Luna's Magic Reference

Suzune Nisiyama

July 14, 2018

Contents

1 Data Structure

1.1 KD tree

```
/* kd_tree : finds the k-th closest point in O(kn^{1-1/k}).
Usage : Stores the data in p[]. Call function init (n, k). Call min_kth (d, k). (or max_kth) (k is 1-based)
Note : Switch to the commented code for Manhattan distance.
Status : SPOJ-FAILURE Accepted.*/
template <int MAXN = 200000, int MAXK = 2>
struct kd_tree {
 int k, size;
 struct point { int data[MAXK], id; } p[MAXN];
 struct kd_node {
  int l, r; point p, dmin, dmax;
  kd_node() {}
  kd_node (const point &rhs) : l (-1), r (-1), p (rhs), dmin (rhs), dmax (rhs) {}
  void merge (const kd_node &rhs, int k) {
   for (register int i = 0; i < k; ++i) {
    dmin.data[i] = std::min (dmin.data[i], rhs.dmin.data[i]);
    dmax.data[i] = std::max (dmax.data[i], rhs.dmax.data[i]); } }
  long long min_dist (const point &rhs, int k) const {
   register long long ret = 0;
   for (register int i = 0; i < k; ++i) {
    if (dmin.data[i] <= rhs.data[i] && rhs.data[i] <= dmax.data[i]) continue;
    ret += std::min (1ll * (dmin.data[i] - rhs.data[i]) * (dmin.data[i] - rhs.data[i]),
     1ll * (dmax.data[i] - rhs.data[i]) * (dmax.data[i] - rhs.data[i]));
//    ret += std::max (0, rhs.data[i] - dmax.data[i]) + std::max (0, dmin.data[i] - rhs.data[i]);
   } return ret; }
  long long max_dist (const point &rhs, int k) {
   long long ret = 0;
   for (int i = 0; i < k; ++i) {
    int tmp = std::max (std::abs (dmin.data[i] - rhs.data[i]), std::abs (dmax.data[i] - rhs.data[i]));
    ret += 1ll * tmp * tmp; }
//    ret += std::max (std::abs (rhs.data[i] - dmax.data[i]) + std::abs (rhs.data[i] - dmin.data[i])); }
   return ret; } } tree[MAXN * 4];
 struct result {
  long long dist; point d; result() {}
  result (const long long &dist, const point &d) : dist (dist), d (d) {}
  bool operator > (const result &rhs) const { return dist > rhs.dist || (dist == rhs.dist && d.id > rhs.d.id); }
  bool operator < (const result &rhs) const { return dist < rhs.dist || (dist == rhs.dist && d.id < rhs.d.id); } };
 long long sqrdist (const point &a, const point &b) {
  long long ret = 0;
  for (int i = 0; i < k; ++i) ret += 1ll * (a.data[i] - b.data[i]) * (a.data[i] - b.data[i]);
//  for (int i = 0; i < k; ++i) ret += std::abs (a.data[i] - b.data[i]);
  return ret; }
 int alloc() { tree[size].l = tree[size].r = -1; return size++; }
 void build (const int &depth, int &rt, const int &l, const int &r) {
  if (l > r) return;
  register int middle = (l + r) >> 1;
  std::nth_element (p + l, p + middle, p + r + 1, [=] (const point & a, const point & b) { return a.data[depth] < b.data[depth]; });
  tree[rt = alloc()] = kd_node (p[middle]);
  if (l == r) return;
  build ((depth + 1) % k, tree[rt].l, l, middle - 1);
  build ((depth + 1) % k, tree[rt].r, middle + 1, r);
  if (~tree[rt].l) tree[rt].merge (tree[tree[rt].l], k);
  if (~tree[rt].r) tree[rt].merge (tree[tree[rt].r], k); }
 std::priority_queue<result, std::vector<result>, std::less <result>> heap_l;
 std::priority_queue<result, std::vector<result>, std::greater <result>> heap_r;
 void _min_kth (const int &depth, const int &rt, const int &m, const point &d) {
  result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
  if ((int)heap_l.size() < m) heap_l.push (tmp);
  else if (tmp < heap_l.top()) {
   heap_l.pop();
   heap_l.push (tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y && sqrdist (d, tree[x].p) > sqrdist (d, tree[y].p)) std::swap (x, y);
  if (~x && ((int)heap_l.size() < m || tree[x].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_l.size() < m || tree[y].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, y, m, d); }
 void _max_kth (const int &depth, const int &rt, const int &m, const point &d) {
  result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
  if ((int)heap_r.size() < m) heap_r.push (tmp);
  else if (tmp > heap_r.top()) {
   heap_r.pop();
   heap_r.push (tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y && sqrdist (d, tree[x].p) < sqrdist (d, tree[y].p)) std::swap (x, y);
  if (~x && ((int)heap_r.size() < m || tree[x].max_dist (d, k) >= heap_r.top().dist))
   _max_kth ((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_r.size() < m || tree[y].max_dist (d, k) >= heap_r.top().dist))
   _max_kth ((depth + 1) % k, y, m, d); }
 void init (int n, int k) { this -> k = k; size = 0; int rt = 0; build (0, rt, 0, n - 1); }
 result min_kth (const point &d, const int &m) { heap_l = decltype (heap_l) (); _min_kth (0, 0, m, d); return heap_l.top (); }
 result max_kth (const point &d, const int &m) { heap_r = decltype (heap_r) (); _max_kth (0, 0, m, d); return heap_r.top (); } };
```

