```
HASHING PRIMES: {1610612741 - 1073676287 - 805306457 - 402653189}
```

Suffix Array

```
// String index from 0
// Usage: string s;  (s[i] > 0)
// sa.LCP[i] = max common prefix suffix of sa.SA[i-1] and sa.SA[i]
struct SuffixArray {
  string a; int N, m;
  vector<int> SA, LCP, x, y, w, c;
  SuffixArray(string _a, int m = 256) :
      a(" " + _a), N(a.length()), m(m), SA(N), LCP(N), x(N), y(N), w(
          max(m, N)), c(N) {
    a[0] = 0; DA(); kasaiLCP();
#define REF(X) { rotate(X.begin(), X.begin()+1, X.end()); X.pop_back(); }
    REF(SA); REF(LCP);
    a = a.substr(1, a.size());
    for (int i = 0; i < (int) SA.size(); ++i)
      --SA[i];
#undef REF
  }
  inline bool cmp(const int a, const int b, const int l) {
    return (y[a] == y[b] && y[a + l] == y[b + l]);
  }
  void Sort() {
    for (int i = 0; i < m; ++i) w[i] = 0;
    for (int i = 0; i < N; ++i)  ++w[x[y[i]]];
    for (int i = 0; i < m - 1; ++i)  w[i + 1] += w[i];
    for (int i = N - 1; i >= 0; --i) SA[--w[x[y[i]]]] = y[i];
  }
  void DA() {
    for (int i = 0; i < N; ++i)  x[i] = a[i], y[i] = i;
    Sort();
    for (int i, j = 1, p = 1; p < N; j <<= 1, m = p) {
      for (p = 0, i = N - j; i < N; i++) y[p++] = i;
      for (int k = 0; k < N; ++k)
        if (SA[k] >= j)  y[p++] = SA[k] - j;
      Sort();
      for (swap(x, y), p = 1, x[SA[0]] = 0, i = 1; i < N; ++i)
        x[SA[i]] = cmp(SA[i - 1], SA[i], j) ? p - 1 : p++;
    }
  }
  void kasaiLCP() {
    for (int i = 0; i < N; i++) c[SA[i]] = i;
    for (int i = 0, j, k = 0; i < N; LCP[c[i++]] = k)
      if (c[i] > 0)
        for (k ? k-- : 0, j = SA[c[i] - 1]; a[i + k] == a[j + k]; k++);
      else k = 0;
  } };
```

Suffix Automaton

```
const int chars = 27;
const int minchar = 'a';
struct SuffixAutomaton {
  vector<vector<int> > edges; // edges[i] : the labeled edges from node i
  // link[i] : the parent of i
  // length[i] : the length of the longest string in the ith class
  // terminal nodes
  vector<int> length, terminals, link;
  int last;
  // the index of the equivalence class of the whole string
  SuffixAutomaton() {
    last = 0;
  }
  SuffixAutomaton(string & s) {
    // add the initial node
    edges.push_back(vector<int>(chars, -1)); last = 0;
    link.push_back(-1);  length.push_back(0);

    for (int i = 0; i < (int) s.size(); i++) {
      // construct r
      edges.push_back(vector<int>(chars, -1));
      length.push_back(i + 1); link.push_back(0);
      int r = edges.size() - 1, p = last;
      while (p >= 0 && edges[p][s[i] - minchar] == -1) {
        edges[p][s[i] - minchar] = r;
        p = link[p];
      }
      if (p == -1) {
        link[r] = 0;
      } else {
        int q = edges[p][s[i] - minchar];
        if (length[p] + 1 == length[q]) link[r] = q;
        else {
          edges.push_back(edges[q]); length.push_back(length[p] + 1);
          link.push_back(link[q]); int qq = edges.size() - 1;
          link[q] = qq; link[r] = qq;
          while (p >= 0 && edges[p][s[i] - minchar] == q)
    edges[p][s[i] - minchar] = qq, p = link[p];
        }
      }
      last = r;
    }
    int p = last;
    while (p > 0) terminals.push_back(p), p = link[p];
    sort(terminals.begin(), terminals.end()); } };
```

Z function

```
void z_function(char *s, int *z) {
```

```
  int n = strlen(s);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r)  z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
}
```

