {
    return d - n * (dot(d, n) * 2.0);
}

int main() {

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

  // 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;
}
```

3 Numerical algorithms

3.1 Number Theory (modular, Chinese remainder, Linear Diophantine)

```
#include <vector>
#include <iostream>
#include <utility>
using namespace std;
typedef vector<int> VI;
typedef pair<int, int> pii;
typedef pair<int, pii> piii;
typedef pair<int, int> PII;

// return smallest positive number equiv to a % b
int mod(int a, int b) {
  return ((a%b) + b) % b;
}

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

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

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

// returns a tuple of 3 ints containing d, x, y s.t. d = a * x + b * y
piii egcd(int a, int b) {
  int x, xx, y, yy;
  xx = y = 0; 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 piii(a, pii(x, y));
}

// returns all solutions to ax = b (mod n)
VI mod_solve(int a, int b, int n) {
  VI ret;
  int g,x;
  piii egcd_ret = egcd(a, n);
  g = egcd_ret.first;
  x = egcd_ret.second.first;
  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;
}

// modular inverse of a mod n, or -1 if gcd(a, n) != 1
int minv(int a, int n) {
  int g,x;
  piii egcd_ret = egcd(a, n);
  g = egcd_ret.first;
  x = egcd_ret.second.first;
  if (g > 1) return -1;
  return mod(x, n);
}

PII crt(int m1, int r1, int m2, int r2) {
  int g, s, t;
  piii egcd_ret = egcd(m1, m2);
  g = egcd_ret.first;
  s = egcd_ret.second.first;
  t = egcd_ret.second.second;
  if (r1 % g != r2 % g) return PII(0, -1);
  return PII(mod(s*r2*m1 + t*r1*m2, m1*m2)/g, m1*m2/g);
}

PII crt(const VI &m, const VI &r) {
  PII ret = PII(r[0], m[0]);
  for (int i = 1; i < m.size(); i++) {
    ret = crt(ret.second, ret.first, m[i], r[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

/*
Multiplying nCr quickly:

Lucas's Theorem reduces nCr % M to

(n0Cr0 % M) (n1Cr1 % M) ... (nkCrk % M)

(nknk-1...n0) is the base M representation of n
(rkrk-1...r0) is the base M representation of r

Pick's Theorem:
Area of a polygon: B/2 + I - 1
*/

int main() {
  cout << "expect 2" << endl;
  cout << gcd(14, 30) << endl;

  int g, x, y;
  piii egcd_ret = egcd(14, 30);
  g = egcd_ret.first;
  x = egcd_ret.second.first;
  y = egcd_ret.second.second;
  cout << "expect 2 -2 1" << endl;
  cout << g << " " << x << " " << y << endl;

  VI sols = mod_solve(14, 30, 100);
  cout << "expect 95 45" << endl;
  for (int i = 0; i < (int)sols.size(); i++) {
    cout << sols[i] << " ";
  }
  cout << endl;

  cout << "expect 8" << endl;
  cout << minv(8, 9) << endl;

  vector<int> v1;
  v1.push_back(3); v1.push_back(5); v1.push_back(7);
  vector<int> v2;
  v2.push_back(2); v2.push_back(3); v2.push_back(2);
  PII ret = crt(v1, v2);
  cout << "expect 23 105" << endl;
  cout << ret.first << " " << ret.second << endl;
}
```

3.2 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; }
    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;
  }
}
```

3.3 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] = {
    {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;
  }
}
```

3.4 Fast Fourier Transform

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.

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

using namespace std;

typedef double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

const double PI = acos(-1.0);

struct FFT {
  VC A;
  int n, L;

  int ReverseBits(int k) {
    int ret = 0;
    for (int i = 0; i < L; i++) {
      ret = (ret << 1) | (k & 1);
      k >>= 1;
    }
    return ret;
  }

  void BitReverseCopy(const VC &a) {
    for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
    A.resize(n);
    for (int k = 0; k < n; k++)
      A[ReverseBits(k)] = a[k];
  }

  VC DFT(const VC &a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      COMPLEX wm = exp(COMPLEX(0, 2.0 * PI / m));
      if (inverse) wm = COMPLEX(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
        COMPLEX w = 1;
        for (int j = 0; j < m/2; j++) {
          COMPLEX t = w * A[k + j + m/2];
          COMPLEX u = A[k + j];
          A[k + j] = u + t;
          A[k + j + m/2] = u - t;
          w = w * wm;
        }
      }
    }
    if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
    return A;
  }

  // c[k] = sum_{i=0}^k a[i] b[k-i]
  VD Convolution(const VD &a, const VD &b) {
```

```
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ? COMPLEX(a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ? COMPLEX(b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i]);
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(cc[i].real());
    return c;
  }

};

int n, m, a, b;
double arr[200005];
FFT fft;
bool flag[200005];
const double EPS = 1e-5;
int main() {
  arr[0] = 1.0;
  cin >> n;
  for (int i = 1; i <= n; i++) {
    cin >> a;
    arr[a] = 1.0;
  }

  VD vv(arr, arr + 200001);

  VD c = fft.Convolution(vv, vv);
  cin >> m;
  int ans = 0;
  for (int i = 1; i <= m; i++) {
    cin >> b;
    if (c[b] > EPS) {
      ++ans;
    }
  }
  cout << ans << endl;
  return 0;
}
```