1.2 Splay

```
void push_down (int x) {
 if (~n[x].c[0]) push (n[x].c[0], n[x].t);
 if (~n[x].c[1]) push (n[x].c[1], n[x].t);
 n[x].t = tag (); }
void update (int x) {
 n[x].m = gen (x);
 if (~n[x].c[0]) n[x].m = merge (n[n[x].c[0]].m, n[x].m);
 if (~n[x].c[1]) n[x].m = merge (n[x].m, n[n[x].c[1]].m); }
void rotate (int x, int k) {
 push_down (x); push_down (n[x].c[k]);
 int y = n[x].c[k]; n[x].c[k] = n[y].c[k ^ 1]; n[y].c[k ^ 1] = x;
 if (n[x].f != -1) n[n[x].f].c[n[n[x].f].c[1] == x] = y;
 n[y].f = n[x].f; n[x].f = y; if (~n[x].c[k]) n[n[x].c[k]].f = x;
 update (x); update (y); }
void splay (int x, int s = -1) {
 push_down (x);
 while (n[x].f != s) {
  if (n[n[x].f].f != s) rotate (n[n[x].f].f, n[n[n[x].f].f].c[1] == n[x].f);
  rotate (n[x].f, n[n[x].f].c[1] == x); }
 update (x);
 if (s == -1) root = x; }
```

1.3 Link-cut tree

```
void access (int x) {
 int u = x, v = -1;
 while (u != -1) {
  splay (u); push_down (u);
  if (~n[u].c[1]) n[n[u].c[1]].f = -1, n[n[u].c[1]].p = u;
  n[u].c[1] = v;
  if (~v) n[v].f = u, n[v].p = -1;
  update (u); u = n[v = u].p; }
 splay (x); }
```

2 Formula

2.1 Zellers congruence

```
/* Zeller's congruence : converts between a calendar date and its Gregorian calendar day. (y >= 1) (0 = Monday, 1 = Tuesday, ..., 6 = Sunday) */
int get_id (int y, int m, int d) {
 if (m < 3) { --y; m += 12; }
 return 365 * y + y / 4 - y / 100 + y / 400 + (153 * (m - 3) + 2) / 5 + d - 307; }
std::tuple <int, int, int> date (int id) {
 int x = id + 1789995, n, i, j, y, m, d;
 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;
 return std::make_tuple (y, m, d); }
```

2.2 Lattice points below segment

```
/* Euclidean-like algorithm : computes the sum of \sum_{i=0}^{n-1} \lfloor (a+bi)/m \rfloor.*/
long long solve(long long n, long long a, long long b, long long m){
 if (b == 0) return n * (a / m);
 if (a >= m) return n * (a / m) + solve (n, a % m, b, m);
 if (b >= m) return (n - 1) * n / 2 * (b / m) + solve (n, a, b % m, m);
 return solve ((a + b * n) / m, (a + b * n) % m, m, b); }
```

2.3 Adaptive Simpson's method

```
/* Adaptive Simpson's method : integrates f in [l, r]. */
struct simpson {
 double area (double (*f) (double), double l, double r) {
  double m = l + (r - l) / 2;
  return (f (l) + 4 * f (m) + f (r)) * (r - l) / 6; }
 double solve (double (*f) (double), double l, double r, double eps, double a) {
  double m = l + (r - l) / 2;
```

```
  double left = area (f, l, m), right = area (f, m, r) ;
  if (fabs (left + right - a) <= 15 * eps) return left + right + (left + right - a) / 15.0;
  return solve (f, l, m, eps / 2, left) + solve (f, m, r, eps / 2, right); }
 double solve (double (*f) (double), double l, double r, double eps) {
  return solve (f, l, r, eps, area (f, l, r)); } };
```

3 Number theory

3.1 Fast power module

```
/* Fast power module : x^n */
int fpm (int x, int n, int mod) {
 int ans = 1, mul = x; while (n) {
  if (n & 1) ans = int (1ll * ans * mul % mod);
  mul = int (1ll * mul * mul % mod); n >>= 1; }
 return ans; }
```

3.2 Euclidean algorithm

```
/* Euclidean algorithm : solves for ax + by = gcd (a, b). */
void euclid (const long long &a, const long long &b, long long &x, long long &y) {
 if (b == 0) x = 1, y = 0;
 else euclid (b, a % b, y, x), y -= a / b * x; }

long long inverse (long long x, long long m) {
 long long a, b; euclid (x, m, a, b); return (a % m + m) % m; }
```

3.3 Discrete Fourier transform

```
/* Discrete Fourier transform : the nafarious you-know-what thing.
Usage : call init for the suggested array size, and solve for the transform. (use f!=0 for the inverse) */
template <int MAXN = 1000000>
struct dft {
  typedef std::complex <double> complex;
  complex e[2][MAXN];
  int init (int n) {
   int len = 1;
   for (; len <= 2 * n; len <<= 1);
   for (int i = 0; i < len; ++i) {
    e[0][i] = complex (cos (2 * PI * i / len), sin (2 * PI * i / len));
    e[1][i] = complex (cos (2 * PI * i / len), -sin (2 * PI * i / len)); }
   return len; }
  void solve (complex *a, int n, int f) {
   for (int i = 0, j = 0; i < n; ++i) {
    if (i > j) std::swap (a[i], a[j]);
    for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
   for (int i = 2; i <= n; i <<= 1)
    for (int j = 0; j < n; j += i)
     for (int k = 0; k < (i >> 1); ++k) {
      complex A = a[j + k];
      complex B = e[f][n / i * k] * a[j + k + (i >> 1)];
      a[j + k] = A + B;
      a[j + k + (i >> 1)] = A - B; }
   if (f == 1) {
    for (int i = 0; i < n; ++i) a[i] = complex (a[i].real () / n, a[i].imag ()); } } };
```