KMP

```
void computeLPSArray(char *pat, int M, int *lps) {
  int len = 0, i = 1; lps[0] = 0;
  while (i < M)
    if (pat[i] == pat[len]) lps[i++] = ++len;
    else if (len != 0) len = lps[len - 1];
    else lps[i++] = 0;
}
void KMPSearch(char *pat, char *txt) {
  int N = strlen(txt), M = strlen(pat), i = 0, j = 0;
  int *lps = (int *) malloc(sizeof(int) * M);
  computeLPSArray(pat, M, lps);
  while (i < N) {
    if (pat[j] == txt[i]) i++, j++;
    if (j == M) j = lps[j - 1];
    else if (i < N && pat[j] != txt[i]) {
      if (j != 0) j = lps[j - 1];
      else i = i + 1;
    }
  }
}
```

Manacher

```
/*
 * manacher(s) = LPS array
 * s = "abcdef" put separator "| a | b | c | d | e | f |"
 * LPS[i] will tell u length of longest palind centered at i
 */
vector<int> manacher(string s) {
  int n = s.size();
  if (n == 0)
    return vector<int>();
  n = 2 * n + 1; //Position count
  vector<int> L(n, 0);
  L[1] = 1;
  int C = 1, R = 2, iMirror;
  int diff = -1;
  for (int i = 2; i < n; i++) {
    iMirror = 2 * C - i, L[i] = 0, diff = R - i;
    if (diff > 0) L[i] = min(L[iMirror], diff);
    while (((i + L[i]) < n && (i - L[i]) > 0)
        && (((i + L[i] + 1) % 2 == 0)
            || (s[(i + L[i] + 1) / 2] == s[(i - L[i] - 1) / 2])))
      L[i]++;
    if (i + L[i] > R) C = i, R = i + L[i];
  }
  return L;
}
```

Aho Corasik

```
/*
 * Usage: read all the small patterns, put them in vector W
 *     AC_FSM fsm;
 fsm.construct_automaton(W);
 read big stirng S
 fsm.aho_corasick(S, W, matches);
 matches is a vvi containing matches for each pattern in W
 */
struct AC_FSM {
#define ALPHABET_SIZE 256
  struct Node {
    int child[ALPHABET_SIZE], failure = 0, match_parent = -1;
    vector<int> match;
    Node() {
      for (int i = 0; i < ALPHABET_SIZE; ++i)child[i] = -1;
    }
  };
  vector<Node> a;
  AC_FSM() { a.push_back(Node());  }
  void construct_automaton(vector<string> &words) {
    for (int w = 0, n = 0; w < words.size(); ++w, n = 0) {
      for (int i = 0; i < words[w].size(); ++i) {
        if (a[n].child[words[w][i]] == -1) {
          a[n].child[words[w][i]] = a.size();
          a.push_back(Node());
        }
        n = a[n].child[words[w][i]];
      }
      a[n].match.push_back(w);
    }
    queue<int> q;
    for (int k = 0; k < ALPHABET_SIZE; ++k) {
      if (a[0].child[k] == -1) a[0].child[k] = 0;
      else if (a[0].child[k] > 0) {
        a[a[0].child[k]].failure = 0;
        q.push(a[0].child[k]);
      }
    }
    while (!q.empty()) {
      int r = q.front(); q.pop();
      for (int k = 0, arck; k < ALPHABET_SIZE; ++k) {
```

```
        if ((arck = a[r].child[k]) != -1) {
          q.push(arck); int v = a[r].failure;
          while (a[v].child[k] == -1)  v = a[v].failure;
          a[arck].failure = a[v].child[k];
          a[arck].match_parent = a[v].child[k];
          while (a[arck].match_parent != -1 &&
a[a[arck].match_parent].match.empty())
            a[arck].match_parent = a[a[arck].match_parent].match_parent;
        }
      }
    }
  }
  void aho_corasick(string &sentence, vector<string> &words,
      vector<vector<int> > &matches) {
    matches.assign(words.size(), vector<int>());
    int state = 0, ss = 0;
    for (int i = 0; i< sentence.length(); ++i, ss = state) {
      while (a[ss].child[sentence[i]] == -1)
        ss = a[ss].failure;
      state = a[state].child[sentence[i]] = a[ss].child[sentence[i]];
      for (ss = state; ss != -1; ss = a[ss].match_parent)
        for (int w : a[ss].match)
          matches[w].push_back(i + 1 - words[w].length());
    }}};
```