3.5 Simplex Algorithm

```
#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;
}
```

4 Graph algorithms

4.1 Bellman-Ford shortest paths with negative edge weights (C++)

```
// Runs Bellman-Ford for Single-Source Shortest Paths with
// negative edge weights.
//
// Running time : O(|V| ^ 3)
//
```

```
//  INPUT:  start, w[i][j] = edge cost from i to j.
//  OUTPUT: dist[i] = min cost path from start to i.
//          prev[i] = previous node of i on best path from start node.

#include <vector>

using namespace std;

const int INF = 1000 * 1000 * 1000;

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

bool BellmanFord(const VVI &w, VI &dist, VI &prev, int start) {
    int n = static_cast<int>(w.size());
    prev = VI(n, -1);
    dist = VI(n, INF);
    dist[start] = 0;

    // Iterate (n - 1) times for algorithm,
    // and once to check for negative cycles.
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[j] > dist[i] + w[i][j]) {
                    if (k == n - 1)
                        return false;
                    dist[j] = dist[i] + w[i][j];;
                    prev[j] = i;
                }
            }
        }
    }

    return true;
}

int main() {
    return 0;
}
```

4.2 Floyd Warshall

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int INF = 1000 * 1000 * 1000;

#define mp make_pair
#define pb push_back

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

typedef pair<int, int> PII;

// Floyd-Warshall algorithm for All-Pairs Shortest paths.
// Also handles negative edge weights. Returns true if a negative
// weight cycle is found.
//
// Running time: O(|V| ^ 3)
//
//  INPUT:  w[i][j] = weight of edge from i to j
//  OUTPUT: w[i][j] = shortest path weight from i to j
//          prev[i][j] = node before j on the best path starting at i

bool FloydWarshall(VVI &w, VVI &prev) {
    int n = (int)w.size();
    prev = VVI(n, VI(n, -1));

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (w[i][j] > w[i][k] + w[k][j]) {
                    w[i][j] = w[i][k] + w[k][j];
                    prev[i][j] = k;
                }
            }
        }
    }

    // Check for negative weight cycles.
    for (int i = 0; i < n; i++)
        if (w[i][i] < 0) return false;
    return true;
}

int main() {
    return 0;
}
```

4.3 Eulerian Path

```
#include <list>
#include <vector>

using namespace std;

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

struct Edge
{
    int next_vertex;
    iter reverse_edge;

    Edge(int next_vertex)
        :next_vertex(next_vertex)
        { }
};

const int max_vertices = 100005;
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;
}
```

4.4 Minimum Spanning Trees

```
// Runs Prim's algorithm for constructing MSTs.
//
// Running time: O(|V| ^ 2)
//
//  INPUT:  w[i][j] = cost of edge from i to j
//          (Make sure that w[i][j] is nonnegative and
//           symmetric. Missing edges should be given -1
//           weight.)
//
//  OUTPUT: edges = list of pair<int, int> in MST
//                  return total weight of tree

#include <vector>
#include <queue>
#include <climits>
#include <iostream>
using namespace std;

typedef pair<int, int> pii;
typedef vector<vector<pii> > Graph;

long long prim(Graph &g, vector<int> &pred) {
    int n = g.size();
    pred.assign(n, -1);
    vector<bool> vis(n);
    vector<int> prio(n, INT_MAX);
```

