Code Template for ACM-ICPC

P_Not_Equal_NP

November 15, 2018

Contents

1 KDTree

1.1 KDTree

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

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

2 AhoCorasick

2.1 AhoCorasick

```
// Aho-Corasick
struct AhoCorasick
{
    struct Node
    {
        int cnt;
        vector<int> id;
```

```
    Node *nextNode, *nextPatternNode,
         *child[ALPHABET_SIZE];

    Node()
    {
        cnt = 0;
        id = vector<int>();
        nextNode = nextPatternNode = NULL;
        FOR(i, 0, ALPHABET_SIZE - 1)
            child[i] = NULL;
    }
} root;

void insertString(const string &s, int id)
{
    Node *p = &root;
    FOR(i, 0, int(s.size()) - 1)
    {
        int z = encode(s[i]);
        if (p->child[z] == NULL)
            p->child[z] = new Node();
        p = p->child[z];
    }
    p->id.pb(id);
}

queue<Node*> q;

void calculateNode()
{
    q.push(&root);
    while(!q.empty())
    {
        Node *p = q.front();
        q.pop();
        FOR(i, 0, ALPHABET_SIZE - 1)
        if (p->child[i] != NULL)
        {
            Node *c = p->child[i];
            Node *f = p->nextNode;
            while(true)
            {
                if (f == NULL)
                {
                    c->nextNode = &root;
                    break;
                }
                if (f->child[i] != NULL)
                {
                    c->nextNode = f->child[i];
                    break;
                }
                f = f->nextNode;
            }
            if (c->nextNode->id.empty())
                c->nextPatternNode =
                     c->nextNode->nextPatternNode;
            else
                c->nextPatternNode = c->nextNode;
            q.push(p->child[i]);
        }
    }
}

void query(const string &s)
{
    Node *p = &root;
        FOR(i, 0, int(s.size()) - 1)
        {
            int z = encode(s[i]);
            while(p != NULL && p->child[z] == NULL)
                p = p->nextNode;
            if (p == NULL)
                p = &root;
            else
            {
                p = p->child[z];
                p->cnt ++;
            }
        }
    }

    stack<Node*> st;

    void pushAnswer(int *ans)
    {
        q.push(&root);
        while(!q.empty())
        {
            Node *p = q.front();
            q.pop();
            st.push(p);
            FOR(i, 0, ALPHABET_SIZE - 1)
            if (p->child[i] != NULL)
                q.push(p->child[i]);
        }
        while(!st.empty())
        {
            Node *p = st.top();
            st.pop();
            FOR(i, 0, int(p->id.size()) - 1)
                ans[p->id[i]] += p->cnt;
            if (p->nextNode != NULL)
                p->nextNode->cnt += p->cnt;
        }
    }
};
```

3 ConvexHull

3.1 ConvexHull

```
// 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
```

4 Dates

4.1 Dates

```
// 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;
}
```

5 Euclid

5.1 Euclid

```
// 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 EulerianPath

6.1 EulerianPath

```
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 = ;
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;
}
```

7 FFT

7.1 FFT

```
// FFT
const int NBIT = 18;
const int DEGREE = 1 << NBIT;
const double PI = acos(-1);
typedef complex<double> cplx;
cplx W[DEGREE];

int reverseBit(int mask)
{
    for(int i = 0, j = NBIT - 1; i < j; i ++, j --)
    if (((mask >> i) & 1) != ((mask >> j) & 1))
    {
        mask ^= 1 << i;
        mask ^= 1 << j;
    }
    return mask;
}

void fft(vector<cplx>& v, bool invert = false)
{
    v.resize(DEGREE);
    FOR(i, 0, DEGREE - 1)
    {
        int j = reverseBit(i);
        if (i < j)
            swap(v[i], v[j]);
    }
    vector<cplx> newV = vector<cplx>(DEGREE);
    for(int step = 1; step < DEGREE; step <<= 1)
    {
        double angle = PI / step;
        if (invert)
            angle = -angle;
        W[0] = cplx(1);
        cplx wn = cplx(cos(angle), sin(angle));
        FOR(i, 1, step - 1)
            W[i] = W[i - 1] * wn;

