CMP ACM-ICPC Team Note

Contents

1 String

1.1 AhoCorasick@

```
/*
this description is from https://github.com/stjepang/snippets
source is from Team Note of Deobureo Minkyu Party

// Aho Corasick
//
// Given a set of patterns, it builds the Aho-Corasick trie. This trie
     allows
// searching all matches in a string in linear time.
//
// To use, first call `node` once to create the root node, then call `
    insert`
// for every pattern, and finally initialize the trie by calling `
    init_aho`.
// Note: It is assumed all strings contains uppercase letters only.
//

// Globals:
// - V is the number of vertices in the trie
// - trie[x][c] is the child of node x labeled with letter 'A' + c
// - fn[x] points from node x to it's "failure" node
//
// Time complexity: O(N), where N is the sum of lengths of all
    patterns
//

// Constants to configure:
// - MAX is the maximum sum of lengths of patterns
// - ALPHA is the size of the alphabet (usually 26)
*/

const int MAXN = 100005, MAXC = 26;

int trie[MAXN][MAXC], fail[MAXN], term[MAXN], piv;
void init(vector<string> &v) {
        memset(trie, 0, sizeof(trie));
        memset(fail, 0, sizeof(fail));
        memset(term, 0, sizeof(term));
        piv = 0;
        for (auto &i : v) {
                int p = 0;
                for (auto &j : i) {
                        int id = j - 'A';
                        if (!trie[p][id]) trie[p][id] = ++piv;
                        p = trie[p][id];
                }
                term[p] = 1;
        }
        queue<int> que;
        for (int i = 0; i < MAXC; i++) {
                if (trie[0][i]) que.push(trie[0][i]);
        }
        while (!que.empty()) {
                int x = que.front();
                que.pop();
                for (int i = 0; i < MAXC; i++) {
                        if (trie[x][i]) {
                                int p = fail[x];
                                while (p && !trie[p][i]) p = fail[p];
                                p = trie[p][i];
                                fail[trie[x][i]] = p;
```

```
                                term[trie[x][i]] += term[p];
                                que.push(trie[x][i]);
                        }
                }
        }
}

int query(string &s) {
        int p = 0, ret = 0;
        for (auto &i : s) {
                int id = i - 'A';
                while (p && !trie[p][id]) p = fail[p];
                p = trie[p][id];
                ret += term[p];
        }
        return ret;
}
```

1.2 KMP

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * O(n+m)
 */
struct KMP{
    string p;
    vector<int> pi;
    KMP(){}
    KMP(string _p){
        p=_p;
        pi.resize(p.size()+1);
        pi[0]=-1;
        for (int i = 0, j = -1; i<p.size(); pi[++i] = ++j) while (~j
            && p[i] ^ p[j]) j = pi[j];
    }
    vector<int> findon(string t){
        vector<int> res;
        for (int i = 0, j = 0; i<t.size(); i++) {
            while (~j && t[i] ^ p[j]) j = pi[j];
            if (++j==p.size()) res.push_back(i + 1 - j), j=pi[j];
        }
        return res;
    }
};
```

1.3 Manacher@

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

// credit: https://github.com/stjepang/snippets/blob/master/manacher.
    cpp
//
// Finds all palindromes in a string
//
// Given a string s of length N, finds all palindromes as its
    substrings.
//
// After calling manacher(s, N, rad), rad[x] will be the radius of the
    largest
// palindrome centered at index x / 2.
// Example:
//   s   = b a n a n a a
//   rad = 0000102010010
//
// Note: Array rad must be of length at least twice the length of the
    string.
// Also, "invalid" characters are denoted by -1, therefore the string
    must not
// contain such characters.
//
// Time complexity: O(N)
//
// Constants to configure:
// - MAX is the maximum length of the string
#define FOR(i, a, b) for (int i = (a); i < (b); ++i)
#define REP(i, n) FOR(i, 0, n)
const int MAX = 100;
void manacher(char *s, int N, int *rad) {
    static char t[2 * MAX];
    int m = 2 * N - 1;

    REP(i, m) t[i] = -1;
    REP(i, N) t[2 * i] = s[i];

    int x = 0;
    FOR(i, 1, m) {
        int &r = rad[i] = 0;
        if (i <= x + rad[x]) r = min(rad[x + x - i], x + rad[x] - i);
        while (i - r - 1 >= 0 && i + r + 1 < m && t[i - r - 1] == t[i
            + r + 1]) ++r;
        if (i + r >= x + rad[x]) x = i;
    }

    REP(i, m) if (i - rad[i] == 0 || i + rad[i] == m - 1) ++rad[i];
    REP(i, m) rad[i] /= 2;
}

char s[] = "banana";
int ret[100];
int main() {
    manacher(s,6, ret);
    for (int i = 0; i < 20; i++) printf("%d ", ret[i]);
    return 0;
}
```

1.4 MinRotation@

```
// Lexicographically minimum rotation of a sequence
//
// Given a sequence s of length N, min_rotation(s, N) returns the
    start index
// of the lexicographically minimum rotation.
//
// Note: array s must be of length of at least 2 * N.
```

```
//
// Time complexity: O(N)

int min_rotation(int *s, int N) {
  REP(i, N) s[N+i] = s[i];

  int a = 0;
  REP(b, N) REP(i, N) {
    if (a+i == b || s[a+i] < s[b+i]) { b += max(0, i-1); break; }
    if (s[a+i] > s[b+i]) { a = b; break; }
  }
  return a;
}
```

1.5 PalindromicTree@

```
// Palindromic tree
//
// Given a string, consider all its palindromic substrings.
// Denote every palindrome by its radius inside out. For example,
    denote the
// palindrome 'abcba' by 'cba'.
// This algorithm constructs a trie of all such radiuses.
//
// The algorithm is very similar to Aho-Corasick.
// More information:
// - http://adilet.org/blog/25-09-14/
// - http://codeforces.com/blog/entry/13959
//
// To run, set N and s, then call paltree().
// Note: It is assumed the string contains uppercase letters only.
//
// Globals:
// - N is the length of the string
// - s is the string
// - V is the number of vertices in the trie
// - trie[x][c] is the child of node x labeled with letter 'A' + c
// - fn[x] points from node x to it's "failure" node
// - len[x] is the depth of node x
//
// The root of even palindromes is node 0.
// The root of odd palindromes is node 1.
//
// Time complexity: O(N)
//
// Constants to configure:
// - MAX is the maximum length of the string
// - ALPHA is the size of the alphabet (usually 26)

int N;
char s[MAX];

int V;
int trie[MAX][ALPHA];
int fn[MAX], len[MAX];

int node() {
  REP(i, ALPHA) trie[V][i] = 0;
  fn[V] = len[V] = 0;
  return V++;
}

int suffix(int t, int i) {
  while (i-len[t]-1 < 0 || s[i-len[t]-1] != s[i]) t = fn[t];
  return t;
}

void paltree() {
  V = 0; node(); node();
  len[0] =  0; fn[0] = 1;
  len[1] = -1; fn[1] = 0;

  int t = 0;
  REP(i, N) {
    int c = s[i] - 'A';
    t = suffix(t, i);

    int &x = trie[t][c];
    if (!x) {
      x = node();
      len[x] = len[t] + 2;
      fn[x] = t == 1 ? 0 : trie[suffix(fn[t], i)][c];
    }
    t = x;
  }
}
```

1.6 SuffixArray

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


/*
credit: http://blog.myungwoo.kr/57


Suffix Array: list of suffix in lexicographical order

LCP (Longest Common Prefix)
lcp[i]: the maximum length of common prefix of ith and i-1th suffix

Ex) banana
 suffix   i   lcp
------------------
 a        6    x
 ana      4    1
 anana    2    3
 banana   1    0
 na       5    0
 nana     3    2

*/

const int MAXN = 500005;
int N, SA[MAXN], lcp[MAXN];
char S[MAXN];

// O(nlgn)
void SuffixArray()
```

```
{
    int i, j, k;
    int m = 26; // the number of alphabet
    vector <int> cnt(max(N, m) + 1, 0), x(N + 1, 0), y(N + 1, 0);
    for (i = 1; i <= N; i++) cnt[x[i] = S[i] - 'a' + 1]++;
    for (i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
    for (i = N; i; i--) SA[cnt[x[i]]--] = i;
    for (int len = 1, p = 1; p < N; len <<= 1, m = p) {
        for (p = 0, i = N - len; ++i <= N;) y[++p] = i;
        for (i = 1; i <= N; i++) if (SA[i] > len) y[++p] = SA[i] - len
            ;
        for (i = 0; i <= m; i++) cnt[i] = 0;
        for (i = 1; i <= N; i++) cnt[x[y[i]]]++;
        for (i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
        for (i = N; i; i--) SA[cnt[x[y[i]]]--] = y[i];
        swap(x, y); p = 1; x[SA[1]] = 1;
        for (i = 1; i < N; i++)
            x[SA[i + 1]] = SA[i] + len <= N && SA[i + 1] + len <= N &&
                 y[SA[i]] == y[SA[i + 1]] && y[SA[i] + len] == y[SA[i
                + 1] + len] ? p : ++p;
    }
}