3.4 Number theoretic transform

```
/* Number theoretic transform : NTT for any module.
Usage : Perform NTT on 3 modules and call crt () to merge the result. */
template <int MAXN = 1000000>
struct ntt {
 int MOD[3] = {1045430273, 1051721729, 1053818881}, PRT[3] = {3, 6, 7};
 void solve (int *a, int n, int f, int mod, int prt) {
  for (int i = 0, j = 0; i < n; ++i) {
   if (i > j) std::swap (a[i], a[j]);
   for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
  for (int i = 2; i <= n; i <<= 1) {
   static int exp[MAXN]; exp[0] = 1;
   exp[1] = fpm (prt, (mod - 1) / i, mod);
   if (f == 1) exp[1] = fpm (exp[1], mod - 2, mod);
   for (int k = 2; k < (i >> 1); ++k) {
    exp[k] = int (1ll * exp[k - 1] * exp[1] % mod); }
   for (int j = 0; j < n; j += i) {
    for (int k = 0; k < (i >> 1); ++k) {
     int &pA = a[j + k], &pB = a[j + k + (i >> 1)];
     int A = pA, B = int (1ll * pB * exp[k] % mod);
     pA = (A + B) % mod;
     pB = (A - B + mod) % mod; } } }
  if (f == 1) {
   int rev = fpm (n, mod - 2, mod);
   for (int i = 0; i < n; ++i) a[i] = int (1ll * a[i] * rev % mod); } }
 int crt (int *a, int mod) {
  static int inv[3][3];
  for (int i = 0; i < 3; ++i) for (int j = 0; j < 3; ++j)
   inv[i][j] = (int) inverse (MOD[i], MOD[j]);
  static int x[3];
  for (int i = 0; i < 3; ++i) { x[i] = a[i];
   for (int j = 0; j < i; ++j) {
    int t = (x[i] - x[j] + MOD[i]) % MOD[i];
    if (t < 0) t += MOD[i];
    x[i] = int (1LL * t * inv[j][i] % MOD[i]); } }
  int sum = 1, ret = x[0] % mod;
  for (int i = 1; i < 3; ++i) {
   sum = int (1LL * sum * MOD[i - 1] % mod);
   ret += int (1LL * x[i] * sum % mod);
   if (ret >= mod) ret -= mod; }
  return ret; } };
```

3.5 Chinese remainder theorem

```
/* Chinese remainder theroem : finds positive integers x = out.first + k * out.second that satisfies x % in[i].second = in[i].first. */
struct crt {
 long long fix (const long long &a, const long long &b) { return (a % b + b) % b; }
 bool solve (const std::vector <std::pair <long long, long long>> &in, std::pair <long long, long long> &out) {
  out = std::make_pair (1LL, 1LL);
  for (int i = 0; i < (int) in.size (); ++i) {
   long long n, u;
   euclid (out.second, in[i].second, n, u);
   long long divisor = gcd (out.second, in[i].second);
   if ((in[i].first - out.first) % divisor) return false;
   n *= (in[i].first - out.first) / divisor;
   n = fix (n, in[i].second);
   out.first += out.second * n;
   out.second *= in[i].second / divisor;
   out.first = fix (out.first, out.second); }
  return true; } };
```

3.6 Linear Recurrence

```
/* Linear recurrence : finds the n-th element of a linear recurrence.
Usage : vector <int> - first n terms, vector <int> - transition function, calc (k) : the kth term mod MOD.
Example : In : {2, 1}, {2, 1} : a_1 = 2, a_2 = 1, a_n = 2a_{n-1} + a_{n-2}, Out : calc (3) = 5, calc (10007) = 959155122 (MOD 1E9+7) */
struct linear_rec {
 const int LOG = 30, MOD = 1E9 + 7; int n;
 std::vector <int> first, trans;
 std::vector <std::vector <int>> bin;
 std::vector <int> add (std::vector <int> &a, std::vector <int> &b) {
  std::vector <int> result(n * 2 + 1, 0);
  for (int i = 0; i <= n; ++i) for (int j = 0; j <= n; ++j)
   if ((result[i + j] += 1LL * a[i] * b[j] % MOD) >= MOD) result[i + j] -= MOD;
  for (int i = 2 * n; i > n; --i) {
   for (int j = 0; j < n; ++j)
    if ((result[i - 1 - j] += 1LL * result[i] * trans[j] % MOD) >= MOD) result[i - 1 - j] -= MOD;
   result[i] = 0; }
  result.erase(result.begin() + n + 1, result.end());
  return result; }
 linear_rec (const std::vector <int> &first, const std::vector <int> &trans) : first(first), trans(trans) {
  n = first.size(); std::vector <int> a(n + 1, 0); a[1] = 1; bin.push_back(a);
  for (int i = 1; i < LOG; ++i) bin.push_back(add(bin[i - 1], bin[i - 1])); }
 int solve (int k) {
  std::vector <int> a(n + 1, 0); a[0] = 1;
  for (int i = 0; i < LOG; ++i) if (k >> i & 1) a = add(a, bin[i]);
  int ret = 0;
  for (int i = 0; i < n; ++i) if ((ret += (long long) a[i + 1] * first[i] % MOD) >= MOD) ret -= MOD;
  return ret; } };
```

3.7 Baby step giant step algorithm

```
/* Baby step giant step algorithm : Solves a^x = b mod c in O(sqrt(c)). */
struct bsgs {
 int solve (int a, int b, int c) {
  std::unordered_map <int, int> bs;
  int m = (int) sqrt ((double) c) + 1, res = 1;
  for (int i = 0; i < m; ++i) {
   if (bs.find (res) == bs.end ()) bs[res] = i;
   res = int (1LL * res * a % c); }
  int mul = 1, inv = (int) inverse (a, c);
  for (int i = 0; i < m; ++i) mul = int (1LL * mul * inv % c);
  res = b % c;
  for (int i = 0; i < m; ++i) {
   if (bs.find (res) != bs.end ()) return i * m + bs[res];
   res = int (1LL * res * mul % c); }
  return -1; } };
```

3.8 Miller Rabin primality test

```
/* Miller Rabin : tests whether a certain integer is prime. */
struct miller_rabin {
 int BASE[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
 bool check (const long long &prime, const long long &base) {
  long long number = prime - 1;
  for (; ~number & 1; number >>= 1);
  long long result = llfpm (base, number, prime);
```

```
  for (; number != prime - 1 && result != 1 && result != prime - 1; number <<= 1)
   result = mul_mod (result, result, prime);
  return result == prime - 1 || (number & 1) == 1; }
 bool solve (const long long &number) {
  if (number < 2) return false;
  if (number < 4) return true;
  if (~number & 1) return false;
  for (int i = 0; i < 12 && BASE[i] < number; ++i) if (!check (number, BASE[i])) return false;
  return true; } };
```