```
    prio[0] = 0;
    priority_queue<pii, vector<pii> , greater<pii> > q;
    q.push(make_pair(0, 0));
    long long res = 0;

    while (!q.empty()) {
        int d = q.top().first;
        int u = q.top().second;
        q.pop();
        if (vis[u])
            continue;
        vis[u] = true;
        res += d;
        for (int i = 0; i < (int) g[u].size(); i++) {
            int v = g[u][i].first;
            if (vis[v])
                continue;
            int nprio = g[u][i].second;
            if (prio[v] > nprio) {
                prio[v] = nprio;
                pred[v] = u;
                q.push(make_pair(nprio, v));
            }
        }
    }
    return res;
}

int main() {
    Graph g(3);
    g[0].push_back(make_pair(1, 10));
    g[1].push_back(make_pair(0, 10));
    g[1].push_back(make_pair(2, 10));
    g[2].push_back(make_pair(1, 10));
    g[2].push_back(make_pair(0, 5));
    g[0].push_back(make_pair(2, 5));

    vector<int> prio;
    long long res = prim(g, prio);
    cout << res << endl;
}
```

4.5 Tarjan's Algorithm

```
/* Complexity: O(E + V)
 Tarjan's algorithm for finding strongly connected
components.
 *d[i] = Discovery time of node i. (Initialize to -1)
 *low[i] = Lowest discovery time reachable from node
 i. (Doesn't need to be initialized)
 *scc[i] = Strongly connected component of node i. (Doesn't
 need to be initialized)
 *s = Stack used by the algorithm (Initialize to an empty
 stack)
 *stacked[i] = True if i was pushed into s. (Initialize to
 false)
 *ticks = Clock used for discovery times (Initialize to 0)
 *current_scc = ID of the current_scc being discovered
 (Initialize to 0)
*/
#include <vector>
#include <stack>

using namespace std;

const int MAXN = 100005;
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
stack<int> s;
int ticks, current_scc;
void tarjan(int u){
  d[u] = low[u] = ticks++;
  s.push(u);
  stacked[u] = true;
  const vector<int> &out = g[u];
  for (int k=0, m=out.size(); k<m; ++k){
    const int &v = out[k];
    if (d[v] == -1){
      tarjan(v);
      low[u] = min(low[u], low[v]);
    }else if (stacked[v]){
      low[u] = min(low[u], low[v]);
    }
  }
  if (d[u] == low[u]){
    int v;
    do{
      v = s.top();
      s.pop();
      stacked[v] = false;
      scc[v] = current_scc;
    }while (u != v);
    current_scc++;
  }
}
```

4.6 Topological Sort (C++)

```
// This function uses performs a non-recursive topological sort.
//
// Running time: O(|V|^2).  If you use adjacency lists (vector<map<int> >),
//               the running time is reduced to O(|E|).
//
//   INPUT:   w[i][j] = 1 if i should come before j, 0 otherwise
//   OUTPUT:  a permutation of 0,...,n-1 (stored in a vector)
//            which represents an ordering of the nodes which
//            is consistent with w
//
// If no ordering is possible, false is returned.

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

using namespace std;

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

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

bool TopologicalSort (const VVI &w, VI &order){
  int n = w.size();
  VI parents (n);
  queue<int> q;
  order.clear();

  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++)
      if (w[j][i]) parents[i]++;
      if (parents[i] == 0) q.push (i);
  }

  while (q.size() > 0){
    int i = q.front();
    q.pop();
    order.push_back (i);
    for (int j = 0; j < n; j++) if (w[i][j]){
      parents[j]--;
      if (parents[j] == 0) q.push (j);
    }
  }

  return (order.size() == n);
}
```

5 Data structures

5.1 Adelson-Valskii Landis Tree

```
// Balanced Binary Search Tree implementation.

#include <cstdio>
#include <algorithm>

using namespace std;

struct node {
        int height, value, size;
        node *l, *r;
};