FFT

```
typedef complex<double> base;
double PI = acos(-1);
base W[20][1 << 20][2];
// call it in the beginning of the problem with n = (powerof2 > n) * 2
void calcW(int n) {
  for (int invert = 0; invert < 2; invert++)
    for (int len = 2, q = 0; len <= n; len <<= 1, q++) {
      double ang = 2 * PI / len * (invert ? -1 : 1);
      W[q][0][invert] = base(1, 0);
      base wlen(cos(ang), sin(ang));
      for (int j = 1; j < len / 2; j++)
        W[q][j][invert] = W[q][j - 1][invert] * wlen;
    }
}
void fft(vector<base> & a, int invert) {
  int n = int(a.size());
  for (int i = 1, j = 0; i < n; ++i) {
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1) j -= bit;
    j += bit;
    if (i < j) swap(a[i], a[j]);
  }
  for (int len = 2, q = 0; len <= n; len <<= 1, q++)
    for (int i = 0; i < n; i += len) {
      for (int j = 0; j < len / 2; ++j) {
        base u = a[i + j], v = a[i + j + len / 2] * W[q][j][invert];
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
      }
    }
  if (invert)  for (int i = 0; i < n; ++i)  a[i] /= n;
}
void multiply(const vector<long long> &a, const vector<long long> &b,
    vector<long long> &res) {
  vector<base> fa(a.begin(),a.end()),fb(b.begin(), b.end());
  size_t n = 1;
  while (n < max(a.size(), b.size()))  n <<= 1; n <<= 1;
  fa.resize(n), fb.resize(n);
  fft(fa, false), fft(fb, false);
  for (size_t i = 0; i < n; ++i) fa[i] *= fb[i];
  fft(fa, true); res.resize(n);
  for (size_t i = 0; i < n; ++i)
    res[i] = fa[i].real() + 0.5;
}
```

Implicit Lazy tree

```
// for update -> update(updLeft, updRight, updVal, 1, leftMostIndex,
rightMostIndex)
// for get -> get(queryLeft, queryRight, 1, leftMostIndex,
rightMostIndex)
struct Tree {
  vector<long long> tree, lazy;
  vector<int> Lchild, Rchild;
  Tree() { pushNode(); pushNode(); }
  void pushNode() {
    tree.push_back(0); lazy.push_back(0);
    Lchild.push_back(-1); Rchild.push_back(-1);
  }
  void createChildren(int node) {
    if (Lchild[node] == -1)
      Lchild[node] = tree.size(), pushNode();
    if (Rchild[node] == -1)
      Rchild[node] = tree.size(), pushNode();
  }
  void propagate(int node, int left, int right) {
    tree[node] += lazy[node] * (right - left + 1);
    if (left != right) {
      lazy[Lchild[node]] += lazy[node];
      lazy[Rchild[node]] += lazy[node];
    }
    lazy[node] = 0;
  }
```

```
  void update(int x, int y, int val, int node, int left, int right) {
    if (y < x)
      return;
    if (left != right) createChildren(node);
    // (x, y) are the query range (left, right) are for the current node
    propagate(node, left, right);
    if (y < left || right < x) return;
    if (x <= left && right <= y) {
      lazy[node] = val; propagate(node, left, right);
      return;
    }
    int mid = (left + right) >> 1;
    update(x, y, val, Lchild[node], left, mid);
    update(x, y, val, Rchild[node], mid + 1, right);
    tree[node] = tree[Lchild[node]] + tree[Rchild[node]];
  }
  long long get(int x, int y, int node, int left, int right) {
    if (y < x) return 0;
    if (left != right) createChildren(node);
    propagate(node, left, right);
    if (y < left || right < x) return 0;
    if (x <= left && right <= y) return tree[node];
    int mid = (left + right) >> 1;
    return get(x, y, Lchild[node], left, mid)
        + get(x, y, Rchild[node], mid + 1, right);
  }
};
```

MO Algorithm

```
#define BLOCK 500
bool cmp(pair<int, int> &x, pair<int, int> &y) {
  if (x.first / BLOCK != y.first / BLOCK)
    return x.first / BLOCK < y.first / BLOCK;
  return x.second < y.second;
}
void queryProcessor(int newL, int newR, int currentL, int currentR) {
  while (currentL < newL) remove(currentL++);
  while (currentL > newL)  add(--currentL);
  while (currentR <= newR) add(currentR++);
  while (currentR > newR + 1) remove(--currentR);
}
```