        int startEven = 0;
        int startOdd = step;
        while(startEven < DEGREE)
        {
            FOR(i, 0, step - 1)
            {
                newV[startEven + i] = v[startEven + i] +
                    W[i] * v[startOdd + i];
                newV[startOdd + i] = v[startEven + i] -
                    W[i] * v[startOdd + i];
            }
            startEven += (step << 1);
            startOdd = startEven + step;
        }

        FOR(i, 0, DEGREE - 1)
            v[i] = newV[i];
    }
    if (invert)
        FOR(i, 0, DEGREE - 1)
            v[i] /= DEGREE;
}
```

8 GaussJordan

8.1 GaussJordan

```
// 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;
  }
}
```

9 Geom3D

9.1 Geom3D

```
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));
  }
}
```

10 Geometry

10.1 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);
}

// 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.
```

```cpp
// 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;
}

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;
}
```

11 HungarianAlgorithm

11.1 HungarianAlgorithm

```
// Hungarian Algorithm
int n, c[mn][mn], fx[mn], fy[mn];
int matchX[mn], matchY[mn], Queue[mn];
int reachX[mn], reachY[mn], inReachY[mn];
int trace[mn], numX, numY, co = 0, ans = 0;

void setup()
{
    cin >> n;
    FOR(x, 1, n)
    FOR(y, 1, n)
        c[x][y] = maxC;
    int u, v;
    while(cin >> u)
    {
        cin >> v;
        cin >> c[u][v];
    }
}

int findArgumentPath(int s)
{
    co ++;
    numX = numY = 0;
    int l = 1, r = 1;
    Queue[1] = s;
    while(l <= r)
    {
        int x = Queue[l ++];
        reachX[++ numX] = x;
```

```
        FOR(y, 1, n)
        if (inReachY[y] != co && C(x, y) == 0)
        {
            inReachY[y] = co;
            reachY[++ numY] = y;
            trace[y] = x;
            if (!matchY[y])
                return y;
            Queue[++ r] = matchY[y];
        }
    }
    return 0;
}

void changeEdge()
{
    int delta = maxC;
    FOR(i, 1, numX)
    {
        int x = reachX[i];
        FOR(y, 1, n)
        if (inReachY[y] != co)
            delta = min(delta, C(x, y));
    }
    FOR(i, 1, numX)
        fx[reachX[i]] += delta;
    FOR(i, 1, numY)
        fy[reachY[i]] -= delta;
}

void argumenting(int y)
{
    while(inReachY[y] == co)
    {
        int x = trace[y];
        int nex = matchX[x];
        matchX[x] = y;
        matchY[y] = x;
        y = nex;
    }
}

void xuly()
{
    FOR(x, 1, n)
    while(true)
    {
        int y = findArgumentPath(x);
        if (y)
        {
            argumenting(y);
            break;
        }
        changeEdge();
    }
    FOR(x, 1, n)
        ans += c[x][matchX[x]];
    cout << ans << '\n';
    FOR(x, 1, n)
        cout << x << ' ' << matchX[x] << '\n';
}
```

12 IO

12.1 IO

```
#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;
}
```

13 JavaFastIO

13.1 JavaFastIO

```
// Fast IO class in Java
static class FastReader
{
    final BufferedReader br;
    StringTokenizer st;

    FastReader()
    {
        br = new BufferedReader(new
                InputStreamReader(System.in));
    }

    String next()
    {
        while (st == null || !st.hasMoreElements())
        {
            try
            {
                st = new StringTokenizer(br.readLine());
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
        return st.nextToken();
    }

    int nextInt()
    {
        return Integer.parseInt(next());
```

```
    }
}

static class FastWriter{
    PrintWriter printWriter;

    FastWriter(){
        printWriter = new PrintWriter(new
            BufferedOutputStream(System.out));
    }

    void print(Object object){
        printWriter.print(object);
    }

    void flush(){
        printWriter.flush();
    }
}
```