struct AVL {
```

```
node *root;
AVL() : root(NULL) {}
int height(node *cur) {
        if (cur == NULL) return 0;
        return cur->height;
}
int size(node *cur) {
        if (cur == NULL) return 0;
        return cur->size;
}
int size() {
        return size(root);
}
void update(node *cur) {
        if (cur == NULL) return;
        cur->height = max(height(cur->l), height(cur->r));
        cur->size = 1 + size(cur->l) + size(cur->r);
}
node *left_rotate(node *cur) {
        node *tmp = cur->l;
        cur->l = tmp->r;
        tmp->r = cur;
        update(cur);
        update(tmp);
        return tmp;
}
node *right_rotate(node *cur) {
        node *tmp = cur->r;
        cur->r = tmp->l;
        tmp->l = cur;
        update(cur);
        update(tmp);
        return tmp;
}
node *balance(node *cur) {
        if (cur == NULL) return cur;
        if (height(cur->l) - height(cur->r) == 2) {
                node *tmp = cur->l;
                if (height(tmp->l) - height(tmp->r) == -1) {
                        cur->l = right_rotate(tmp);
                }
                return left_rotate(cur);
        }
        if (height(cur->l) - height(cur->r) == -2) {
                node *tmp = cur->r;
                if (height(tmp->l) - height(tmp->r) == 1) {
                        cur->r = left_rotate(tmp);
                }
                return right_rotate(cur);
        }
        update(cur);
        return cur;
}
node *insert(node *cur, int k) {
        if (cur == NULL) {
                cur = new node;
                cur->l = cur->r = NULL;
                cur->height = 1;
                cur->value = k;
                cur->size = 1;
                return balance(cur);
        } else {
                if (k < cur->value) {
                        cur->l = insert(cur->l, k);
                } else if (k > cur->value) {
                        cur->r = insert(cur->r, k);
                }
                return balance(cur);
        }
}
void insert(int k) {
        root = insert(root, k);
}
node *erase(node *cur, int k) {
        if (cur == NULL) return cur;
        if (cur->value == k) {
                if (cur->l == NULL || cur->r == NULL) {
                        node *tmp = cur->l;
                        if (tmp == NULL) tmp = cur->r;
                        delete cur;
                        return balance(tmp);
                } else {
                        node *tmp = cur->r;
                        while (tmp->l) tmp = tmp->l;
                        cur->value = tmp->value;
                        cur->r = erase(cur->r, tmp->value);
                        return balance(cur);
                }
        } else if (cur->value > k) {
                cur->l = erase(cur->l, k);
        } else if (cur->value < k) {
                cur->r = erase(cur->r, k);
                }
                return balance(cur);
        }
        void erase(int k) {
                root = erase(root, k);
        }
        int rank(node *cur, int k) {
                if (cur == NULL) return 0;
                if (cur->value <= k)
                        return size(cur->l) + 1 + rank(cur->r, k);
                else
                        return rank(cur->l, k);
        }
        int rank(int k) {
                return rank(root, k);
        }
        int kth(node *cur, int k) {
                if (size(cur->l) >= k) return kth(cur->l, k);
                if (size(cur->l) + 1 == k) return cur->value;
                return kth(cur->r, k - size(cur->l) - 1);
        }
        int kth(int k) {
                return kth(root, k);
        }
};

int main() {
    return 0;
}
```

5.2 Union-find Disjoin Set

```
#include <iostream>
#include <vector>

using namespace std;

int find(vector<int> &C, int x) {
    return (C[x] == x) ? x : C[x] = find(C, C[x]);
}
void merge(vector<int> &C, int x, int y) {
    C[find(C, x)] = find(C, y);
}

int main() {
    return 0;
}
```

5.3 Lowest Common Ancestor

```
// Lowest Common Ancestor algorithm for two nodes in a tree.
//
// Running time: O(|V|log|V|) for precomputation, O(log|V|) per query

#include <cstdio>
#include <vector>

using namespace std;

const int max_nodes = 100000, log_max_nodes = 17;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes];
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(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 < (int)children[i].size(); j++) {
```

```
        dfs(children[i][j], l + 1);
    }
}

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

    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() {
    log_num_nodes = lb(num_nodes);

    for (int i = 0; i < num_nodes; i++) {
        int p; scanf("%d", &p);
        A[i][0] = p; // parent of node i is node p
        if (p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    for (int j = 1; j <= log_max_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;
        }
    }

    dfs(root, 0);
}
```

6 Strings

6.1 Knuth-Morris-Pratt

```
// Knuth-Morris-Pratt Algorithm for searching a substring s
// inside another string w (of length k). Returns the 0-based
// index of the first match (k if no match is found).
//
// Running Time: O(k)