3.9 Pollard's Rho algorithm

```
/* Pollard Rho : factorizes an integer. */
struct pollard_rho {
 miller_rabin is_prime;
 const long long threshold = 13E9;
 long long factorize (const long long &number, const long long &seed) {
  long long x = rand () % (number - 1) + 1, y = x;
  for (int head = 1, tail = 2; ; ) {
   x = mul_mod (x, x, number);
   x = (x + seed) % number;
   if (x == y) return number;
   long long answer = gcd (abs (x - y), number);
   if (answer > 1 && answer < number) return answer;
   if (++head == tail) { y = x; tail <<= 1; } } }
 void search (const long long &number, std::vector <long long> &divisor) {
  if (number > 1) {
   if (is_prime.solve (number)) divisor.push_back (number);
   else {
    long long factor = number;
    for (; factor >= number; factor = factorize (number, rand () % (number - 1) + 1));
    search (number / factor, divisor); search (factor, divisor); } } }
 std::vector <long long> solve (const long long &number) {
  std::vector <long long> ans;
  if (number > threshold) search (number, ans);
  else {
   long long rem = number;
   for (long long i = 2; i * i <= rem; ++i)
    while (!(rem % i)) { ans.push_back (i); rem /= i; }
   if (rem > 1) ans.push_back (rem); }
  return ans; } };
```

4 Geometry

```
#define cd const double &
const double EPS = 1E-8, PI = acos (-1);
int sgn (cd x) { return x < -EPS ? -1 : x > EPS; }
int cmp (cd x, cd y) { return sgn (x - y); }
double sqr (cd x) { return x * x; }
```

4.1 Point

```
#define cp const point &
struct point {
 double x, y;
 explicit point (cd x = 0, cd y = 0) : x (x), y (y) {}
 int dim () const { return sgn (y) == 0 ? sgn (x) < 0 : sgn (y) < 0; }
 point unit () const { double l = sqrt (x * x + y * y); return point (x / l, y / l); }
 //counter-clockwise
 point rot90 () const { return point (-y, x); }
 //clockwise
 point _rot90 () const { return point (y, -x); }
 point rot (cd t) const {
  double c = cos (t), s = sin (t);
  return point (x * c - y * s, x * s + y * c); } };
bool operator == (cp a, cp b) { return cmp (a.x, b.x) == 0 && cmp (a.y, b.y) == 0; }
bool operator != (cp a, cp b) { return cmp (a.x, b.x) != 0 || cmp (a.y, b.y) != 0; }
bool operator < (cp a, cp b) { return (cmp (a.x, b.x) == 0) ? cmp (a.y, b.y) < 0 : cmp (a.x, b.x) < 0; }
point operator - (cp a) { return point (-a.x, -a.y); }
point operator + (cp a, cp b) { return point (a.x + b.x, a.y + b.y); }
point operator - (cp a, cp b) { return point (a.x - b.x, a.y - b.y); }
point operator * (cp a, cd b) { return point (a.x * b, a.y * b); }
point operator / (cp a, cd b) { return point (a.x / b, a.y / b); }
double dot (cp a, cp b) { return a.x * b.x + a.y * b.y; }
double det (cp a, cp b) { return a.x * b.y - a.y * b.x; }
double dis2 (cp a, cp b = point ()) { return sqr (a.x - b.x) + sqr (a.y - b.y); }
double dis (cp a, cp b = point ()) { return sqrt (dis2 (a, b)); }
```

4.2 Line

```
#define cl const line &
struct line {
 point s, t;
 explicit line (cp s = point (), cp t = point ()) : s (s), t (t) {} };
bool point_on_segment (cp a, cl b) { return sgn (det (a - b.s, b.t - b.s)) == 0 && sgn (dot (b.s - a, b.t - a)) <= 0; }
bool two_side (cp a, cp b, cl c) { return sgn (det (a - c.s, c.t - c.s)) * sgn (det (b - c.s, c.t - c.s)) < 0; }
bool intersect_judgment (cl a, cl b) {
 if (point_on_segment (b.s, a) || point_on_segment (b.t, a)) return true;
 if (point_on_segment (a.s, b) || point_on_segment (a.t, b)) return true;
 return two_side (a.s, a.t, b) && two_side (b.s, b.t, a); }
point line_intersect (cl a, cl b) {
 double s1 = det (a.t - a.s, b.s - a.s), s2 = det (a.t - a.s, b.t - a.s);
 return (b.s * s2 - b.t * s1) / (s2 - s1); }
double point_to_line (cp a, cl b) { return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t); }
point project_to_line (cp a, cl b) { return b.s + (b.t - b.s) * (dot (a - b.s, b.t - b.s) / dis2 (b.t, b.s)); }
double point_to_segment (cp a, cl b) {
 if (sgn (dot (b.s - a, b.t - b.s) * dot (b.t - a, b.t - b.s)) <= 0) return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
 return std::min (dis (a, b.s), dis (a, b.t)); }
bool in_polygon (cp p, const std::vector <point> & po) {
 int n = (int) po.size (), counter = 0;
 for (int i = 0; i < n; ++i) {
  point a = po[i], b = po[(i + 1) % n];
  //Modify the next line if necessary.
  if (point_on_segment (p, line (a, b))) return true;
  int x = sgn (det (p - a, b - a)), y = sgn (a.y - p.y), z = sgn (b.y - p.y);
  if (x > 0 && y <= 0 && z > 0) counter++;
  if (x < 0 && z <= 0 && y > 0) counter--; }
 return counter != 0; }
double polygon_area (const std::vector <point> &a) {
 double ans = 0.0;
 for (int i = 0; i < (int) a.size (); ++i) ans += det (a[i], a[ (i + 1) % a.size ()]) / 2.0;
 return ans; }
```