14 JavaGeometry

14.1 JavaGeometry

```
// 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:
                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)).

   }
}
```

15 Knuth

15.1 Knuth

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

/*
* Complexity: O(N^2)
* f[i][j] = min(f[i][k] + f[k][j] + c[i][j], i < k < j)
* a[i][j] = min(k | i < k < j && f[i][j] = f[i][k] +
    f[k][j] + c[i][j])
* Sufficient condition: a[i][j - 1] <= a[i][j] <= a[i +
    1][j] or
* c[x][z] + c[y][t] <= c[x][t] + c[y][z] (quadrangle
    inequality) and c[y][z] <= c[x][t] (monotonicity),
    x <= y <= z <= t
*/

const int oo = (int) 1e9;
const int MAXN = 1e3 + 5;
int n;
int f[MAXN][MAXN];
int c[MAXN][MAXN];
int a[MAXN][MAXN];

void knuth() {
   for (int i = 0; i < n; i++) {
      f[i][i] = 0;
      a[i][i] = i;
   }
   for (int i = 0; i < n; i++) {
      for (int j = i + 1; j < n; j++) {
         f[i][j] = oo;
         for (int k = a[i][j - 1]; k <= a[i + 1][j];
             k++) {
            if (f[i][j] > f[i][k] + f[k][j] +
                c[i][j]) {
               f[i][j] = f[i][k] + f[k][j] + c[i][j];
               a[i][j] = k;
            }
         }
      }
   }
}

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

16 LinkcutTree2

16.1 LinkcutTree2

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

```
struct Node {
	Node();
	Node *l, *r, *p;
	int size, root, key, cnt;
	int rev, lz;
};
Node* nil = new Node();
Node::Node() {
	l = r = p = nil;
	size = root = cnt = 1;
	key = rev = lz = 0;
}
void init() {
	nil->size = nil->cnt = 0;
}
void setchild(Node* p, Node* c, int l) {
	c->p = p; l ? p->l = c : p->r = c;
}
void updatelz(Node* x, int val) {
	if (x == nil) return;
	x->lz += val;
	x->cnt += val;
}
void pushdown(Node* x) {
	if (x == nil) return;
	Node *u = x->l, *v = x->r;
	if (x->rev) {
		if (u != nil) {swap(u->l, u->r); u->rev
		    ^= 1;}
		if (v != nil) {swap(v->l, v->r); v->rev
		    ^= 1;}
		x->rev = 0;
	}
	if (x->lz) {
		if (u != nil) updatelz(u, x->lz);
		if (v != nil) updatelz(v, x->lz);
		x->lz = 0;
	}
}
void pushup(Node* x) {
	x->size = x->l->size + x->r->size + 1;
}
void rotate(Node* x) {
	Node* y = x->p;
	int l = x->p->l == x;
	if (!y->root) {
		setchild(y->p, x, y->p->l == y);
	}
	else {
		x->root = 1;
		y->root = 0;
		x->p = y->p;
	}
	setchild(y, l ? x->r : x->l, l);
	setchild(x, y, !l);
	pushup(y);
}
void splay(Node* x) {
	pushdown(x);
	while (!x->root) {
		pushdown(x->p->p); pushdown(x->p);
		    pushdown(x);
		if (!x->p->root) rotate((x->p->l == x)
		    == (x->p->p->l == x->p) ? x->p : x);
		rotate(x);
	}
	pushup(x);
}
Node* access(Node* x) {
	Node* z = nil;
	for (Node* y = x; y != nil; y = y->p) {
		splay(y);
		y->r->root = 1;
		y->r = z;
		z->root = 0;
		pushup(z = y);
	}
	splay(x);
	return z;
}
void link(Node* x, Node* y) {
	access(y); access(x);
	y->cnt += x->cnt;
	updatelz(y->l, x->cnt);
	x->p = y;
	access(x);
}
void cut(Node* x) {
	access(x);
	x->l->root = 1;
	x->l->p = nil;
	updatelz(x->l, -x->cnt);
	x->l = nil;
	pushup(x);
}
Node* findroot(Node* x) {
	access(x);
	while (x->l != nil) pushdown(x), x = x->l;
	splay(x);
	return x;
}
Node* lca(Node* x, Node* y) {
	if (findroot(x) != findroot(y)) return nil;
	access(x);
	return access(y);
}
void makeroot(Node* x) {
	access(x);
	swap(x->l, x->r);
	x->rev ^= 1;
}
int connected(Node* x, Node* y) {
    if (x == y) return 1;
    access(x); access(y);
    return x->p != nil;
}

const int MAXN = 100000 + 10;
Node node[MAXN];

int main() {
	init();
	int n = 10;
	for (int i = 1; i <= n; i++) {
		node[i].key = i;
	}
	link(node + 2, node + 1);
	link(node + 5, node + 1);
	link(node + 3, node + 2);
	link(node + 4, node + 2);
	link(node + 6, node + 5);
	cout << lca(node + 3, node + 4)->key << "\n";
	cout << findroot(node + 2)->key << "\n";
```

```
        cut(node + 3);
        cout << findroot(node + 3)->key << "\n";
        return 0;
}
```