Ordinary Treap

```
struct item {
  int key, prior;
  item * l, *r;
  item() {
    l = r = NULL;
    key = prior = 0;
  }
  item(int key, int prior) :
      key(key), prior(prior), l(NULL), r(NULL) {
  }
};
typedef item * pitem;
void split(pitem t, int key, pitem & l, pitem & r) {
  if (!t)  l = r = NULL;
  else if (key < t->key) split(t->l, key, l, t->l), r = t;
  else split(t->r, key, t->r, r), l = t;
}
void insert(pitem & t, pitem it) {
  if (!t) t = it;
  else if (it->prior > t->prior)
    split(t, it->key, it->l, it->r), t = it;
  else
    insert(it->key < t->key ? t->l : t->r, it);
}
void merge(pitem & t, pitem l, pitem r) {
  if (!l || !r) t = l ? l : r;
  else if (l->prior > r->prior) merge(l->r, l->r, r), t = l;
  else merge(r->l, l, r->l), t = r;
}
void erase(pitem & t, int key) {
  if (t->key == key) merge(t, t->l, t->r);
  else erase(key < t->key ? t->l : t->r, key);
}
```

Implicit Treap

```
enum {
  LEFT, RIGHT, PARENTLESS
};
typedef struct item * pitem;
struct item {
  int prior, value, cnt, rev, type;
  pitem l, r, p;
  item() {
    prior = value = cnt = rev = 0;
    type = PARENTLESS;
    l = r = p = NULL;
  }
  item(int val) {
    prior = cnt = rev = 0;
    type = PARENTLESS; value = val;
    l = r = p = NULL;
  }
};
int cnt(pitem it) { return it ? it->cnt : 0; }
```

```
void upd_cnt(pitem it) {
  if (it) it->cnt = cnt(it->l) + cnt(it->r) + 1;
}
void noParent(pitem it) {
  if (it) it->p = NULL, it->type = PARENTLESS;
}
void zabbat(pitem it) {
  if (it && it->r) it->r->p = it, it->r->type = RIGHT;
  if (it && it->l) it->l->p = it, it->l->type = LEFT;
}
void push(pitem it) {
  if (it && it->rev) {
    it->rev = false; swap(it->l, it->r);
    if (it->l) it->l->rev ^= true; if (it->r) it->r->rev ^= true;
  }
  zabbat(it);
}
void merge(pitem & t, pitem l, pitem r) {
  push(l); push(r);
  noParent(l); noParent(r);
  if (!l || !r) t = l ? l : r;
  else if (l->prior > r->prior) merge(l->r, l->r, r), t = l;
  else merge(r->l, l, r->l), t = r;
  zabbat(t); upd_cnt(t);
}
void split(pitem t, pitem & l, pitem & r, int key, int add = 0) {
  if (!t) return void(l = r = 0); push(t);
  int cur_key = add + cnt(t->l);
  if (key <= cur_key) split(t->l, l, t->l, key, add), r = t;
  else split(t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
  zabbat(l); zabbat(r); upd_cnt(t);
}
void reverse(pitem t, int l, int r) {
  pitem t1, t2, t3; split(t, t1, t2, l);
  split(t2, t2, t3, r - l + 1); t2->rev ^= true;
  merge(t, t1, t2); merge(t, t, t3);
}
void output(pitem t) {
  if (!t) return;
  push(t); output(t->l);
  printf("%d ", t->value); output(t->r);
}
// adding new item to the treap:
//pitem p = new item(); p->value = a[i];
//p->prior = rand(); M[i] = p;
//merge(treap, treap, p);
```

Heavy Light Decomposition

```
// init: chainHead(-1), sz(size of all subtrees), g.clear()
const int MAXN = 1e5 + 5;
vector<int> g[MAXN];
int curChainId = 0, chainHead[MAXN], chainPos[MAXN], chainId[MAXN],
    chainSize[MAXN], sz[MAXN], absPos[MAXN], accSz = 0, revPos[MAXN];
void hld(int u, int par) {
  if (chainHead[curChainId] == -1) chainHead[curChainId] = u;
  chainId[u] = curChainId; chainPos[u] = chainSize[curChainId]++;
  revPos[accSz] = u; absPos[u] = accSz++;
  int hvyChild = -1;
  for (int i = 0; i < g[u].size(); i++)
    if (g[u][i] != par && (hvyChild == -1 || sz[g[u][i]] > sz[hvyChild]))
      hvyChild = g[u][i];
  if (hvyChild != -1) hld(hvyChild, u);
  for (int i = 0; i < g[u].size(); i++)
    if (g[u][i] != hvyChild && g[u][i] != par)
      curChainId++, hld(g[u][i], u);
}
```