// O(n)
// calculate lcp[i] for 1<i<=N
void LCP()
{
    int i, j, k = 0;
    vector <int> rank(N + 1, 0);
    for (i = 1; i <= N; i++) rank[SA[i]] = i;
    for (i = 1; i <= N; lcp[rank[i++]] = k)
        for (k ? k-- : 0, j = SA[rank[i] - 1]; S[i + k] == S[j + k]; k
            ++);
}
```

1.7 SuffixAutomaton@

```
//credit: https://github.com/PetarV-/Algorithms/blob/master/Data%20
    Structures/Suffix%20Automaton.cpp

/*
 Petar 'PetarV' Velickovic
 Data Structure: Suffix Automaton
*/

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

#define MAX_N 250001
#define MAX_S 1000010
#define MAX_K 26

typedef long long lld;
typedef unsigned long long llu;
using namespace std;

/*
 The suffix automaton is a minimal deterministic finite automaton (DFA
     ) accepting all
 suffices of a given string. It is useful for a variety of actions,
     one of which is efficiently
 computing the longest common substring (LCS) between two strings.
 Complexity:    O(n) for constructing the automaton (character by
     character)
                O(m) for computing the LCS
*/

char inp[MAX_N];

struct Node
{
    int len, prev;
    int adj[MAX_K];
};

Node DFA[MAX_S];
int tot = 0, sink;

inline void init_automaton()
{
    DFA[0].len = 0;
    DFA[0].prev = -1;
    for (int i = 0; i < 26; i++) DFA[0].adj[i] = -1;
    sink = 0;
    tot = 1;
}

inline void extend_automaton(char ch)
{
    int cc = ch - 'a';

    int tail = tot++;
    DFA[tail].len = DFA[sink].len + 1;
    for (int i = 0; i < 26; i++) DFA[tail].adj[i] = -1;

    int curr = sink;
    while (curr != -1 && DFA[curr].adj[cc] == -1)
    {
        DFA[curr].adj[cc] = tail;
        curr = DFA[curr].prev;
    }

    if (curr == -1)
    {
        DFA[tail].prev = 0;
    }
    else
    {
        int nxt = DFA[curr].adj[cc];
        if (DFA[nxt].len == DFA[curr].len + 1)
        {
            DFA[tail].prev = nxt;
        }
        else
        {
            int cl = tot++;
            DFA[cl].len = DFA[curr].len + 1;
            DFA[cl].prev = DFA[nxt].prev;
            for (int i = 0; i < 26; i++) DFA[cl].adj[i] = DFA[nxt].adj
                [i];
```

```
            while (curr != -1 && DFA[curr].adj[cc] == nxt)
            {
                DFA[curr].adj[cc] = cl;
                curr = DFA[curr].prev;
            }

            DFA[tail].prev = DFA[nxt].prev = cl;
        }
    }
    sink = tail;
}

inline string lcs(string needle)
{
    int curr_len = 0;
    int best_len = 0, best_pos = -1;
    int st = 0;

    for (int i = 0; i < needle.length(); i++)
    {
        char cc = needle[i] - 'a';
        if (DFA[st].adj[cc] == -1)
        {
            while (st != -1 && DFA[st].adj[cc] == -1)
            {
                st = DFA[st].prev;
            }
            if (st == -1)
            {
                st = 0, curr_len = 0;
                continue;
            }
            curr_len = DFA[st].len;
        }
        curr_len++;
        if (best_len < curr_len)
        {
            best_len = curr_len;
            best_pos = i;
        }
        st = DFA[st].adj[cc];
    }
    return needle.substr(best_pos - best_len + 1, best_len);
}

int main()
{
    string s1 = "alsdfkjfjkdsal";
    string s2 = "fdjskalajfkdsla";

    init_automaton();
    for (int i = 0; i < s1.length(); i++) extend_automaton(s1[i]);

    cout << lcs(s2) << endl;

    return 0;
}
```

2 Geometry

2.1 ConvexHull

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;
/*
 * credit: Youngin Cho
 * every points must be distinct
 * O(nlgn)
 * solve() : return counter-clockwise convex points vector
 * farthest(..) : return the vector, ith element refers the index of
     the point which farthest to p[i]
 *                the input should counter-clockwise convex points
     vector with size >= 2
 */
struct ConvexHull {
        vector<pii> p; // points pair
#define x first
#define y second
        long long ccw(pii i, pii j, pii k) {
                return 1LL * (j.x - i.x)*(k.y - i.y) - 1LL * (k.x - i.
                    x)*(j.y - i.y);
        }
        vector<pii> solve() {
                swap(p[0], *min_element(p.begin(), p.end()));
                sort(p.begin() + 1, p.end(), [&](pii u, pii v) {
                        long long t = ccw(p[0], u, v);
                        return t > 0 || !t&&u < v;
                });
                vector<pii> stk;
                for (auto it : p) {
                        while (stk.size() > 1 && ccw(stk[stk.size() -
                            2], stk[stk.size() - 1], it) <= 0) stk.
                            pop_back();
                        stk.push_back(it);
                }
                return stk;
        }
        vector<int> farthest(vector<pii> ch) {
                vector<int> ret;
                int sz = ch.size();
                for (int i = 0, j = 1; i < sz; i++) {
                        while (ccw(ch[i], ch[(i + 1) % sz],
                                { ch[i].x + ch[(j + 1) % sz].x - ch[j
                                    ].x, ch[i].y + ch[(j + 1) % sz].y
                                    - ch[j].y }) > 0)
                                j = (j + 1) % sz;
                        ret.push_back(j);
                }
                return ret;
        }
#undef x,y
};
```

2.2 GeomLibrary@

```
// C++ routines for computational geometry.
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 << ")";
}

// 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;
}

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;
}
```

2.3 PlaneSweeping

```
#include<cstdio>
#include<vector>
#include<algorithm>
using namespace std;
/*
 * credit: Youngin Cho
 * add_plane(...): add colored plane
 * solve(): the size of colored loacation
 */
struct PS {
    vector<int> idx, lt, ct;
    struct Line {
        int x, y1, y2, t;
    };
    vector<Line> line;
    PS(){}
    void add_plane(int minx, int maxx, int miny, int maxy) {
        line.push_back({ minx,miny,maxy,1 });
        line.push_back({ maxx,miny,maxy,-1 });
        idx.push_back(miny);
        idx.push_back(maxy);
    }
    void update(int h, int l, int r, int gl, int gr, int x) {
        if (r < gl || gr < l) return;
        if (gl <= l && r <= gr) ct[h] += x;
        else {
            update(h * 2 + 1, l, (l + r) / 2, gl, gr, x);
            update(h * 2 + 2, (l + r) / 2 + 1, r, gl, gr, x);
        }
        if (ct[h]) lt[h] = idx[r + 1] - idx[l];
        else lt[h] = l^r ? lt[h * 2 + 1] + lt[h * 2 + 2] : 0;
    }
    long long solve() {
        sort(line.begin(), line.end(), [](Line l, Line r) {return l.x
            < r.x; });
        sort(idx.begin(), idx.end());
        idx.resize(unique(idx.begin(), idx.end()) - idx.begin());
        lt.resize(idx.size() * 4);
        ct.resize(idx.size() * 4);
        long long res = 0;
        for (int i = 0; i < line.size(); i++) {
            if (i) res += (long long)lt[0] * (line[i].x - line[i - 1].
                x);
            update(0, 0, idx.size() - 1, lower_bound(idx.begin(), idx.
                end(), line[i].y1) - idx.begin(),
                lower_bound(idx.begin(), idx.end(), line[i].y2) - idx.
                    begin() - 1, line[i].t);
        }
        return res;
    }
};
```