17 MatrixInverse

17.1 MatrixInverse

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

/*
* Complexity: O(N^3)
*/
#define EPS 1e-9
typedef double T;
typedef vector<T> ROW;
typedef vector<ROW> MATRIX;

inline int sign(T x) {return x < -EPS ? -1 : x > +EPS;}
MATRIX MatrixInverse(MATRIX a) {
    int i, j, k, n = a.size();
    MATRIX res;
    res.resize(n);
    for (i = 0; i < n; i++) {
        res[i].resize(n);
        for (j = 0; j < n; j++) res[i][j] = 0;
        res[i][i] = 1;
    }
    for (i = 0; i < n; i++) {
        if (!sign(a[i][i])) {
            for (j = i + 1; j < n; j++) {
                if (sign(a[j][i])) {
                    for (k = 0; k < n; k++) {
                        a[i][k] += a[j][k];
                        res[i][k] += res[j][k];
                    }
                    break;
                }
            }
            if (j == n) {
                res.clear();
                return res;
            }
        }
        T tmp = a[i][i];
        for (k = 0; k < n; k++) {
            a[i][k] /= tmp;
            res[i][k] /= tmp;
        }
        for (j = 0; j < n; j++) {
            if (j == i) continue;
            tmp = a[j][i];
            for (k = 0; k < n; k++) {
                a[j][k] -= a[i][k] * tmp;
                res[j][k] -= res[i][k] * tmp;
            }
        }
    }
    return res;
}

int main() {
    srand(time(NULL));
    int n = 100;
    MATRIX a(n, ROW(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = rand();
        }
    }
    MATRIX ia = MatrixInverse(a);
    MATRIX b(n, ROW(n, 0));
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                b[i][j] += a[i][k] * ia[k][j];
            }
        }
    }
    for (int i = 0; i < n; i++) {
        if (sign(b[i][i] - 1)) {
            cout << "Wrong!\n";
            return 0;
        }
        for (int j = 0; j < n; j++) {
            if (i != j && sign(b[i][j])) {
                cout << "Wrong!\n";
                return 0;
            }
        }
    }
    cout << "Correct!\n";
    return 0;
}
```

18 MillerRabin

18.1 MillerRabin

```
// Randomized Primality Test (Miller-Rabin):
//   Error rate: 2^(-TRIAL)
//   Almost constant time. srand is needed