#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> VI;

void precompute_kmp(string &w, VI &t) {
    t = VI((int)w.length());
    int i = 2, j = 0;
    t[0] = -1; t[1] = 0;

    while (i < (int)w.length()) {
        if (w[i-1] == w[j]) { t[i] = j + 1; i++, j++; }
        else if (j > 0) j = t[j];
        else { t[i] = 0; i++; }
    }
}

int KMP(string &s, string &w) {
    int m = 0, i = 0;
    VI t;

    precompute_kmp(w, t);
    while (m + i < (int)s.length()) {
        if (w[i] == s[m + i]) {
            i++;
            if (i == (int)w.length()) return m;
        } else {
            m += (i - t[i]);
            if (i > 0) i = t[i];
        }
    }
    return (int)s.length();
}

int main()
{
  string a = (string) "The example above illustrates the general technique for assembling "+
    "the table with a minimum of fuss. The principle is that of the overall search: "+
    "most of the work was already done in getting to the current position, so very "+
    "little needs to be done in leaving it. The only minor complication is that the "+
    "logic which is correct late in the string erroneously gives non-proper "+
    "substrings at the beginning. This necessitates some initialization code.";

  string b = "table";

  int p = KMP(a, b);
  cout << p << ": " << a.substr(p, b.length()) << " " << b << endl;
}
```

6.2 Suffix Array

```
/*
Suffix array O(n lg^2 n)
LCP table O(n)
*/
#include <cstdio>
#include <algorithm>
#include <cstring>

using namespace std;

const int MAXN = 1 << 21;
char * S;
int N, gap;
int sa[MAXN], pos[MAXN], tmp[MAXN], lcp[MAXN];

bool sufCmp(int i, int j)
{
    if (pos[i] != pos[j])
        return pos[i] < pos[j];
    i += gap;
    j += gap;
    return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}

void buildSA()
{
    N = strlen(S);
    for (int i = 0; i < N; i++)
        sa[i] = i, pos[i] = S[i];
    for (gap = 1;; gap *= 2)
    {
        sort(sa, sa + N, sufCmp);
        for (int i = 0; i < N - 1; i++)
            tmp[i + 1] = tmp[i] + sufCmp(sa[i], sa[i + 1]);
        for (int i = 0; i < N; i++)
            pos[sa[i]] = tmp[i];
        if (tmp[N - 1] == N - 1) break;
    }
}

void buildLCP()
{
    for (int i = 0, k = 0; i < N; ++i) if (pos[i] != N - 1)
    {
        for (int j = sa[pos[i] + 1]; S[i + k] == S[j + k];)
        ++k;
        lcp[pos[i]] = k;
        if (k)--k;
    }
}

/*
Suffix array O(n lg n)
*/
char str [MAXN];
int m, SA [MAXN], LCP [MAXN];
int x [MAXN], y [MAXN], w [MAXN], c [MAXN];

inline bool cmp (const int a, const int b, const int l) { return (y [a] == y [b] && y [a + l] == y [b
      + l]); }
```

```
void Sort () {
    for (int i = 0; i < m; ++i) w [i] = 0;
    for (int i = 0; i < N; ++i) ++w [x [y [i]]];
    for (int i = 0; i < m - 1; ++i) w [i + 1] += w [i];
    for (int i = N - 1; i >= 0; --i) SA [--w [x [y [i]]]] = y [i];
}

void DA () {
    for (int i = 0; i < N; ++i) x [i] = str [i], y[i] = i;
    Sort ();
    for (int i, j = 1, p = 1; p < N; j <<= 1, m = p) {
        for (p = 0, i = N - j; i < N; i++) y [p++] = i;
        for (int k = 0; k < N; ++k) if (SA [k] >= j) y [p++] = SA [k] - j;
        Sort ();
        for (swap (x, y), p = 1, x [SA [0]] = 0, i = 1; i < N; ++i) x [SA [i]] = cmp (SA [i - 1], SA [
              i], j) ? p - 1 : p++;
    }
}

void kasaiLCP () {
    for (int i = 0; i < N; i++) c [SA [i]] = i;
    for (int i = 0, j, k = 0; i < N; LCP [c [i++]] = k)
        if (c [i] > 0) for (k ? k-- : 0, j = SA [c [i] - 1]; str [i + k] == str [j + k]; k++);
        else k = 0;
}

void suffixArray () {
    m = 256;
    N = strlen (str);
    DA ();
    kasaiLCP ();
}

int main() {
    return 0;
}
```

6.3 Manacher's Algorithm

```
// Runs Manacher's algorithm to compute the longest palindrome
// in a string in linear time.