3 Network Flow

3.1 Dinic+LRMaxFlow

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * Dinic(v,s,t): compute maxflow using 0..v-1 as node indices, source:
      s, sink: t
 * O(min(V^2E,Ef))
 */
struct Dinic{
    const int inf = 1e9;
    int v,s,t;
    struct edge{
        int x,cap,rev;
    };
    vector<vector<edge> > adj;
    vector<int> lv,work;
    Dinic(){}
    Dinic(int _v,int _s,int _t):v(_v),s(_s),t(_t),adj(v),lv(v),work(v)
        {}
    void add_edge(int x,int y,int cap){
        int sz=adj[x].size(),rsz=adj[y].size();
        adj[x].push_back({y,cap,rsz});
        adj[y].push_back({x,0,sz});
    }
    bool bfs(){
        for(int i=0; i<v; i++) lv[i]=work[i]=0;
        queue<int> q;
        q.push(s);
        lv[s]=1;
        while(!q.empty()){
            int h=q.front();
            q.pop();
            for(auto it:adj[h]) if(it.cap && !lv[it.x]){
                lv[it.x]=lv[h]+1;
                q.push(it.x);
            }
        }
        return lv[t];
    }
    int dfs(int h,int flow){
        if(h==t) return flow;
        for(int &i=work[h]; i<adj[h].size(); i++) if(lv[h] < lv[adj[h
            ][i].x] && adj[h][i].cap){
            int ret = dfs(adj[h][i].x,min(flow,adj[h][i].cap));
            if(ret){
                adj[h][i].cap-=ret;
                adj[adj[h][i].x][adj[h][i].rev].cap+=ret;
                return ret;
            }
        }
        return 0;
    }
    int solve(){
        int res=0;
        while(bfs()) for(int flow; flow=dfs(s,inf);) res+=flow;
        return res;
    }
};
/*
 * LRFlow(v,s,t) use 0..v-1 as node index, source: s, sink: t
 * note that new sink, source is not included in 0..v-1
 *
 * int solve(vector<vector<int> > &res)
 * return value: maxflow or -1 if impossible
 * res: (x,y,f,c) x->y? flow(f), capacity(c)
 *
 */
struct LRFlow{
    const int inf = 1e9;
    int v,s,t,ns,nt,goal=0;
    struct infor{
        int x,y,low,forw,back;
    };
    vector<infor> ad;
    Dinic *dn;
    LRFlow(){}
    LRFlow(int _v,int _s, int _t){
        v=_v;
        s=_s;
        t=_t;
        ns=v;
        nt=v+1;
        dn = new Dinic(v+2,s,t);
        dn->add_edge(t,s,inf);
    }
    void add_edge(int x,int y,int l,int r){
        dn->add_edge(x,nt,l);
        dn->add_edge(ns,y,l);
        ad.push_back({x,y,l,int(dn->adj[x].size()),int(dn->adj[y].size
            ())});
        dn->add_edge(x,y,r-l);
        goal+=l;
    }
    int solve(vector<vector<int> > &res){
        dn->s=ns; dn->t=nt;
        if(dn->solve()^goal) return -1;
        dn->s=s; dn->t=t;
        int ret = dn->solve();
        for(auto it:ad){
            int f=it.low+dn->adj[it.y][it.back].cap, c=f+dn->adj[it.x
                ][it.forw].cap;
            res.push_back({it.x,it.y,f,c});
        }
        return ret;
    }
};
```

3.2 HopecroftKarp

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit:Youngin Cho
 * the index starts at 1
```

```
 * O(V^0.5E)
 */
struct HK{
    int vl,vr;
    vector<int> lv,mc,rev;
    vector<vector<int> > adj;
    HK(){}
    HK(int _vl,int _vr):vl(_vl),vr(_vr),lv(vl+1),mc(vl+1),rev(vr+1),
        adj(vl+1){}
    void add_edge(int x,int y){
        adj[x].push_back(y);
    }
    int bfs(){
        queue<int> q;
        for(int i=1; i<=vl; i++){
            if(!mc[i]) lv[i]=1,q.push(i);
            else lv[i]=0;
        }
        int ret=0;
        while(!q.empty()){
            int h=q.front();
            q.pop();
            for(int it:adj[h]){
                if(!rev[it]) ret=1;
                else if(!lv[rev[it]]) lv[rev[it]]=lv[h]+1,q.push(rev[
                    it]);
            }
        }
        return ret;
    }
    int dfs(int h){
        for(int it:adj[h]) if(!rev[it] || lv[rev[it]]>lv[h] && dfs(rev
            [it])){
            mc[h]=it;
            rev[it]=h;
            return 1;
        }
        lv[h]=0;
        return 0;
    }
    vector<pair<int,int> > solve(){
        vector<pair<int,int> > res;
        while(bfs()) for(int i=1; i<=vl; i++) if(!mc[i]) dfs(i);
        for(int i=1; i<=vl; i++) if(mc[i]) res.push_back({i,mc[i]});
        return res;
    }
};
```

3.3 MCMF(SPFA)

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * MCMF(v,s,t): use 0..v-1 as node index, s: source, t: sink
 */
struct MCMF{
    const int inf = 1e9;
    int v,s,t;
    vector<int> ck,fr,mini;
    struct edge{
        int x,cap,cost,rev;
    };
    vector<vector<edge> > adj;
    MCMF(){}
    MCMF(int _v,int _s,int _t):v(_v),s(_s),t(_t),ck(v),fr(v),mini(v),
        adj(v){}
    void add_edge(int x,int y,int cap,int cost){
        int sz=adj[x].size(),rsz=adj[y].size();
        adj[x].push_back({y,cap,cost,rsz});
        adj[y].push_back({x,0,-cost,sz});
    }
    pair<int,int> solve(){
        int totfl=0,totcst=0;
        for(;;){
            for(int i=0; i<v; i++) mini[i]=inf;
            queue<int> q;
            q.push(s);
            mini[s]=0;
            while(!q.empty()){
                int h=q.front();
                q.pop();
                ck[h]=0;
                for(auto &it:adj[h]) if(it.cap && mini[it.x]>mini[h]+
                    it.cost){
                    mini[it.x]=mini[h]+it.cost;
                    fr[it.x]=it.rev;
                    if(!ck[it.x]) ck[it.x]=1,q.push(it.x);
                }
            }
            if(mini[t]==inf) break;
            int flow=inf;
            for(int i=t; i!=s; i=adj[i][fr[i]].x) flow=min(flow,adj[
                adj[i][fr[i]].x][adj[i][fr[i]].rev].cap);
            for(int i=t; i!=s; i=adj[i][fr[i]].x){
                adj[adj[i][fr[i]].x][adj[i][fr[i]].rev].cap-=flow;
                adj[i][fr[i]].cap+=flow;
            }
            totfl+=flow;
            totcst+=mini[t]*flow;
        }
        return {totfl,totcst};
    }
};
```

4 Query Problem

4.1 Dynamic+CHT@

```
// code credit : https://github.com/niklasb/contest-algos/
// blob/master/convex_hull/dynamic.cpp
using line_t = double;
const line_t is_query = -1e18;
struct Line {
    line_t m, b;
    mutable function<const Line*()> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
```

```
        const Line* s = succ();
        if (!s) return 0;
        line_t x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line> { // will maintain upper
    hull for maximum
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x
            ->m);
    }
    void insert_line(line_t m, line_t b) {
        auto y = insert({ m, b });
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    line_t query(line_t x) {
        auto l = *lower_bound((Line) { x, is_query });
        return l.m * x + l.b;
    }
}H;
```

4.2 DynamicConnectivityProblem+Offline