LCA With Sparse Table

```
// init: level(depth of all nodes in the tree)
void buildLCA(vector<int>&par, int sparse[][21]) {
  memset(sparse, -1, sizeof sparse);
  for (int i = 0; i < par.size(); i++) sparse[i][0] = par[i];
  for (int j = 1; j < 21; j++) {
    for (int i = 0; i < par.size(); i++)
      if (sparse[i][j - 1] != -1)
        sparse[i][j] = sparse[sparse[i][j - 1]][j - 1];
  }
}
int pthParent(int u, int p, int sparse[][21]) {
  for (int j = 20; j >= 0 && u != -1; j--)
    if ((1 << j) <= p) p -= (1 << j), u = sparse[u][j];
  return u;
}
int getLCA(int u, int v, int sparse[][21]) {
  if (level[u] < level[v]) swap(u, v);
  u = pthParent(u, level[u] - level[v], sparse);
  if (u == v) return u;
  for (int j = 20; j >= 0; j--)
    if (sparse[u][j] != sparse[v][j])
      u = sparse[u][j], v = sparse[v][j];
  return sparse[u][0];
}
```

DSU on tree

```
// init: sz(size of all subtrees), hvy(0)
// just edit the add() and fill result in dfs()
int sz[MAXN], cnt[MAXN], col[MAXN];
bool hvy[MAXN];
```

```cpp
void add(int u, int par, int x) {
  cnt[col[u]] += x;
  for (int i = 0; i < g[u].size(); i++)
    if (g[u][i] != par && !hvy[g[u][i]])
      add(g[u][i], u, x);
}
void dfs(int u, int par, bool keep) {
  int hvyChild = -1;
  for (int i = 0; i < g[u].size(); i++)
    if (g[u][i] != par && (hvyChild == -1 || sz[g[u][i]] > sz[hvyChild]))
      hvyChild = g[u][i];
  for (int i = 0; i < g[u].size(); i++)
    if (g[u][i] != par && g[u][i] != hvyChild)
      dfs(g[u][i], u, 0);
  if (hvyChild != -1)  dfs(hvyChild, u, 1), hvy[hvyChild] = 1;
  add(u, par, 1);
  /*** HERE answer queries for subtree u HERE ***/
  /*** now cnt[c] is no. of vertexes in subtree u with col c ***/
  if (hvyChild != -1) hvy[hvyChild] = 0;
  if (!keep) add(u, par, -1);
}
```

Centroid Decomposition

```cpp
struct CentroidDecomposition {
  int sz[MAXN]; bool done[MAXN];
  vector<pair<int, int> > saved[MAXN];
  vector<int> ctree[MAXN];
  vector<int> g[MAXN];

  // init: done(0), get graph in constructor
  CentroidDecomposition() {}
  int updateSZ(int u, int par) {
    sz[u] = 1;
    for (int i = 0; i < g[u].size(); i++)
      if (!done[g[u][i]] && g[u][i] != par)
        sz[u] += updateSZ(g[u][i], u);
    return sz[u];
  }
  int getCentroid(int u, int par, int n) {
    int hvyChild = -1;
    for (int i = 0; i < g[u].size(); i++)
      if (!done[g[u][i]] && g[u][i] != par
    && (hvyChild == -1 || sz[g[u][i]] > sz[hvyChild]))
        hvyChild = g[u][i];
    if (sz[hvyChild] <= n / 2) return u;
    return getCentroid(hvyChild, u, n);
  }
  void save(int u, int par, int cen, int x) {
    saved[cen].push_back(make_pair(u, x));
    for (int i = 0; i < g[u].size(); i++)
      if (!done[g[u][i]] && g[u][i] != par)
        save(g[u][i], u, cen, x + 1);
  }
  void decompose(int u, int par = -1) {
    int cen = getCentroid(u, par, updateSZ(u, par));
    done[cen] = 1; save(cen, -1, cen, 0);
    if (par != -1) ctree[par].push_back(cen);
    for (int i = 0; i < g[cen].size(); i++)
      if (!done[g[cen][i]]) decompose(g[cen][i], cen);
  }
};
```