4.3 Circle

```
#define cc const circle &
struct circle {
 point c; double r;
 explicit circle (point c = point (), double r = 0) : c (c), r (r) {} };
bool operator == (cc a, cc b) { return a.c == b.c && cmp (a.r, b.r) == 0; }
bool operator != (cc a, cc b) { return !(a == b); }
bool in_circle (cp a, cc b) { return cmp (dis (a, b.c), b.r) <= 0; }
circle make_circle (cp a, cp b) { return circle ((a + b) / 2, dis (a, b) / 2); }
circle make_circle (cp a, cp b, cp c) { point p = circumcenter (a, b, c); return circle (p, dis (p, a)); }
//In the order of the line vector.
std::vector <point> line_circle_intersect (cl a, cc b) {
 if (cmp (point_to_line (b.c, a), b.r) > 0) return std::vector <point> ();
 double x = sqrt (sqr (b.r) - sqr (point_to_line (b.c, a)));
 return std::vector <point> ({project_to_line (b.c, a) + (a.s - a.t).unit () * x, project_to_line (b.c, a) - (a.s - a.t).unit () * x}); }
double circle_intersect_area (cc a, cc b) {
 double d = dis (a.c, b.c);
 if (sgn (d - (a.r + b.r)) >= 0) return 0;
 if (sgn (d - abs(a.r - b.r)) <= 0) {
  double r = std::min (a.r, b.r); return r * r * PI; }
 double x = (d * d + a.r * a.r - b.r * b.r) / (2 * d), t1 = acos (min (1., max (-1., x / a.r))), t2 = acos (min (1., max (-1., (d - x) / b.r)));
 return a.r * a.r * t1 + b.r * b.r * t2 - d * a.r * sin (t1); }
//Counter-clockwise with respect of vector O_aO_b.
std::vector <point> circle_intersect (cc a, cc b) {
 if (a.c == b.c || cmp (dis (a.c, b.c), a.r + b.r) > 0 || cmp (dis (a.c, b.c), std::abs (a.r - b.r)) < 0) return std::vector <point> ();
 point r = (b.c - a.c).unit ();
 double d = dis (a.c, b.c);
 double x = ((sqr (a.r) - sqr (b.r)) / d + d) / 2;
 double h = sqrt (sqr (a.r) - sqr (x));
 if (sgn (h) == 0) return std::vector <point> ({a.c + r * x});
 return std::vector <point> ({a.c + r * x - r.rot90 () * h, a.c + r * x + r.rot90 () * h}); }
//Counter-clockwise with respect of point a.
std::vector <point> tangent (cp a, cc b) { circle p = make_circle (a, b.c); return circle_intersect (p, b); }
//Counter-clockwise with respect of point O_a.
std::vector <line> extangent (cc a, cc b) {
 std::vector <line> ret;
 if (cmp (dis (a.c, b.c), std::abs (a.r - b.r)) <= 0) return ret;
 if (sgn (a.r - b.r) == 0) {
  point dir = b.c - a.c; dir = (dir * a.r / dis (dir)).rot90 ();
  ret.push_back (line (a.c - dir, b.c - dir));
  ret.push_back (line (a.c + dir, b.c + dir));
 } else {
  point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
  std::vector pp = tangent (p, a), qq = tangent (p, b);
  if (pp.size () == 2 && qq.size () == 2) {
   if (cmp (a.r, b.r) < 0) std::swap (pp[0], pp[1]), std::swap (qq[0], qq[1]);
```

```
    ret.push_back (line (pp[0], qq[0]));
    ret.push_back (line (pp[1], qq[1])); } }
  return ret; }
//Counter-clockwise with respect of point O_a.
std::vector <line> intangent (cc c1, cc c2) {
  point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
  std::vector pp = tangent (p, a), qq = tangent (p, b);
  if (pp.size () == 2 && qq.size () == 2) {
    ret.push_back (line (pp[0], qq[0]));
    ret.push_back (line (pp[1], qq[1])); }
  return ret; }
```

4.4 Centers of a triangle

```
point incenter (cp a, cp b, cp c) {
  double p = dis (a, b) + dis (b, c) + dis (c, a);
  return (a * dis (b, c) + b * dis (c, a) + c * dis (a, b)) / p; }
point circumcenter (cp a, cp b, cp c) {
  point p = b - a, q = c - a, s (dot (p, p) / 2, dot (q, q) / 2);
  return a + point (det (s, point (p.y, q.y)), det (point (p.x, q.x), s)) / det (p, q); }
point orthocenter (cp a, cp b, cp c) { return a + b + c - circumcenter (a, b, c) * 2; }
```

4.5 Fermat point

```
/* Fermat point : finds a point P that minimizes |PA| + |PB| + |PC|. */
point fermat_point (cp a, cp b, cp c) {
  if (a == b) return a; if (b == c) return b; if (c == a) return c;
  double ab = dis (a, b), bc = dis (b, c), ca = dis (c, a);
  double cosa = dot (b - a, c - a) / ab / ca;
  double cosb = dot (a - b, c - b) / ab / bc;
  double cosc = dot (b - c, a - c) / ca / bc;
  double sq3 = PI / 3.0; point mid;
  if (sgn (cosa + 0.5) < 0) mid = a;
  else if (sgn (cosb + 0.5) < 0) mid = b;
  else if (sgn (cosc + 0.5) < 0) mid = c;
  else if (sgn (det (b - a, c - a)) < 0) mid = line_intersect (line (a, b + (c - b).rot (sq3)), line (b, c + (a - c).rot (sq3)));
  else mid = line_intersect (line (a, c + (b - c).rot (sq3)), line (c, b + (a - b).rot (sq3)));
  return mid; }
```

4.6 Convex hull

```
//Counter-clockwise, with minimum number of points.
bool turn_left (cp a, cp b, cp c) { return sgn (det (b - a, c - a)) >= 0; }
std::vector <point> convex_hull (std::vector <point> a) {
  int cnt = 0; std::sort (a.begin (), a.end ());
  std::vector <point> ret (a.size (), point ());
  for (int i = 0; i < (int) a.size (); ++i) {
    while (cnt > 1 && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) --cnt;
    ret[cnt++] = a[i]; }
  int fixed = cnt;
  for (int i = (int) a.size () - 1; i >= 0; --i) {
    while (cnt > fixed && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) --cnt;
    ret[cnt++] = a[i]; }
  return std::vector (ret.begin (), ret.begin () + cnt - 1); }
```