```
#include<bits/stdc++.h>
using namespace std;
/*
 * Union-find structure with rank supporting restore
 * this structure only use rank compression(by size)
 * UF(n): use 0..n-1
 * find(x): return the root of tree containing x
 * merge(x,y): union the tree containing x and the tree containing y
 *             it return the index of operation (if x and y are in the
      same group, then return -1)
 * restore(x): restore the structure immediately before executing
     operation x
 */
struct UF {
    vector<int> p, sz;
    vector<pair<int, int> > rc;
    UF() {}
    UF(int n) {
        p.resize(n);
        sz.resize(n);
        for (int i = 0; i < n; i++) p[i] = i, sz[i] = 1;
    }
    int find(int x) { return x == p[x] ? x : find(p[x]); }
    int merge(int x, int y) {
        int rx = find(x), ry = find(y);
        if (rx == ry) return -1;
        if (sz[rx] < sz[ry]) swap(rx, ry);
        p[ry] = rx;
        sz[rx] += sz[ry];
        rc.emplace_back(rx, ry);
        return (int)rc.size() - 1;
    }
    void restore(int idx) {
        while (rc.size() > idx) {
            int pa = rc.back().first, ch = rc.back().second;
            rc.pop_back();
            sz[pa] -= sz[ch];
            p[ch] = ch;
        }
    }
};
/*
 * DCP(n): use 0..n-1 as node index
 * connect(x,y): add connecting operation between x and y to query
     list
 *                 if x and y are already connected, then ignore
 * disconnect(x,y): add disconnecting operation between x and y to
     query list
 *                 if x and y are already disconnected, then ignore
 * check(x,y): add checking operation between x and y to query list
 * solve(res): save the result of each check operation (0:
     disconnected, 1:connected)
 *             this function must be used once
 */
struct DCP {
    UF *uf;
    vector<vector<pair<int, int> > > tree;
    map<pair<int, int>, int> mp;
    struct ins {
        int s, e;
        pair<int, int> eg;
    };
    vector<ins> inserts;
    vector<pair<int, int> > q;
    DCP() {}
    DCP(int n) {
        uf = new UF(n);
    }
    void update(int h, int l, int r, int gl, int gr, pair<int, int> eg
        ) {
        if (r < gl || gr < l) return;
        if (gl <= l && r <= gr) tree[h].push_back(eg);
        else {
            int mid = l + r >> 1;
            update(h * 2 + 1, l, mid, gl, gr, eg);
            update(h * 2 + 2, mid + 1, r, gl, gr, eg);
        }
    }
    void query(int h, int l, int r, vector<int> &res) {
        int t = uf->rc.size();
        for (auto &it : tree[h]) uf->merge(it.first, it.second);
        if (l == r) res[l] = uf->find(q[l].first) == uf->find(q[l].
            second);
        else {
            int mid = l + r >> 1;
            query(h * 2 + 1, l, mid, res);
            query(h * 2 + 2, mid + 1, r, res);
```

```
        }
        uf->restore(t);
    }
    void connect(int x, int y) {
        if (x > y) swap(x, y);
        if (mp.find({ x,y }) == mp.end()) mp[{ x, y }] = q.size();
    }
    void disconnect(int x, int y) {
        if (x > y) swap(x, y);
        auto it = mp.find({ x,y });
        if (it == mp.end()) return;
        inserts.push_back({ it->second,(int)q.size() - 1,it->first });
        mp.erase(it);
    }
    void check(int x, int y) {
        q.emplace_back(x, y);
    }
    void solve(vector<int> &res) {
        while (!mp.empty()) {
            auto it = mp.begin();
            inserts.push_back({ it->second,(int)q.size() - 1,it->first
                 });
            mp.erase(it);
        }
        tree.resize(q.size() * 4);
        for (auto &it : inserts) update(0, 0, (int)q.size() - 1, it.s,
             it.e, it.eg);
        res.resize(q.size());
        if(!q.empty()) query(0, 0, q.size() - 1, res);
    }
};
```

4.3 HLD

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * HLD(n,rt): use 1..n as node index, the index of root is rt
 * After excuting generate(), you can use the following *O(N)
 * - idx[i]: the new index(' notation in the below) assigned to node i
     (1~n),
 *           in the same block, these index's consist of consecutive
     numbers (child<parent)
 * - par[i]: the parent index of node i
 * - cnt[i]: the number of node in the subtree whose root is node i
 * - last[i]: the index of last node(lowest depth) in the block
     containing node i
 *            you may use this to identify the block containing node i
 * lca(x,y): return index' pairs correspond to path between node x, y
     *O(lgN)
 *           for each pair (u,v), u<=v and the path u-..-v+1 is in the
      same block
 */
struct HLD{
    vector<int> last,idx,cnt,par;
    vector<vector<int> > adj;
    int n,rt,c=0;
    HLD(){}
    HLD(int _n,int _rt):n(_n),rt(_rt),last(n+1),idx(n+1),cnt(n+1),par(
        n+1),adj(n+1){}
    void add_edge(int x,int y){
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    void count(int h, int p) {
        for (int it : adj[h]) if (it^p) count(it, h), cnt[h] += cnt[it
            ];
        cnt[h]++;
    }
    void build(int h, int p) {
        int t = 0;
        for (int it : adj[h]) if (it^p && cnt[t] < cnt[it]) t = it;
        for (int it : adj[h]) if (it^p && it^t) build(it, h);
        if (!last[h]) last[h] = h;
        if (t) last[t] = last[h], build(t, h);
        par[h] = p;
        idx[h] = ++c;
    }
    void generate(){
        count(rt, -1);
        build(rt, -1);
    }
    vector<pair<int,int> > lca(int x, int y) {
        vector<pair<int,int> > ret;
        while (last[x] ^ last[y]) {
            if (cnt[last[x]] > cnt[last[y]]) swap(x, y);
            ret.push_back({idx[x], idx[last[x]]});
            x = par[last[x]];
        }
        if (cnt[x] ^ cnt[y]){
            if (cnt[x] > cnt[y]) swap(x, y);
            ret.push_back({idx[x],idx[y]-1});
        }
        return ret;
    }
};
```

4.4 KDTree+closest

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

/*
 * credit: Youngin Cho
 * KDT(vp): constructs from n points in O(nlg^2n) time
 *  - vp must contain at least one point
 * closest(p): return the shortest distance to p
 *  - it takes O(lg n) if points are well distributed
 *  - it may take linear time in pathological case
 */

const int MX = 1e9, MN = -1e9;
typedef pair<int, int> point;
long long mydist(point a, point b) {
    return 1LL * (a.first - b.first)*(a.first - b.first) + 1LL * (a.
        second - b.second)*(a.second - b.second);
}
struct KDT {
```

```
    struct box {
        int minx, miny, maxx, maxy;
        long long dist(point p) {
            point t;
            if (p.first < minx) t.first = minx;
            else if (p.first > maxx) t.first = maxx;
            else t.first = p.first;
            if (p.second < miny) t.second = miny;
            else if (p.second > maxy) t.second = maxy;
            else t.second = p.second;
            return mydist(p, t);
        }
    };
    vector<box> node;
    vector<point> vp;
    KDT(vector<point> _vp) {
        vp = _vp;
        node.resize(vp.size() * 4);
        init(0, 0, vp.size() - 1);
    }
    void init(int h, int l, int r) {
        int minx = MX, miny = MX, maxx = MN, maxy = MN;
        for (int i = l; i <= r; i++) {
            minx = min(minx, vp[i].first);
            miny = min(miny, vp[i].second);
            maxx = max(maxx, vp[i].first);
            maxy = max(maxy, vp[i].second);
        }
        node[h] = { minx,miny,maxx,maxy };
        if (l == r) return;
        sort(vp.begin() + l, vp.begin() + r + 1, [&](point a, point b)
             {
            if (maxx - minx > maxy - miny) return a.first < b.first;
            return a.second < b.second;
        });
        int mid = l + r >> 1;
        init(h * 2 + 1, l, mid);
        init(h * 2 + 2, mid + 1, r);
    }
    long long search(int h, int l, int r, point p) {
        if (l == r) return mydist(vp[l], p);
        int mid = l + r >> 1;
        long long ret, minl = node[h * 2 + 1].dist(p), minr = node[h *
             2 + 2].dist(p);
        if (minl < minr) {
            ret = search(h * 2 + 1, l, mid, p);
            if (ret > minr) ret = min(ret, search(h * 2 + 2, mid + 1,
                r, p));
        }
        else {
            ret = search(h * 2 + 2, mid + 1, r, p);
            if (ret > minl) ret = min(ret, search(h * 2 + 1, l, mid, p
                ));
        }
        return ret;
    }
    long long closest(point p) {
        return search(0, 0, vp.size() - 1, p);
    }
};
```

4.5 LiChaoSegmentTree

```
#include<cstdio>
#include<algorithm>
using namespace std;
typedef long long ll;
/*
 * credit: Youngin Cho
 * insert_line(a,b): add line y = ax+b
 * eval(x): calculate max(a_i*x + b_i)
 */
ll xmin = -1LL << 40, xmax = 1LL << 40, ymin = -1LL << 62;
struct line {
    ll a, b;
    line() :a(0), b(ymin) {}
    line(ll _a, ll _b) :a(_a), b(_b) {}
    ll eval(ll x) {
        return a*x + b;
    }
};
struct node {
    line f;
    node *left = NULL, *right = NULL;
};
struct lct_max {
    node *root = NULL;
    void update(node* &h, line li, ll l, ll r) {
        if (!h) h = new node;
        if (h->f.eval(l) < li.eval(l)) swap(h->f, li);
        if (h->f.eval(r) >= li.eval(r)) return;
        ll m = l + r >> 1;
        if (h->f.eval(m)>li.eval(m)) update(h->right, li, m + 1, r);
        else swap(h->f, li), update(h->left, li, l, m);
    }
    ll query(node *h, ll x, ll l, ll r) {
        if (!h || r < x || x < l) return ymin;
        if (l == r) return h->f.eval(x);
        ll m = l + r >> 1;
        return max({ h->f.eval(x), query(h->left, x, l, m), query(h->
            right, x, m + 1, r) });
    }
    void insert_line(line li) {
        update(root, li, xmin, xmax);
    }
    ll eval(ll x) {
        return query(root, x, xmin, xmax);
    }
};
```

4.6 PersistentSegmentTree