MO On Trees

Flatten the tree **using** dfs in/out time (increment time **for** both instances),
for path related queries (u, v) where start(u) <= st(v) **and** lca(u, v) = p,
answer is **for** the nodes that appears odd number of times in the range R,
if(l == u) R = [start(u),start(v)], **else** R = [end(u),start(v)] + [start(p),start(p)]

Simplex

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

Adjacency matrix implementation of Stoer Wagner min cut

```
// Running time: O(|V|^3)
// INPUT: graph, constructed using AddEdge()
// OUTPUT: (min cut value, nodes in half of min cut)
typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;
pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut; int best_weight = -1;
  for (int phase = N - 1; phase >= 0; phase--) {
    VI w = weights[0], added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last; last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase - 1) {
        for (int j = 0; j < N; j++)
          weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++)
          weights[j][prev] = weights[prev][j];
        used[last] = true; cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight)
          best_cut = cut,  best_weight = w[last];
      } else
          for (int j = 0; j < N; j++)
            w[j] += weights[last][j], added[last] = true;
    }
  }
  return make_pair(best_weight, best_cut); }
```

Divide and conquer optimization

```
// Original Recurrence
//   dp[i][j] = min(dp[i-1][k] + C[k][j]) for k < j
// Sufficient condition:
```

```
//   A[i][j] <= A[i][j+1]
//   where A[i][j] = smallest k that gives optimal answer
// How to use:
//   1-indexed
//   set dp[0][i] = INF
//   build(i,1,N,0,N); for each i
const int mxN = 4011;
const int inf = 1000111000;
int n, k, cost[mxN][mxN], dp[mxN][mxN];
inline int getCost(int i, int j) {
  return cost[j][j]-cost[j][i - 1]-cost[i - 1][j] + cost[i - 1][i - 1];
}
void build(int i, int L, int R, int optL, int optR) {
  if (L > R) return;
  int mid = (L + R) >> 1, savek = optL; dp[i][mid] = inf;
  for (int k = optL; k <= min(mid - 1, optR); k++) {
    int cur = dp[i - 1][k] + getCost(k + 1, mid);
    if (cur < dp[i][mid]) dp[i][mid] = cur, savek = k;
  }
  build(i, L, mid - 1, optL, savek);
  build(i, mid + 1, R, savek, optR);
}
// Dynamic Convex hull trick DP
const long long is_query = -(1LL << 62);
struct Line {
  long long 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;
    long long x = rhs.m;
    return b - s->b < (s->m - m) * x;
  }};
struct HullDynamic: public multiset<Line> {
  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(long long m, long long 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));
  }
  long long eval(long long x) {
    auto l = *lower_bound((Line ) { x, is_query });
    return l.m * x + l.b;
  }
};
```

Knuth DP optimization

```
// DP[i][j] = min_k {DP[i][k] + DP[k][j]} + C[i][j]
const int mxN = 2100;
long long dp[mxN][mxN];
int opt[mxN][mxN];
long long C[mxN][mxN];
void build(int N) {
  for (int i = 0; i < N; i++) dp[i][i] = C[i][i], opt[i][i] = i;
  for (int len = 1; len < N; len++) {
    for (int i = 0; i + len < N; i++) {
      int j = i + len; dp[i][j] = LLONG_MAX - C[i][j];
      for (int k = opt[i][j - 1]; k <= opt[i + 1][j]; k++) {
        if (dp[i][k] + dp[k][j] <= dp[i][j])
          dp[i][j] = dp[i][k] + dp[k][j], opt[i][j] = k;
      }
      dp[i][j] += C[i][j]; }}}
```