#include <stdlib.h>
#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
        LL ret=0, c=a;
        while(b)
        {
                if(b&1) ret=(ret+c)%m;
                b>>=1; c=(c+c)%m;
        }
        return ret;
}
LL ModularExponentiation(LL a, LL n, LL m)
{
        LL ret=1, c=a;
        while(n)
        {
                if(n&1) ret=ModularMultiplication(ret,
                    c, m);
                n>>=1; c=ModularMultiplication(c, c, m);
        }
        return ret;
```

```
}
bool Witness(LL a, LL n)
{
        LL u=n-1;
  int t=0;
        while(!(u&1)){u>>=1; t++;}
        LL x0=ModularExponentiation(a, u, n), x1;
        for(int i=1;i<=t;i++)
        {
                x1=ModularMultiplication(x0, x0, n);
                if(x1==1 && x0!=1 && x0!=n-1) return
                    true;
                x0=x1;
        }
        if(x0!=1) return true;
        return false;
}
LL Random(LL n)
{
  LL ret=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand();
  return ret%n;
}
bool IsPrimeFast(LL n, int TRIAL)
{
  while(TRIAL--)
  {
    LL a=Random(n-2)+1;
    if(Witness(a, n)) return false;
  }
  return true;
}
```

19 MinCostMaxFlow

19.1 MinCostMaxFlow

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

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k],
            1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best =
            k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
```

```
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};

// BEGIN CUT
// The following code solves UVA problem #10594: Data
    Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);

    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K,
          v[i][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K,
          v[i][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

20 NTT

20.1 NTT

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

const int pr[3] = {1004535809, 1007681537, 1012924417};
    //2 ^ 20 * {958, 961, 966} + 1
const int pw[3] = {3, 3, 5}; //primitive roots
struct NTT {
    static const int MAXF = 1 << 18;
    int pr;
    int rts[MAXF + 1];
    int bitrev[MAXF];
    int iv[MAXF + 1];

    int fpow(int a, int k, int p) {
        if (!k) return 1;
        int res = a, tmp = a;
        k--;
        while (k) {
            if (k & 1) {
                res = (long long) res * tmp % p;
            }
            tmp = (long long) tmp * tmp % p;
            k >>= 1;
        }
        return res;
    }
    void init(int pr, int pw) {
        this->pr = pr;
        int k = 0; while ((1 << k) < MAXF) k++;
        bitrev[0] = 0;
        for (int i = 1; i < MAXF; i++) {
            bitrev[i] = bitrev[i >> 1] >> 1 | ((i & 1)
                << k - 1);
        }
        pw = fpow(pw, (pr - 1) / MAXF, pr);
        rts[0] = 1;
        for (int i = 1; i <= MAXF; i++) {
            rts[i] = (long long) rts[i - 1] * pw % pr;
        }
        for (int i = 1; i <= MAXF; i <<= 1) {
            iv[i] = fpow(i, pr - 2, pr);
        }
    }
    void dft(int a[], int n, int sign) {
        int d = 0; while ((1 << d) * n != MAXF) d++;
        for (int i = 0; i < n; i++) {
            if (i < (bitrev[i] >> d)) {
                swap(a[i], a[bitrev[i] >> d]);
            }
        }
        for (int len = 2; len <= n; len <<= 1) {
            int delta = MAXF / len * sign;
            for (int i = 0; i < n; i += len) {
                int *w = sign > 0 ? rts : rts + MAXF;
                for (int k = 0; k + k < len; k++) {
                    int &a1 = a[i + k + (len >> 1)], &a2
                        = a[i + k];
                    int t = (long long) *w * a1 % pr;
                    a1 = a2 - t;
                    a2 = a2 + t;
                    a1 += a1 < 0 ? pr : 0;
                    a2 -= a2 >= pr ? pr : 0;
                    w += delta;
                }
            }
        }
        if (sign < 0) {
            int in = iv[n];
            for (int i = 0; i < n; i++) {
                a[i] = (long long) a[i] * in % pr;
            }
        }
    }
    void multiply(int a[], int b[], int na, int nb, int
        c[]) {
```

```
        static int fa[MAXF], fb[MAXF];
        int n = na + nb - 1; while (n != (n & -n)) n +=
            n & -n;
        for (int i = 0; i < n; i++) fa[i] = fb[i] = 0;
        for (int i = 0; i < na; i++) fa[i] = a[i];
        for (int i = 0; i < nb; i++) fb[i] = b[i];
        dft(fa, n, 1), dft(fb, n, 1);
        for (int i = 0; i < n; i++) fa[i] = (long long)
            fa[i] * fb[i] % pr;
        dft(fa, n, -1);
        for (int i = 0; i < n; i++) c[i] = fa[i];
    }
    vector<int> multiply(vector<int> a, vector<int> b) {
        static int fa[MAXF], fb[MAXF], fc[MAXF];
        int na = a.size(), nb = b.size();
        for (int i = 0; i < na; i++) fa[i] = a[i];
        for (int i = 0; i < nb; i++) fb[i] = b[i];
        multiply(fa, fb, na, nb, fc);
        int k = na + nb - 1;
        vector<int> res(k);
        for (int i = 0; i < k; i++) res[i] = fc[i];
        return res;
    }
} ntt;

const int MAXF = 1 << 18;
int n;
int a[MAXF];
int b[MAXF];
int c[MAXF];
int d[MAXF];

int main() {
    srand(time(NULL));
    ntt.init(pr[0], pw[0]);
    n = 1000;
    for (int i = 0; i < n; i++) {
        a[i] = rand() % pr[0];
        b[i] = rand() % pr[0];
    }
    ntt.multiply(a, b, n, n, c);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            d[i + j] = (d[i + j] + (long long) a[i] *
                b[j]) % pr[0];
        }
    }
    for (int i = 0; i < n + n - 1; i++) {
        assert(c[i] == d[i]);
    }
    cerr << "Correct\n";
    cerr << "\nTime elapsed: " << 1000 * clock() /
         CLOCKS_PER_SEC << "ms\n";
    return 0;
}
```