```
/*
 * credit: Youngin Cho
 * tree[i]: The number of elements in [l,r] with sequence[1,i]
 * O((n+m)lgL)
 * INPUT:  The order of input as follow.
 *         n = length of sequence, ai = sequence, m = number of query,
```

```
 *         (x,y,z) = query for the number of elements which is upper
     than z in a[x..y]
 * OUTPUT: answers for queries
 */
#include<cstdio>
int n, m;
struct st {
    st *left = 0, *right = 0;
    int s = 0;
}*tree[100001];
st *update(st *now, int l, int r, int g) {
    if (r < g || g < l) return now;
    st *ret = new st();
    if (l == r) ret->s = now->s + 1;
    else {
        if (!now->left) now->left = new st;
        if (!now->right) now->right = new st;
        ret->left = update(now->left, l, (l + r) / 2, g);
        ret->right = update(now->right, (l + r) / 2 + 1, r, g);
        ret->s = ret->left->s + ret->right->s;
    }
    return ret;
}
int query(st *now, int l, int r, int g) {
    if (r <= g || !now) return 0;
    if (g < l) return now->s;
    return query(now->left, l, (l + r) / 2, g) + query(now->right, (l
        + r) / 2 + 1, r, g);
}
int main() {
    scanf("%d", &n);
    tree[0] = new st;
    for (int i = 1, x; i <= n; i++) {
        scanf("%d", &x);
        tree[i] = update(tree[i - 1], 1, 1e9, x);
    }
    scanf("%d", &m);
    for (int i = 0, x, y, z; i < m; i++) {
        scanf("%d%d%d", &x, &y, &z);
        printf("%d\n", query(tree[y], 1, 1e9, z) - query(tree[x - 1],
            1, 1e9, z));
    }
    return 0;
}
```

5 Graph

5.1 CutEdges

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * given graph must be simple
 * the node index starts at 1
 * O(V+E)
 */
struct CutEdges{
    int v;
    vector<int> t;
    vector<vector<int> > adj;
    vector<pair<int,int> > res;
    CutEdges(){}
    CutEdges(int _v):v(_v),t(v+1),adj(v+1){}
    void add_edge(int x,int y){
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    void dfs(int h, int p, int d) {
        t[h] = d;
        for (int it : adj[h]) {
            if (it == p) continue;
            if (!t[it]) {
                dfs(it, h, d + 1);
                if (t[it] == d + 1) res.push_back({h, it});
            }
            t[h] = min(t[h], t[it]);
        }
    }
    vector<pair<int,int> > solve(){
        for(int i=1; i<=v; i++) if(!t[i]) dfs(i,-1,1);
        return res;
    }
};
```

5.2 CutVertices

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * given graph must be simple
 * the node index starts at 1
 * O(V+E)
 */
struct CutVertices{
    int v,cnt=0;
    vector<int> ck,cutv;
    vector<vector<int> > adj;
    CutVertices(){}
    CutVertices(int _v):v(_v),ck(v+1),cutv(v+1),adj(v+1){}
    void add_edge(int x,int y){
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    void dfs(int h, bool rt) {
        ck[h] = ++cnt;
        int vcnt = 0, mini = v;
        for (auto t : adj[h]) {
            if (!ck[t]) {
                dfs(t, false);
                if (ck[t] == ck[h]) cutv[h] = true;
                vcnt++;
            }
            mini = min(mini, ck[t]);
        }
        ck[h] = mini;
        if (rt) cutv[h] = vcnt >= 2;
```

```
    }
    vector<int> solve(){
        vector<int> res;
        for (int i = 1; i <= v; i++) if (!ck[i]) dfs(i, true);
        for (int i = 1; i <= v; i++) if(cutv[i]) res.push_back(i);
        return res;
    }
};
```

5.3 Delaunay@

```
// credit: https://github.com/ayush-tulsyan/ACM-ICPC-Handbook/blob/
    master/code/Delaunay.cc

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

#include<vector>
using namespace std;

typedef double T;

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

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

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

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

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

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

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

5.4 DirectedMST@

```
// Directed minimum spanning tree
//
// Given a directed weighted graph and root node, computes the minimum
     spanning
// directed tree (arborescence) on it.
//
// Complexity: O(N * M), where N is the number of nodes, and M the
    number of edges

struct Edge { int x, y, w; };

int dmst(int N, vector<Edge> E, int root) {
  const int oo = 1e9;

  vector<int> cost(N), back(N), label(N), bio(N);
  int ret = 0;

  for (;;) {
    REP(i, N) cost[i] = oo;
    for (auto e : E) {
      if (e.x == e.y) continue;
      if (e.w < cost[e.y]) cost[e.y] = e.w, back[e.y] = e.x;
    }
    cost[root] = 0;
    REP(i, N) if (cost[i] == oo) return -1;
    REP(i, N) ret += cost[i];

    int K = 0;
    REP(i, N) label[i] = -1;
    REP(i, N) bio[i] = -1;

    REP(i, N) {
      int x = i;
      for (; x != root && bio[x] == -1; x = back[x]) bio[x] = i;

      if (x != root && bio[x] == i) {
```

```
        for (; label[x] == -1; x = back[x]) label[x] = K;
        ++K;
      }
    }
    if (K == 0) break;

    REP(i, N) if (label[i] == -1) label[i] = K++;

    for (auto &e : E) {
      int xx = label[e.x];
      int yy = label[e.y];
      if (xx != yy) e.w -= cost[e.y];
      e.x = xx;
      e.y = yy;
    }

    root = label[root];
    N = K;
  }

  return ret;
}
```

5.5 EulerianCircuit@

```
#include<cstdio>
/*
 * credit: Youngin Cho
 * test: https://www.acmicpc.net/problem/1199
 * O(V+E)
 * adj[i][j]: the number of edges in i->j
 */
const int MAX_N = 1e3;
int n, adj[MAX_N][MAX_N], it[MAX_N];
void dfs(int h) {
    for (int &i = it[h]; i < n; i++) while (i < n && adj[h][i]) {
        adj[h][i]--;
        adj[i][h]--;
        dfs(i);
    }
    printf("%d ", h + 1);
}
int main() {
    scanf("%d", &n);
    for (int i = 0, c = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &adj[i][j]);
            c += adj[i][j];
        }
        if (c & 1) {
            puts("-1");
            return 0;
        }
    }
    dfs(0);
    return 0;
}
```

5.6 SCC

```
#include<bits/stdc++.h>
using namespace std;
/*
 * credit: Youngin Cho
 * the node index starts at 1
 * O(V+E)
 */
struct SCC{
    vector<vector<int> > adj[2];
    vector<int> vis,lst;
    int v;
    SCC(){}
    SCC(int _v){
        v=_v;
        adj[0].resize(v+1);
        adj[1].resize(v+1);
        vis.resize(v+1);
    }
    void add_edge(int x,int y){
        adj[1][x].push_back(y);
        adj[0][y].push_back(x);
    }
    void dfs(int h,int t){
        if(vis[h]==t) return;
        vis[h]=t;
        for(int it:adj[t][h]) dfs(it,t);
        lst.push_back(h);
    }
    vector<vector<int> > solve(){
        vector<vector<int> > res;
        for(int i=1; i<=v; i++) if(!vis[i]) dfs(i,1);
        vector<int> tlst=lst;
        reverse(tlst.begin(),tlst.end());
        for(auto it:tlst) if(vis[it]){
            lst.clear();
            dfs(it,0);
            res.push_back(lst);
        }
        return res;
    }
};
```