4.7 Half plane intersection

```
/* Online half plane intersection : complexity O(n) each operation. */
std::vector <point> cut (const std::vector<point> &c, line p) {
  std::vector <point> ret;
  if (c.empty ()) return ret;
  for (int i = 0; i < (int) c.size (); ++i) {
    int j = (i + 1) % (int) c.size ();
    if (turn_left (p.s, p.t, c[i])) ret.push_back (c[i]);
    if (two_side (c[i], c[j], p)) ret.push_back (line_intersect (p, line (c[i], c[j]))); }
  return ret; }
/* Offline half plane intersection : complexity O(n log n). */
bool turn_left (cl l, cp p) { return turn_left (l.s, l.t, p); }
int cmp (cp a, cp b) { return a.dim () != b.dim () ? (a.dim () < b.dim () ? -1 : 1) : -sgn (det (a, b)); }
std::vector <point> half_plane_intersect (std::vector <line> h) {
  typedef std::pair <point, line> polar;
  std::vector <polar> g; g.resize (h.size ());
  for (int i = 0; i < (int) h.size (); ++i) g[i] = std::make_pair (h[i].t - h[i].s, h[i]);
  sort (g.begin (), g.end (), [&] (const polar &a, const polar &b) {
    if (cmp (a.first, b.first) == 0) return sgn (det (a.second.t - a.second.s, b.second.t - a.second.s)) < 0;
    else return cmp (a.first, b.first) < 0; });
  h.resize (std::unique (g.begin (), g.end (), [] (const polar &a, const polar &b) { return cmp (a.first, b.first) == 0 }) - g.begin ());
  for (int i = 0; i < (int) h.size (); ++i) h[i] = g[i].second;
  int fore = 0, rear = -1; std::vector <line> ret (h.size (), line ());
  for (int i = 0; i < (int) h.size (); ++i) {
    while (fore < rear && !turn_left (h[i], line_intersect (ret[rear - 1], ret[rear]))) --rear;
    while (fore < rear && !turn_left (h[i], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
    ret.push_back[++rear] = h[i]; }
  while (rear - fore > 1 && !turn_left (ret[fore], line_intersect (ret[rear - 1], ret[rear]))) --rear;
  while (rear - fore > 1 && !turn_left (ret[rear], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
  if (rear - fore < 2) return std::vector <point> ();
  std::vector <point> ans; ans.resize (rear + 1);
  for (int i = 0; i < rear + 1; ++i) ans[i] = line_intersect (ret[i], ret[(i + 1) % (rear + 1)]);
  return ans; }
```

4.8 Minimum circle

```
circle minimum_circle (std::vector <point> p) {
  circle ret; std::random_shuffle (p.begin (), p.end ());
  for (int i = 0; i < (int) p.size (); ++i) if (!in_circle (p[i], ret)) {
    ret = circle (p[i], 0); for (int j = 0; j < i; ++j) if (!in_circle (p[j], ret)) {
      ret = make_circle (p[j], p[i]); for (int k = 0; k < j; ++k)
        if (!in_circle (p[k], ret)) ret = make_circle (p[i], p[j], p[k]); } }
  return ret; }
```

4.9 Intersection of a polygon and a circle

```
struct polygon_circle_intersect {
  double sector_area (cp a, cp b, const double &r) {
    double c = (2.0 * r * r - dis2 (a, b)) / (2.0 * r * r);
    return r * r * acos (c) / 2.0; }
  double area (cp a, cp b, const double &r) {
    double dA = dot (a, a), dB = dot (b, b), dC = point_to_segment (point (), line (a, b));
    if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0) return det (a, b) / 2.0;
    point tA = a.unit () * r, tB = b.unit () * r;
    if (sgn (dC - r) > 0) return sector_area (tA, tB, r);
    std::vector <point> ret = line_circle_intersect (line (a, b), circle (point (), r));
    if (sgn (dA - r * r) > 0 && sgn (dB - r * r) > 0)
      return sector_area (tA, ret[0], r) + det (ret[0], ret[1]) / 2.0 + sector_area (ret[1], tB, r);
    if (sgn (dA - r * r) > 0) return det (ret[0], b) / 2.0 + sector_area (tA, ret[0], r);
    else return det (a, ret[1]) / 2.0 + sector_area (ret[1], tB, r); }
  double solve (const std::vector <point> &p, cc c) {
    double ret = 0.0;
    for (int i = 0; i < (int) p.size (); ++i) {
      int s = sgn (det (p[i] - c.c, p[(i + 1) % p.size ()] - c.c));
      if (s > 0) ret += area (p[i] - c.c, p[(i + 1) % p.size ()] - c.c, c.r);
      else ret -= area (p[(i + 1) % p.size ()] - c.c, p[i] - c.c, c.r); }
    return std::abs (ret); } };
```