21 NTTLowMem

21.1 NTTLowMem

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

namespace NTT {
    const int maxf = 1 << 20;
    int pr, pw;

    int fpow(int a, int k, int p) {
        if (!k) return 1;
        int res = a, tmp = a;
        k--;
        while (k) {
            if (k & 1) {
                res = (long long) res * tmp % p;
            }
            tmp = (long long) tmp * tmp % p;
            k >>= 1;
        }
        return res;
    }
    void init(int _pr, int _pw) {
        pr = _pr, pw = _pw;
    }
    void dft(int a[], int pr, int pw, int n) {
        for (int m = n, h; h = m / 2, m >= 2; pw =
            (long long) pw * pw % pr, m = h) {
            for (int i = 0, w = 1; i < h; i++, w = (long
                long) w * pw % pr) {
                for (int j = i; j < n; j += m) {
                    int k = j + h, x = a[j] - a[k];
                    a[j] += a[k];
                    a[j] -= a[j] >= pr ? pr : 0;
                    a[k] = (long long) w * (x + pr) % pr;
                }
            }
        }
        for (int i = 0, j = 1; j < n - 1; j++) {
            for (int k = n / 2; k > (i ^= k); k /= 2);
            if (j < i) swap(a[i], a[j]);
        }
    }
    int fa[maxf], fb[maxf], fc[maxf];
    void multiply(int a[], int b[], int na, int nb, int
         c[]) {
        int n = na + nb - 1; while (n != (n & -n)) n +=
            n & -n;
        for (int i = na; i < n; i++) fa[i] = 0;
        for (int i = nb; i < n; i++) fb[i] = 0;
        int pwn = fpow(pw, (pr - 1) / n, pr);
        dft(fa, pr, pwn, n), dft(fb, pr, pwn, n);
        for (int i = 0; i < n; i++) fc[i] = (long long)
            fa[i] * fb[i] % pr;
        dft(fc, pr, fpow(pwn, pr - 2, pr), n);
        int in = fpow(n, pr - 2, pr);
        for (int i = 0; i < n; i++) fc[i] = (long long)
            fc[i] * in % pr;
    }
    vector<int> multiply(vector<int> a, vector<int> b) {
        int na = a.size(), nb = b.size();
        for (int i = 0; i < na; i++) fa[i] = a[i];
        for (int i = 0; i < nb; i++) fb[i] = b[i];
        multiply(fa, fb, na, nb, fc);
        int k = na + nb - 1;
        vector<int> res(k);
        for (int i = 0; i < k; i++) res[i] = fc[i];
        return res;
    }
}