5.7 2-Satisfiability

```
#include<cstdio>
#include<vector>
using namespace std;
/*
* O(V+E)
* TwoSat(v): use 0~2v-1 as node index, not x corresponds to (x+v)%(2*v
    )
* add_or(x,y): add (x|y)
* solve(res): return the result if it is vaild(0: false or 1: true),
*             res[i]: the logical value of node i(0~v-1) (0: false or
    1: true)
*/
```

```
struct TwoSat {
    int v;
    vector<vector<int> > adj[2];
    vector<int> vis, lst;
    TwoSat(){}
    TwoSat(int _v) {
        v = _v;
        adj[0].resize(v * 2);
        adj[1].resize(v * 2);
        vis.resize(v * 2);
    }
    void dfs(int h, int t) {
        if (vis[h] == t) return;
        vis[h] = t;
        for (int it : adj[t][h]) dfs(it, t);
        lst.push_back(h);
    }
    void add_edge(int x, int y) {
        adj[1][x].push_back(y);
        adj[0][y].push_back(x);
    }
    void add_or(int x, int y) {
        add_edge((x + v)%(2*v), y);
        add_edge((y + v)%(2*v), x);
    }
    int solve(vector<int> &res) {
        for (int i = 0; i < v * 2; i++) if (!vis[i]) dfs(i, 1);
        vector<int> la(v * 2), tlst = lst;
        for (int i = tlst.size() - 1; i >= 0; i--) if (vis[tlst[i]]) {
            lst.clear();
            dfs(tlst[i], 0);
            for (int &it : lst) la[it] = i;
        }
        for (int i = 0; i < v; i++)
            if (la[i] == la[i + v]) return 0;
        res.resize(v);
        for (int i = 0; i < v; i++) res[i] = la[i] < la[i + v];
        return 1;
    }
};
```

5.8 MaxClique@

```
// credit: https://www.dropbox.com/s/sff4hula5gc60do/MolaMola2018.docx
    ?dl=0
ll G[40]; // 0-index
    void get_clique(int R = 0, ll P = (1ll<<N)-1, ll X = 0){
      if((P|X) == 0){
        cur = max(cur, R);
        return;
      }
      int u = __builtin_ctzll(P|X);
      ll c = P&~G[u];
      while(c){
        int v = __builtin_ctzll(c);
        get_clique(R + 1, P&G[v], X&G[v]);
        P ^= 1ll << v;
        X |= 1ll << v;
        c ^= 1ll << v;
      }
```

6 Numerical Algorithm

6.1 DifferenceConstraints@

```
#include<bits/stdc++.h>
using namespace std;
/*
 * IES(n): use 1..n as node index
 * Before executing solve(..), you must execute set_range(x,..) for x
    =1..n
 * solve(): returns the values of ith variable
 *          if 0th value is lower than 0, then the system is invalid
 */
const int mx = 1e9;
struct IES {
    vector<vector<pair<int,int> > > adj;
    IES(int n) : adj(n + 1) {}
    void add_inequlity(int x, int y, int c) { // x-y<=c
        adj[y].push_back({ x,c });
    }
    void add_equlity(int x, int y, int c) { // x-y=c
        add_inequlity(x, y, c);
        add_inequlity(y, x, -c);
    }
    void set_range(int x, int mini, int maxi) { // mini<=x<=maxi
        add_inequlity(0, x, -mini);
        add_inequlity(x, 0, maxi);
    }
    vector<int> solve() {
        vector<int> dis(adj.size(), mx), ck(adj.size());
        queue<int> q;
        dis[0] = 0;
        ck[0] = 1;
        q.push(0);
        while (!q.empty() && !dis[0]) {
            int h = q.front(); q.pop();
            ck[h] = 0;
            for (auto &it : adj[h]) if (dis[it.first] > dis[h] + it.
                second) {
                dis[it.first] = dis[h] + it.second;
                if (!ck[it.first]) {
                    ck[it.first] = 1;
                    q.push(it.first);
                }
            }
        }
        return dis;
    }
};
```

6.2 CountPrimes@

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

```
// credit : https://github.com/stjepang/snippets/blob/master/
    count_primes.cpp
// Primes up to 10^12 can be counted in ~1 second.

// Count prime numbers up to N
// To initialize, call init_count_primes() first.
// Function count_primes(N) will compute the number of prime numbers
    lower than
// or equal to N.
// Time complexity: Around O(N ^ 0.75)

typedef long long lint;
const int MAXN = 1000005; // MAXN is the maximum value of sqrt(N) + 2
bool prime[MAXN];
int prec[MAXN];
vector<int> P;
void init() {
    prime[2] = true;
    for (int i = 3; i < MAXN; i += 2) prime[i] = true;
    for (int i = 3; i*i < MAXN; i += 2) {
        if (prime[i]) {
            for (int j = i * i; j < MAXN; j += i + i) prime[j] = false
                ;
        }
    }
    for (int i = 1; i < MAXN; i++) {
        if (prime[i]) P.push_back(i);
        prec[i] = prec[i - 1] + prime[i];
    }
}
lint rec(lint N, int K) {
    if (N <= 1 || K < 0) return 0;
    if (N <= P[K]) return N - 1;
    if (N < MAXN && 1ll * P[K] * P[K] > N) return N - 1 - prec[N] +
        prec[P[K]];
    const int LIM = 250;
    static int memo[LIM*LIM][LIM];
    bool ok = N < LIM*LIM;
    if (ok && memo[N][K]) return memo[N][K];
    lint ret = N / P[K] - rec(N / P[K], K - 1) + rec(N, K - 1);
    if (ok) memo[N][K] = ret;
    return ret;
}
lint count_primes(lint N) { //less than or equal to
    if (N < MAXN) return prec[N];
    int K = prec[(int)sqrt(N) + 1];
    return N - 1 - rec(N, K) + prec[P[K]];
}
```

6.3 DivRound@

```
// Integer division with rounding down/up

llint div_floor(llint a, llint b) {
  if (b < 0) a = -a, b = -b;
  return a/b - (a%b < 0);
}

llint div_ceil(llint a, llint b) {
  if (b < 0) a = -a, b = -b;
  return a/b + (a%b > 0);
}
```

6.4 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<bits/stdc++.h>

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)
 * if a or b is negative, gcd(a,b) may be negative.
 */
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
// it must ensure {value}^2 in type range
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 d = gcd(a, b); finds x, y such that d = ax + by
 * if a or b is negative, d may be negative.
 * let g=abs(d), then it ensures abs(x)*g <= max(abs(b),g), abs(y)*g
     <= max(abs(a),g)
 * thus, for d!=0, abs(x) <= max(abs(b/d),1), abs(y) <= max(abs(a/d)
     ,1)
 */
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() {
    int tc = clock();
    for (int i = -1000000; i <= 1000000; i++) {
        for (int j = -1000000; j <= 1000000; j++) {
            int t = gcd_recursive(i, j);
        }
    }
    printf("%dms\n", clock() - tc);

    /*
    // 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;
}
```

6.5 FFT

```
#include<cstdio>
#include<complex>
#include<vector>
using namespace std;
/*
 * credit: Youngin Cho
 */
void FFT(vector<complex<double> > &a, int inv) {
    int sz = a.size();
    for (int i = 0; i < sz; i++) {
        int j = 0;
        for (int k = 0; 1 << k < sz; k++) j = j << 1 | i >> k & 1;
        if (i < j) swap(a[i], a[j]);
    }
    complex<double> w, b, u, v;
    for (int i = 1; i < sz; i <<= 1) {
        w = polar(1.0, inv * acos(-1) / i);
        for (int j = 0; j < sz; j += i << 1) {
            b = 1;
            for (int k = j; k < j + i; k++) {
                u = a[k]; v = b*a[k + i];
                a[k] = u + v;
                a[k + i] = u - v;
                b *= w;
            }
        }
    }
}
vector<int> multiply(vector<int> &a, vector<int> &b) {
    vector<complex<double> > u(a.begin(),a.end()), v(b.begin(),b.end()
        );
    int sz = 1;
    while (sz < max(a.size(),b.size())) sz <<= 1;
    sz <<= 1;
    u.resize(sz); v.resize(sz);
    FFT(u, 1); FFT(v, 1);
    for (int i = 0; i < sz; i++) u[i] *= v[i];
    FFT(u, -1);
    for (int i = 0; i < sz; i++) u[i] /= sz;
    vector<int> ret(sz);
    for (int i = 0; i < sz; i++) ret[i] = floor(real(u[i]) + 0.5);
    return ret;
}
```

6.6 GaussMod@