#include <string>
#include <vector>

using namespace std;

string preprocess(string &s) {
    int n = (int)s.length();
    if (n == 0) return "^$";
    string ret = "^";
    for (int i = 0; i < n; i++) {
        ret += "#" + s.substr(i, 1);
    }
    ret += "#$";
    return ret;
}

string longestPalindrome(string &s) {
    string T = preprocess(s);
    int n = (int)T.length();
    vector<int> P(n, 0);
    int c = 0, r = 0;
    for (int i = 1; i < n - 1; i++) {
        int i_mirror = 2 * c - i;

        P[i] = (r > i) ? min(r - i, P[i_mirror]) : 0;

        // Attempt to expand palindrome centered at i
        while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
            P[i]++;

        // If palindrome cenetered at i expands past r,
        // adjust center based on expanded palindrome.
        if (i + P[i] > r) {
            c = i;
            r = i + P[i];
        }
    }

    // Find the maximum element in P.
    int maxlen = 0;
    int centerIndex = 0;
    for (int i = 1; i < n - 1; i++) {
        if (P[i] > maxlen) {
            maxlen = P[i];
            centerIndex = i;
        }
    }
    return s.substr((centerIndex - 1 - maxlen) / 2, maxlen);
}

int main() {
    return 0;
}
```

6.4 Rabin Karp Algorithm

```
#include <cstdio>
#include <algorithm>
#include <map>
#include <set>
#include <utility>
#include <queue>
#include <iostream>
#include <vector>
#include <cstring>

using namespace std;

typedef long long LL;
typedef pair<int, int> pii;

const int INF = 1000 * 1000 * 1000;
const LL LLINF = 1000000000000000000LL;

#define mp make_pair
#define pb push_back

const LL MOD = 1000000009;
const LL P = 2;

int r1, c1, r2, c2;
char arr1[2005][2005];
char arr2[2005][2005];
LL hash2[2005][2005];
int xx[2005][2005];
int yy[2005][2005];
LL prec[4000005];

inline LL update(LL old, int len, LL rem, LL add, int primeEx) {
        LL val = old - prec[len - primeEx] * rem;
        val %= MOD;
        val *= prec[primeEx];
        val %= MOD;
        val += add;
        return val % MOD;
}

int main() {
        prec[0] = 1;
        for (int i = 1; i <= 4000000; i++) {
                prec[i] = prec[i-1] * P;
                prec[i] %= MOD;
        }

        scanf("%d %d %d %d",&r1,&c1,&r2,&c2);

        for (int i = 1; i <= r1; i++) {
                for (int j = 1; j <= c1; j++) {
                        register int ch = 0;
                        while (ch != 'o' && ch != 'x') ch = getchar();
                        xx[i][j] = (ch == 'o');
                }
        }

        for (int i = 1; i <= r2; i++) {
                for (int j = 1; j <= c2; j++) {
                        register int ch = 0;
                        while (ch != 'o' && ch != 'x') ch = getchar();
                        yy[i][j] = (ch == 'o');
                }
        }

        LL hash1 = 0;
        for (int i = 1; i <= r1; i++) {
                for (int j = 1; j <= c1; j++) {
                        hash1 = update(hash1, 1, 0, xx[i][j], 1);
                }
        }

        for (int i = 1; i <= r2; i++) {
                LL temp = 0;
```

```
            for (int j = 1; j <= c2; j++) {
                temp = update(temp, c1, (j > c1) ? yy[i][j - c1] : 0, yy[i][j], 1);
                if (j >= c1) {
                    hash2[i][j + 1 - c1] = temp;
                }
            }
        }

        int ans = 0;
        for (int j = 1; j <= c2 - c1 + 1; j++) {
            LL temp = 0;
            for (int i = 1; i <= r2; i++) {
                temp = update(temp, r1 * c1, (i > r1) ? hash2[i - r1][j] : 0, hash2[i][j], c1)
                    ;
                if (i >= r1 && temp == hash1) {
                    ans++;
                }
            }
        }

        printf("%d\n", ans);
        return 0;
}
```

6.5 Z Algorithm

```
// Given a string s of length n, the Z-Algorithm produces an array
// Z where Z[i] is the length of the longest substring starting from
// S[i] which is also a prefix of S.

#include <string>
#include <vector>

using namespace std;

void z_algo(const string &s, vector<int> &z) {
    int n = (int)s.length();
    int l = 0, r = 0;
    for (int i = 1; i <= n; i++) {
        if (i > r) {
            l = r = i;
            while (r < n && s[r - l] == s[r]) r++;
            z[i] = r - l; r--;
        } else {
            int k = i - l;
            if (z[k] < r - i + 1) z[i] = z[k];
            else {
                l = i;
                while (r < n && s[r - l] == s[r]) r++;
                z[i] = r - l; r--;
            }
        }
    }
}

int main() {
    return 0;
}
```