const int mod = 998244353;

int main() {
```

```
    srand(time(NULL));
    vector<int> a(123), b(123);
    for (int& x : a) x = rand() % mod;
    for (int& x : b) x = rand() % mod;
    NTT::init(mod, 3);
    vector<int> c = NTT::multiply(a, b);
    vector<int> d(a.size() + b.size() - 1);
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < b.size(); j++) {
            d[i + j] += (long long) a[i] * b[j] % mod;
            d[i + j] -= d[i + j] >= mod ? mod : 0;
        }
    }
    assert(c == d);
    cerr << "Correct\n";
    cerr << "\nTime elapsed: " << 1000 * clock() /
        CLOCKS_PER_SEC << "ms\n";
    return 0;
}
```

22 Simplex

22.1 Simplex

```
// 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;
}
```

23 SuffixArray

23.1 SuffixArray

```
// Suffix Array and LCP Array
void calculateSuffixArray(string &s, int* sa, int*
    group, pair< pair<int, int> , int > * data)
{
    int n = s.size();
    FOR(i, 1, n)
        group[i] = s[i - 1];
    for(int length = 1; length <= n; length <<= 1)
    {
        FOR(i, 1, n)
            data[i] = mp(mp(group[i], (i + length > n?
                -1 : group[i + length])), i);
        sort(data + 1, data + n + 1);
        FOR(i, 1, n)
            group[data[i].S] = group[data[i - 1].S] +
                (data[i].F != data[i - 1].F);
    }
    FOR(i, 1, n)
        sa[i] = data[i].S;
}

void calculateLCPArray(string &s, int* lcp, int* sa,
    int* pos)
{
    int n = s.size();
    FOR(i, 1, n)
        pos[sa[i]] = i;
    int result = 0;
    FOR(i, 1, n)
    {
        if (pos[i] == n)
        {
            result = 0;
            continue;
        }
        int j = sa[pos[i] + 1];
        while(i + result <= n && j + result <= n && s[i
            + result - 1] == s[j + result - 1])
            result ++;
        lcp[pos[i]] = result;
        if (result)
            result --;
    }
}
```

24 SuffixAutomaton

24.1 SuffixAutomaton

```
// Suffix Automaton
class SuffixAutomaton
{
    private:
        class SAState
        {
            public:
                int length;
                SAState *link, *next[26];

                SAState(int length = 0, SAState *link =
                    NULL): length(length), link(link)
                {
                    FOR(i, 0, 25)
                        next[i] = NULL;
                }
        };

        SAState *root, *last;

    public:
        SuffixAutomaton()
        {
            last = root = new SAState(0, NULL);
        }

        void insert(char c)
        {
            c -= 'a';
            SAState* newState = new SAState(last->length
                + 1);
            while (last != NULL && last->next[c] == NULL)
            {
                last->next[c] = newState;
                last = last->link;
            }
            if (last == NULL)
                newState->link = root;
            else
            {
```

```
            SAState* stateC = last->next[c];
            if (stateC->length == last->length + 1)
                newState->link = stateC;
            else
            {
                SAState* cloneState = new
                    SAState(last->length + 1,
                    stateC->link);
                FOR(i, 0, 25)
                    cloneState->next[i] =
                        stateC->next[i];
                while (last != NULL && last->next[c]
                    == stateC)
                {
                    last->next[c] = cloneState;
                    last = last->link;
                }
                newState->link = stateC->link =
                    cloneState;
            }
        }
        last = newState;
    }

    bool checkSubstring(string& s)
    {
        SAState* state = root;
        FOR(i, 0, int(s.size()) - 1)
        {
            if (state->next[s[i] - 'a'] == NULL)
                return false;
            state = state->next[s[i] - 'a'];
        }
        return true;
    }
};
```

25 Treap

25.1 Treap