```
// Modular Gaussian elimination
//
// Solves systems of linear modular equations.
//
// To use, build a matrix of coefficients and call run(mat, R, C, mod)
    .
// If there is no solution, -1 will be returned, otherwise the number
    of free
// variables will be returned.
// If the i-th variable is free, row[i] will be -1, otherwise it's
    value will
// be ans[i].
//
// Time complexity: O(R * C^2)
//
// Constants to configure:
// - MAXC is the number of columns

namespace Gauss {
  const int MAXC = 1001;

  int row[MAXC];
  llint ans[MAXC];

  llint power(llint a, llint b, llint mod) {
    llint ret = 1;
    for (; b; b /= 2) {
      if (b % 2) ret = ret*a % mod;
      a = a*a % mod;
    }
    return ret;
  }
  llint inv(llint x, llint mod) { return power(x, mod-2, mod); }

  int run(llint mat[][MAXC], int R, int C, llint mod) {
    REP(i, C) row[i] = -1;

    int r = 0;
    REP(c, C) {
      int k = r;
      while (k < R && mat[k][c] == 0) ++k;
      if (k == R) continue;

      REP(j, C+1) swap(mat[r][j], mat[k][j]);
      llint div = inv(mat[r][c], mod);

      REP(i, R) if (i != r) {
        llint w = mat[i][c] * (mod - div) % mod;
        REP(j, C+1) mat[i][j] = (mat[i][j] + mat[r][j] * w) % mod;
      }
      row[c] = r++;
    }

    REP(i, C) {
      int r = row[i];
      ans[i] = r == -1 ? 0 : mat[r][C] * inv(mat[r][i], mod) % mod;
    }

    FOR(i, r, R) if (mat[i][C]) return -1;
    return C - r;
  }
}
```

6.7 Kitamasa@

```
#include <cstdio>
```

```
/*
 * credit: http://cubelover.tistory.com/21?category=211404
 * Given a_n = sum( a_(n-k)*C_k : k=1..m )
 * Kitamasa calculate the nth term of the sequence in O(m^2*lgn)
 */

const int P = 1000000007;

int a[1001];
int c[1001];
int d[1001];
int t[2002];

void Kitamasa(int n, int m) {
    int i, j;
    if (n == 1) {
        d[1] = 1;
        for (i = 2; i <= m; i++) d[i] = 0;
        return;
    }
    if (n & 1) {
        Kitamasa(n ^ 1, m);
        j = d[m];
        for (i = m; i >= 1; i--) d[i] = (d[i - 1] + (long long)c[m - i
             + 1] * j) % P;
        return;
    }
    Kitamasa(n >> 1, m);
    for (i = 1; i <= m + m; i++) t[i] = 0;
    for (i = 1; i <= m; i++) for (j = 1; j <= m; j++) t[i + j] = (t[i
        + j] + (long long)d[i] * d[j]) % P;
    for (i = m + m; i > m; i--) for (j = 1; j <= m; j++) t[i - j] = (t
        [i - j] + (long long)c[j] * t[i]) % P;
    for (i = 1; i <= m; i++) d[i] = t[i];
}

int main() {
    int i, n, m, r;
    scanf("%d%d", &n, &m);
    for (i = 1; i <= m; i++) scanf("%d", &a[i]);
    for (i = 1; i <= m; i++) scanf("%d", &c[i]);
    Kitamasa(n, m);
    r = 0;
    for (i = 1; i <= m; i++) r = (r + (long long)a[i] * d[i]) % P;
    printf("%d", r);
}
```

6.8 MobiusFunction

```
#include<cstdio>

// credit: Youngin Cho

typedef long long ll;
const long long MX = 1e6;
int mo[MX + 1];
void mobius() {
    mo[1] = 1;
    for (int i = 1; i <= MX; i++) {
        for (int j = i << 1; j <= MX; j += i) mo[j] -= mo[i];
    }
}

/*
 * return the number of non-squarefree in 1^x. (mobius() must be
     executed in advance)
 * A positive integer k is called squarefree
 * if k is not divisible by d^2 for any d > 1
 */
ll count(ll x) {
    ll r = 0;
    for (ll i = 2; i*i <= x; i++) {
        r -= x / (i*i) * mo[i];
    }
    return r;
}
```

6.9 NTT@

```
#include <cstdio>

/*
 *
 * credit: http://cubelover.tistory.com/12?category=211404
 * Usually we use w = cos(2pi/k) + i sin(2pi/k) when we compute FFT of
     2^n size array
 * Note that k is the power of 2 and w^k = 1
 * Consider a prime p = a * 2^b + 1 and let x be the primitive root of
     p.
 * (x^a)^(2^b) mod p = 1
 * By using (x^a)^(2^b/k) instead of w, we can compute FFT of 2^n size
     array for n<=b.
 * Below is the table of big prime satisfying the condition.
 * |-------------------------------------------------------------------|
 * |      p      |  a  |  b  |  x  |    addition    | multiplication |
 * |-------------------------------------------------------------------|
 * | 3221225473  | 3   | 30  | 5   | 64bit signed   | 64bit unsigned |
 * | 2281701377  | 17  | 27  | 3   | 64bit signed   | 64bit signed   |
 * | 2013265921  | 15  | 27  | 31  | 32bit unsigned | 64bit signed   |
 * | 469762049   | 7   | 26  | 3   | 32bit signed   | 64bit signed   |
 * |-------------------------------------------------------------------|
 *
 * koosaga said "998244353 = 119 * 2^23 + 1 is also prime and has 3 as
     a primitive root."
 *
 */

const int A = 7, B = 26, P = A << B | 1, R = 3;
const int SZ = 20, N = 1 << SZ;

int Pow(int x, int y) {
        int r = 1;
        while (y) {
                if (y & 1) r = (long long)r * x % P;
                x = (long long)x * x % P;
                y >>= 1;
        }
        return r;
}
```

```
void FFT(int *a, bool f) {
        int i, j, k, x, y, z;
        j = 0;
        for (i = 1; i < N; i++) {
                for (k = N >> 1; j >= k; k >>= 1) j -= k;
                j += k;
                if (i < j) {
                        k = a[i];
                        a[i] = a[j];
                        a[j] = k;
                }
        }
        for (i = 1; i < N; i <<= 1) {
                x = Pow(f ? Pow(R, P - 2) : R, P / i >> 1);
                for (j = 0; j < N; j += i << 1) {
                        y = 1;
                        for (k = 0; k < i; k++) {
                                z = (long long)a[i | j | k] * y % P;
                                a[i | j | k] = a[j | k] - z;
                                if (a[i | j | k] < 0) a[i | j | k] +=
                                    P;
                                a[j | k] += z;
                                if (a[j | k] >= P) a[j | k] -= P;
                                y = (long long)y * x % P;
                        }
                }
        }
        if (f) {
                j = Pow(N, P - 2);
                for (i = 0; i < N; i++) a[i] = (long long)a[i] * j % P
                    ;
        }
}

int X[N];

int main() {
        int i, n;
        scanf("%d", &n);
        for (i = 0; i <= n; i++) scanf("%d", &X[i]);
        FFT(X, false);
        for (i = 0; i < N; i++) X[i] = (long long)X[i] * X[i] % P;
        FFT(X, true);
        for (i = 0; i <= n + n; i++) printf("%d ", X[i]);
}
```

6.10 NumberTheory@

```
// credit: https://github.com/niklasb/contest-algos/

ll multiply_mod(ll a, ll b, ll mod) {
  if (b == 0) return 0;
  if (b & 1) return ((ull)multiply_mod(a, b-1, mod) + a) % mod;
  return multiply_mod(((ull)a + a) % mod, b/2, mod);
}

ll powmod(ll a, ll n, ll mod) {
  if (n == 0) return 1 % mod;
  if (n & 1) return multiply_mod(powmod(a, n-1, mod), a, mod);
  return powmod(multiply_mod(a, a, mod), n/2, mod);
}

// simple modinv, returns 0 if inverse doesn't exist
ll modinv(ll a, ll m) {
  return a < 2 ? a : ((1 - m * 1ll * modinv(m % a, a)) / a % m + m) %
      m;
}
ll modinv_prime(ll a, ll p) { return powmod(a, p-2, p); }

tuple<ll,ll,ll> egcd(ll a, ll b) {
  if (!a) return make_tuple(b, 0, 1);
  ll g, y, x;
  tie(g, y, x) = egcd(b % a, a);
  return make_tuple(g, x - b/a * y, y);
}