4.10 Union of circles

```
template <int MAXN = 500> struct union_circle {
  int C; circle c[MAXN]; double area[MAXN];
  struct event {
    point p; double ang; int delta;
    event (cp p = point (), double ang = 0, int delta = 0) : p(p), ang(ang), delta(delta) {}
    bool operator < (const event &a) { return ang < a.ang; }
  };
  void addevent(cc a, cc b, std::vector <event> &evt, int &cnt) {
    double d2 = dis2 (a.c, b.c), d_ratio = ((a.r - b.r) * (a.r + b.r) / d2 + 1) / 2,
      p_ratio = sqrt (std::max (0., -(d2 - sqr(a.r - b.r)) * (d2 - sqr(a.r + b.r)) / (d2 * d2 * 4)));
    point d = b.c - a.c, p = d.rot(PI / 2), q0 = a.c + d * d_ratio + p * p_ratio, q1 = a.c + d * d_ratio - p * p_ratio;
    double ang0 = atan2 ((q0 - a.c).y, (q0 - a.c).x), ang1 = atan2 ((q1 - a.c).x, (q1 - a.c).y);
    evt.emplace_back(q1, ang1, 1); evt.emplace_back(q0, ang0, -1); cnt += ang1 > ang0; }
  bool same(cc a, cc b) { return sgn (dis (a.c, b.c)) == 0 && sgn (a.r - b.r) == 0; }
  bool overlap(cc a, cc b) { return sgn (a.r - b.r - dis (a.c, b.c)) >= 0; }
  bool intersect(cc a, cc b) { return sgn (dis (a.c, b.c) - a.r - b.r) < 0; }
  void solve() {
    std::fill (area, area + C + 2, 0);
    for (int i = 0; i < C; ++i) {
      int cnt = 1; std::vector <event> evt;
      for (int j = 0; j < i; ++j) if (same (c[i], c[j])) ++cnt;
      for (int j = 0; j < C; ++j) if (j != i && !same (c[i], c[j]) && overlap (c[j], c[i])) ++cnt;
```

```
  for (int j = 0; j < C; ++j) if (j != i && !overlap (c[j], c[i]) && !overlap (c[i], c[j]) && intersect (c[i], c[j]))
   addevent (c[i], c[j], evt, cnt);
  if (evt.empty ()) area[cnt] += PI * c[i].r * c[i].r;
  else {
   std::sort (evt.begin (), evt.end ());
   evt.push_back (evt.front ());
   for (int j = 0; j + 1 < (int) evt.size (); ++j) {
    cnt += evt[j].delta; area[cnt] += det(evt[j].p, evt[j + 1].p) / 2;
    double ang = evt[j + 1].ang - evt[j].ang; if (ang < 0) ang += PI * 2;
    area[cnt] += ang * c[i].r * c[i].r / 2 - sin(ang) * c[i].r * c[i].r / 2; } } } } };
```

5 Graph

5.1 Hopcoft-Karp algorithm

```
/* Hopcoft-Karp algorithm : unweighted maximum matching for bipartition graphs with complexity O(m√n). */
template <int MAXN = 100000, int MAXM = 100000>
struct hopcoft_karp {
 using edge_list = std::vector <int> [MAXN];
 int mx[MAXN], my[MAXM], lv[MAXN];
 bool dfs (edge_list <MAXN, MAXM> &e, int x) {
  for (int y : e[x]) {
   int w = my[y];
   if (!~w || (lv[x] + 1 == lv[w] && dfs (e, w))) {
    mx[x] = y; my[y] = x; return true; } }
  lv[x] = -1; return false; }
 int solve (edge_list &e, int n, int m) {
  std::fill (mx, mx + n, -1); std::fill (my, my + m, -1);
  for (int ans = 0; ; ) {
   std::vector <int> q;
   for (int i = 0; i < n; ++i)
    if (mx[i] == -1) {
     lv[i] = 0; q.push_back (i);
    } else lv[i] = -1;
   for (int head = 0; head < (int) q.size(); ++head) {
    int x = q[head];
    for (int y : e[x]) { int w = my[y]; if (~w && lv[w] < 0) {
     lv[w] = lv[x] + 1; q.push_back (w); } } }
   int d = 0; for (int i = 0; i < n; ++i) if (!~mx[i] && dfs (e, i)) ++d;
   if (d == 0) return ans; else ans += d; } } };
```

5.2 Kuhn-Munkres algorithm

```
/* Kuhn Munkres algorithm : weighted maximum ming algorithm for bipartition graphs with complexity O(N^3).
Note : The graph is 1-based. */
template <int MAXN = 500>
struct kuhn_munkres {
 int n, w[MAXN][MAXN], lx[MAXN], ly[MAXN], m[MAXN], way[MAXN], sl[MAXN];
 bool u[MAXN];
 void hungary(int x) {
  m[0] = x;  int j0 = 0;
  std::fill (sl, sl + n + 1, INF); std::fill (u, u + n + 1, false);
  do {
   u[j0] = true; int i0 = m[j0], d = INF, j1 = 0;
   for (int j = 1; j <= n; ++j)
    if (u[j] == false) {
     int cur = -w[i0][j] - lx[i0] - ly[j];
     if (cur < sl[j]) { sl[j] = cur; way[j] = j0; }
     if (sl[j] < d) { d = sl[j]; j1 = j; } }
   for (int j = 0; j <= n; ++j) {
    if (u[j]) { lx[m[j]] += d; ly[j] -= d; }
    else sl[j] -= d; }
   j0 = j1; } while (m[j0] != 0);
  do {
   int j1 = way[j0]; m[j0] = m[j1]; j0 = j1;
  } while (j0); }
 int solve() {
  for (int i = 1; i <= n; ++i) m[i] = lx[i] = ly[i] = way[i] = 0;
  for (int i = 1; i <= n; ++i) hungary (i);
  int sum = 0; for (int i = 1; i <= n; ++i) sum += w[m[i]][i];
  return sum; } };
```