```
// Implicit Treap
template <typename T> class Treap
{
    private:
        class TreapNode
        {
            public:
                T value;
                int priority, cnt;
                TreapNode *lc, *rc;

                TreapNode() {}

                TreapNode(T value): value(value)
                {
                    priority = getRandom(1, maxC);
                    cnt = 1;
                    lc = rc = NULL;
                }
        };

        int getCount(TreapNode* node)
        {
            return (node? node->cnt : 0);
        }

        void updateCount(TreapNode* node)
        {
            if (node)
                node->cnt = getCount(node->lc) +
                    getCount(node->rc) + 1;
        }

        TreapNode* merge(TreapNode* l, TreapNode* r)
        {
            if (!l || !r)
                return (l? l : r);
            TreapNode* re = NULL;
            if (l->priority > r->priority)
            {
                l->rc = merge(l->rc, r);
                re = l;
            }
            else
            {
                r->lc = merge(l, r->lc);
                re = r;
            }
            updateCount(re);
            return re;
        }

        void split(TreapNode* node, TreapNode*& l,
            TreapNode*& r, int pos, int add = 0)
        {
            if (!node)
            {
                l = r = NULL;
                return;
            }
            int currentPos = add + getCount(node->lc);
            if (pos <= currentPos)
            {
                split(node->lc, l, node->lc, pos, add);
                r = node;
            }
            else
            {
                split(node->rc, node->rc, r, pos,
                    currentPos + 1);
                l = node;
            }
            updateCount(node);
        }

        TreapNode* get(TreapNode* node, int pos, int
            add = 0)
        {
            if (!node)
                return NULL;
            int currentPos = add + getCount(node->lc);
            if (pos == currentPos)
                return node;
            if (pos < currentPos)
                return get(node->lc, pos, add);
            return get(node->rc, pos, currentPos + 1);
        }

        void erase(TreapNode*& node, int pos, int add =
            0)
        {
```

```
        if (!node)
            return;
        int currentPos = add + getCount(node->lc);
        if (pos == currentPos)
        {
            delete node;
            node = merge(node->lc, node->rc);
        }
        else if (pos < currentPos)
            erase(node->lc, pos, add);
        else
            erase(node->rc, pos, currentPos + 1);
        updateCount(node);
    }

    void print(TreapNode* node)
    {
        if (!node)
            return;
        print(node->lc);
        cout << node->value << ' ';
        print(node->rc);
    }

    TreapNode* root;

public:
    Treap()
    {
        root = NULL;
    }

    int size()
    {
        return getCount(root);
    }

    void insert(T value, int pos)
    {
        TreapNode *l = NULL, *r = NULL;
        split(root, l, r, pos);
        TreapNode* newItem = new TreapNode(value);
        root = merge(merge(l, newItem), r);
    }

    void insert(T value)
    {
        insert(value, size());
    }

    T get(int pos)
    {
        return get(root, pos)->value;
    }

    void erase(int pos)
    {
        erase(root, pos);
    }

    void print()
    {
        print(root);
        cout << '\n';
    }
};
```

26 ZFunction

26.1 ZFunction

```
// Z Function
void calculateZFunction(string &s, int *z)
{
    int n = s.size(), l = 1, r = 1;
    FOR(i, 2, n)
    {
        int k = i - l + 1;
        if (r < i || (r >= i && z[k] >= r - i + 1))
        {
            l = i, r = max(r, i - 1);
            while(r < n && s[r] == s[r - l + 1])
                r ++;
            z[i] = r - l + 1;
        }
        else
            z[i] = z[k];
    }
}
```