// solve the linear equation a x == b (mod n)
// returns the number of solutions up to congruence (can be 0)
//    sol: the minimal positive solution
//    dis: the distance between solutions
ll linear_mod(ll a, ll b, ll n, ll &sol, ll &dis) {
  a = (a % n + n) % n, b = (b % n + n) % n;
  ll d, x, y;
  tie(d, x, y) = egcd(a, n);
  if (b % d)
    return 0;
  x = (x % n + n) % n;
  x = b / d * x % n;
  dis = n / d;
  sol = x % dis;
  return d;
}

bool rabin(ll n) {
  // bases chosen to work for all n < 2^64, see https://oeis.org/
      A014233
  set<int> p { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
  if (n <= 37) return p.count(n);
  ll s = 0, t = n - 1;
  while (~t & 1)
    t >>= 1, ++s;
  for (int x: p) {
    ll pt = powmod(x, t, n);
    if (pt == 1) continue;
    bool ok = 0;
    for (int j = 0; j < s && !ok; ++j) {
      if (pt == n - 1) ok = 1;
      pt = multiply_mod(pt, pt, n);
    }
    if (!ok) return 0;
  }
  return 1;
}

ll rho(ll n) { // will find a factor < n, but not necessarily prime
  if (~n & 1) return 2;
  ll c = rand() % n, x = rand() % n, y = x, d = 1;
  while (d == 1) {
    x = (multiply_mod(x, x, n) + c) % n;
    y = (multiply_mod(y, y, n) + c) % n;
```

```
    y = (multiply_mod(y, y, n) + c) % n;
    d = __gcd(abs(x - y), n);
  }
  return d == n ? rho(n) : d;
}

void factor(ll n, map<ll, int> &facts) {
  if (n == 1) return;
  if (rabin(n)) {
    facts[n]++;
    return;
  }
  ll f = rho(n);
  factor(n/f, facts);
  factor(f, facts);
}

// use inclusion-exclusion to get the number of integers <= n
// that are not divisable by any of the given primes.
// This essentially enumerates all the subsequences and adds or
    subtracts
// their product, depending on the current parity value.
ll count_coprime_rec(int primes[], int len, ll n, int i, ll prod, bool
     parity) {
  if (i >= len || prod * primes[i] > n) return 0;
  return (parity ? 1 : (-1)) * (n / (prod*primes[i]))
        + count_coprime_rec(primes, len, n, i + 1, prod, parity)
        + count_coprime_rec(primes, len, n, i + 1, prod * primes[i], !
            parity);
}
// use cnt(B) - cnt(A-1) to get matching integers in range [A..B]
ll count_coprime(int primes[], int len, ll n) {
  if (n <= 1) return max(0LL, n);
  return n - count_coprime_rec(primes, len, n, 0, 1, true);
}

// find x.  a[i] x = b[i] (mod m[i])  0 <= i < n. m[i] need not be
    coprime
bool crt(int n, ll *a, ll *b, ll *m, ll &sol, ll &mod) {
  ll A = 1, B = 0, ta, tm, tsol, tdis;
  for (int i = 0; i < n; ++i) {
    if (!linear_mod(a[i], b[i], m[i], tsol, tdis)) return 0;
    ta = tsol, tm = tdis;
    if (!linear_mod(A, ta - B, tm, tsol, tdis)) return 0;
    B = A * tsol + B;
    A = A * tdis;
  }
  sol = B, mod = A;
  return 1;
}

// get number of permutations {P_1, ..., P_n} of size n,
// where no number is at its original position (that is, P_i != i for
    all i)
// also called subfactorial !n
ll get_derangement_mod_m(ll n, ll m) {
  vector<ll> res(m * 2);
  ll d = 1 % m, p = 1;
  res[0] = d;
  for (int i = 1; i <= min(n, 2 * m - 1); ++i) {
    p *= -1;
    d = (1LL * i * d + p + m) % m;
    res[i] = d;
    if (i == n) return d;
  }
  // it turns out that !n mod m == !(n mod 2m) mod m
  return res[n % (2 * m)];
}

// compute totient function for integers <= n
vector<int> compute_phi(int n) {
  vector<int> phi(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    phi[i] += i;
    for (int j = 2 * i; j <= n; j += i) {
      phi[j] -= phi[i];
    }
  }
  return phi;
}

// checks if g is primitive root mod p. Generate random g's to find
    primitive root.
bool is_primitive(ll g, ll p) {
  map<ll, int> facs;
  factor(p - 1, facs);
  for (auto& f : facs)
    if (1 == powmod(g, (p-1)/f.first, p))
      return 0;
  return 1;
}

ll dlog(ll g, ll b, ll p) { // find x such that g^x = b (mod p)
  ll m = (ll)(ceil(sqrt(p-1))+0.5); // better use binary search here
      ...
  unordered_map<ll,ll> powers; // should compute this only once per g
  rep(j,0,m) powers[powmod(g, j, p)] = j;
  ll gm = powmod(g, -m + 2*(p-1), p);
  rep(i,0,m) {
    if (powers.count(b)) return i*m + powers[b];
    b = b * gm % p;
  }
  assert(0); return -1;
}

// compute p(n,k), the number of possibilities to write n as a sum of
// k non-zero integers
ll count_partitions(int n, int k) {
  if (n==k) return 1;
  if (n<k || k==0) return 0;
  vector<ll> p(n + 1);
  for (int i = 1; i <= n; ++i) p[i] = 1;
  for (int l = 2; l <= k; ++l)
    for (int m = l+1; m <= n-l+1; ++m)
      p[m] = p[m] + p[m-l];
  return p[n-k+1];
}
```

6.11 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;
}
```

7 Cheat Sheet

7.1 DynamicProgrammingOptimizations

```
credit: https://codeforces.com/blog/entry/8219

A[i][j] - the smallest k that gives optimal answer, for example in dp[
    i][j] = dp[i - 1][k] + C[k][j]
C[i][j] - some given cost function

<Divide and Conquer Optimization>
Original Recurrence : dp[i][j] = min{dp[i - 1][k] + C[k][j] : k < j}
Sufficient Condition: A[i][j] <= A[i][j + 1]
Original Complexity : O(kn^2)
Optimized Complexity: O(knlogn)

<Knuth Optimization>
Original Recurrence : dp[i][j] = min{dp[i][k] + dp[k][j] : i < k < j}
    + C[i][j]
Sufficient Condition: A[i, j - 1] <= A[i, j] <= A[i + 1, j]
Original Complexity : O(n^3)
Optimized Complexity: O(n^2)
"
It is claimed (in the references) that Knuth Optimization is
    applicable if C[i][j] satisfies the following 2 conditions:
quadrangle inequality: C[a][c] + C[b][d] <= C[a][d] + C[b][c], a <= b
    <= c <= d
monotonicity: C[b][c] <= C[a][d], a <= b <= c <= d
"
```

7.2 PrimeTable

```
credit: https://primes.utm.edu/lists/2small/0bit.html
five least k's for which 2^n-k is prime.
 n               k          |  n                  k
-------------------------------------------------------------
 8       5, 15, 17, 23, 27  | 36         5, 17, 23, 65,117
 9       3,  9, 13, 21, 25  | 37        25, 31, 45, 69,123
10       3,  5, 11, 15, 27  | 38        45, 87,107,131,153
11       9, 19, 21, 31, 37  | 39         7, 19, 67, 91,135
12       3,  5, 17, 23, 39  | 40        87,167,195,203,213
13       1, 13, 21, 25, 31  | 41        21, 31, 55, 63, 73
14       3, 15, 21, 23, 35  | 42        11, 17, 33, 53, 65
15      19, 49, 51, 55, 61  | 43        57, 67,117,175,255
16      15, 17, 39, 57, 87  | 44        17,117,119,129,143
17       1,  9, 13, 31, 49  | 45        55, 69, 81, 93,121
18       5, 11, 17, 23, 33  | 46        21, 57, 63, 77,167
19       1, 19, 27, 31, 45  | 47       115,127,147,279,297
20       3,  5, 17, 27, 59  | 48        59, 65, 89, 93,147
21       9, 19, 21, 55, 61  | 49        81,111,123,139,181
22       3, 17, 27, 33, 57  | 50        27, 35, 51, 71,113
23      15, 21, 27, 37, 61  | 51       129,139,165,231,237
24       3, 17, 33, 63, 75  | 52        47,143,173,183,197
25      39, 49, 61, 85, 91  | 53       111,145,231,265,315
26       5, 27, 45, 87,101  | 54        33, 53,131,165,195
27      39, 79,111,115,135  | 55        55, 67, 99,127,147
28      57, 89, 95,119,125  | 56         5, 27, 47, 57, 89
29       3, 33, 43, 63, 73  | 57        13, 25, 49, 61, 69
30      35, 41, 83,101,105  | 58        27, 57, 63,137,141
31       1, 19, 61, 69, 85  | 59        55, 99,225,427,517
32       5, 17, 65, 99,107  | 60        93,107,173,179,257
33       9, 25, 49, 79,105  | 61         1, 31, 45,229,259
34      41, 77,113,131,143  | 62        57, 87,117,143,153
35      31, 49, 61, 69, 79  | 63        25,165,259,301,375
```