5.3 Maximum flow

```
/* Sparse graph maximum flow : isap.*/
template <int MAXN = 1000, int MAXM = 100000>
struct isap {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] = 0; begin[v] = size++; } };
 int pre[MAXN], d[MAXN], gap[MAXN], cur[MAXN];
 int solve (flow_edge_list &e, int n, int s, int t) {
  for (int i = 0; i < n; ++i) { pre[i] = d[i] = gap[i] = 0; cur[i] = e.begin[i]; }
  gap[0] = n; int u = pre[s] = s, v, maxflow = 0;
  while (d[s] < n) {
   v = n; for (int i = cur[u]; ~i; i = e.next[i])
    if (e.flow[i] && d[u] == d[e.dest[i]] + 1) {
     v = e.dest[i]; cur[u] = i; break; }
   if (v < n) {
    pre[v] = u; u = v;
    if (v == t) {
     int dflow = INF, p = t; u = s;
     while (p != s) { p = pre[p]; dflow = std::min (dflow, e.flow[cur[p]]); }
     maxflow += dflow; p = t;
     while (p != s) { p = pre[p]; e.flow[cur[p]] -= dflow; e.flow[cur[p] ^ 1] += dflow; } }
   } else {
    int mindist = n + 1;
    for (int i = e.begin[u]; ~i; i = e.next[i])
     if (e.flow[i] && mindist > d[e.dest[i]]) {
      mindist = d[e.dest[i]]; cur[u] = i; }
    if (!--gap[d[u]]) return maxflow;
    gap[d[u] = mindist + 1]++; u = pre[u]; } }
  return maxflow; } };
/* Dense graph maximum flow : dinic. */
template <int MAXN = 1000, int MAXM = 100000>
struct dinic {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, d[MAXN], w[MAXN], q[MAXN];
 int bfs (flow_edge_list &e) {
  std::fill (d, d + n, -1);
  int l, r; q[l = r = 0] = s, d[s] = 0;
  for (; l <= r; l ++)
   for (int k = e.begin[q[l]]; ~k; k = e.next[k])
    if (!~d[e.dest[k]] && e.flow[k] > 0) d[e.dest[k]] = d[q[l]] + 1, q[++r] = e.dest[k];
  return ~d[t] ? 1 : 0; }
 int dfs (flow_edge_list &e, int u, int ext) {
  if (u == t) return ext; int k = w[u], ret = 0;
  for (; ~k; k = e.next[k], w[u] = k) {
   if (ext == 0) break;
   if (d[e.dest[k]] == d[u] + 1 && e.flow[k] > 0) {
    int flow = dfs (e, e.dest[k], std::min (e.flow[k], ext));
    if (flow > 0) {
     e.flow[k] -= flow, e.flow[k ^ 1] += flow;
     ret += flow, ext -= flow; } } }
  if (!~k) d[u] = -1; return ret; }
 int solve (flow_edge_list &e, int n_, int s_, int t_) {
  int ans = 0; n = n_; s = s_; dinic::t = t_;
  while (bfs (e)) {
   for (int i = 0; i < n; ++i) w[i] = e.begin[i];
   ans += dfs (e, s, INF); }
  return ans; } };
```

5.4 Minimum cost flow

```
/* Sparse graph minimum cost flow : EK. */
template <int MAXN = 1000, int MAXM = 100000>
struct minimum_cost_flow {
 struct cost_flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  cost_flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int c, int f) {
   dest[size] = v; next[size] = begin[u]; cost[size] = c; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; cost[size] = -c; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, prev[MAXN], dist[MAXN], occur[MAXN];
 bool augment (cost_flow_edge_list &e) {
  std::vector <int> queue;
  std::fill (dist, dist + n, INF); std::fill (occur, occur + n, 0);
  dist[s] = 0; occur[s] = true; queue.push_back (s);
  for (int head = 0; head < (int)queue.size(); ++head) {
   int x = queue[head];
   for (int i = e.begin[x]; ~i; i = e.next[i]) {
    int y = e.dest[i];
    if (e.flow[i] && dist[y] > dist[x] + e.cost[i]) {
     dist[y] = dist[x] + e.cost[i]; prev[y] = i;
     if (!occur[y]) {
      occur[y] = true; queue.push_back (y); } } }
   occur[x] = false; }
  return dist[t] < INF; }
 std::pair <int, int> solve (cost_flow_edge_list &e, int n_, int s_, int t_) {
  n = n_; s = s_; t = t_; std::pair <int, int> ans = std::make_pair (0, 0);
  while (augment (e)) {
   int num = INF;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1]) {
    num = std::min (num, e.flow[prev[i]]); }
   ans.first += num;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1]) {
    e.flow[prev[i]] -= num; e.flow[prev[i] ^ 1] += num;
    ans.second += num * e.cost[prev[i]]; } }
  return ans; } };
/* Dense graph minimum cost flow : zkw. */
template <int MAXN = 1000, int MAXM = 100000>
struct zkw_flow {
```

```
struct cost_flow_edge_list {
 int size, begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM], flow[MAXM];
 void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
 cost_flow_edge_list (int n = MAXN) { clear (n); }
 void add_edge (int u, int v, int c, int f) {
  dest[size] = v; next[size] = begin[u]; cost[size] = c; flow[size] = f; begin[u] = size++;
  dest[size] = u; next[size] = begin[v]; cost[size] = -c; flow[size] = 0; begin[v] = size++; } };
int n, s, t, tf, tc, dis[MAXN], slack[MAXN], visit[MAXN];
int modlable() {
 int delta = INF;
 for (int i = 0; i < n; i++) {
  if (!visit[i] && slack[i] < delta) delta = slack[i];
  slack[i] = INF; }
 if (delta == INF) return 1;
 for (int i = 0; i < n; i++) if (visit[i]) dis[i] += delta;
 return 0; }
int dfs (cost_flow_edge_list &e, int x, int flow) {
 if (x == t) { tf += flow; tc += flow * (dis[s] - dis[t]); return flow; }
 visit[x] = 1; int left = flow;
 for (int i = e.begin[x]; ~i; i = e.next[i])
  if (e.flow[i] > 0 && !visit[e.dest[i]]) {
   int y = e.dest[i];
   if (dis[y] + e.cost[i] == dis[x]) {
    int delta = dfs (e, y, std::min (left, e.flow[i]));
    e.flow[i] -= delta; e.flow[i ^ 1] += delta; left -= delta;
    if (!left) { visit[x] = false; return flow; }
   } else
    slack[y] = std::min (slack[y], dis[y] + e.cost[i] - dis[x]); }
 return flow - left; }
std::pair <int, int> solve (cost_flow_edge_list &e, int n_, int s_, int t_) {
 n = n_; s = s_; t = t_; tf = tc = 0;
 std::fill (dis + 1, dis + t + 1, 0);
 do { do {
  std::fill (visit + 1, visit + t + 1, 0);
 } while (dfs (e, s, INF)); } while (!modlable ());
 return std::make_pair (tf, tc);
} };
```