DP optimizations ideas

```
// * Visit all submasks in only NcK
for(int i = mask; i > 0; i = (i-1) & mask)
// * Another way to do cumulative operation over all submasks
for(int i = 0; i<(1<<N); ++i)
F[i] = BASEVALUE[i];
for(int i = 0;i < N; ++i)
for(int mask = 0; mask < (1<<N); ++mask) {
  if(mask & (1<<i))
  F[mask] = cumulative(F[mask],F[mask^(1<<i)]);
}
//
// * Some times We don't need to visit all the states. Examples:
//   - When one parameter is the sum of the arithematic progression of
another parameter
//   - When one parameter is exponential in another parameter.
//   - When inner iteration must be validated and there are small number
of possible transistions. Precompute those possible transistions.
//
// * Connected componenets DP:
```

```
//   Example problem:
//Number ofpermutations 2 ? 1 3 with max cost of the consecutive elements
//   Condition: cost(i,j) = cost(i,j-1) + cost(j-1,j)
//We assumethat not filled ? are the maximum number we are at and proceed
//kl = 1 if there is a segment connected to the left border, 0 otherwise
//kr = 1 if there is a segment connected to the right border, 0 otherwise
//   k is the number of filled segments in the middle
if (at > 0) {
  // add the penalty from the last element
  nxtSum += (kl+kr+2*k)*cost(arr[at]-arr[at-1]);
}
res += dp(at+1, nxtSum, 1, k, kr); // connect to left segment
res += dp(at+1, nxtSum, 1, k-1, kr)*k;// connect to left segment, and
join to some middle segment
res += dp(at+1, nxtSum, kl, k, 1);// connect to right segment
res += dp(at+1, nxtSum, kl, k-1, 1)*k;// connect to right segment, and
join to some middle segment
res += dp(at+1, nxtSum, kl, k+1, kr);// new segment
res += dp(at+1, nxtSum, kl, k, kr)*k*2;// connect to some middle segment
res += dp(at+1, nxtSum, kl, k-1, kr)*k*(k-1);// join two middle segments
//
// * 2 X 1 + trick
// Find a way to compute dp[2*i][STATE] from dp[i][STATE] then use map
```

MinimumVertexCover

```
struct MinimumVertexCover {
  int n, id;
  vector<vector<int> > g;
  vector<int> color, m[2], seen, comp[2];
  MinimumVertexCover(int n, vector<vector<int> > g) {
    this->n = n, this->g = g;
    color = m[0] = m[1] = vector<int>(n, -1);
    seen = vector<int>(n, 0);
    makeBipartite();
  }
  void dfsBipartite(int node, int col) {
    if (color[node] != -1) {
      assert(color[node] == col); /* MSH BIPARTITE YA BASHMOHANDES */
      return;
    }
    color[node] = col, comp[col].push_back(node);
    for (int i = 0; i < int(g[node].size()); i++)
      dfsBipartite(g[node][i], 1 - col);
  }
  void makeBipartite() {
    for (int i = 0; i < n; i++) if (color[i] == -1) dfsBipartite(i, 0);
  }
  // match a node
  bool dfs(int node) {
    for (int i = 0; i < g[node].size(); i++) {
      int child = g[node][i];
      if (m[1][child] == -1) {
        m[0][node] = child; m[1][child] = node;
        return true;
      }
      if (seen[child] == id) continue;
      seen[child] = id; int enemy = m[1][child];
      m[0][node] = child;  m[1][child] = node;  m[0][enemy] = -1;
      if (dfs(enemy))  return true;
      m[0][node] = -1; m[1][child] = enemy; m[0][enemy] = child;
    }
    return false;
  }
  void makeMatching() {
    for (int i = 0; i < int(comp[0].size()); i++) {
      id++;
      if (m[0][i] == -1) dfs(comp[0][i]);
    }
  }
  void recurse(int node, int x, vector<int> &minCover, vector<int>
&maxIndep, vector<int> &done) {
    if (m[x][node] != -1) return;
    if (done[node]) return;
    maxIndep.push_back(node); done[node] = 1;
    for (int i = 0; i < int(g[node].size()); i++) {
      int child = g[node][i], newnode = m[x ^ 1][child];
      if (done[child]) continue;
      done[child] = 2; minCover.push_back(child);
      m[x][newnode] = -1;  recurse(newnode, x, minCover, maxIndep, done);
    }
  }
  vector<int> getAnswer(bool isMinCover) {
    vector<int> minCover, maxIndep, done(n, 0);
    makeMatching();
    for (int x = 0; x < 2; x++)
      for (int i = 0; i < int(comp[x].size()); i++) {
        int node = comp[x][i];
        if (m[x][node] == -1) recurse(node, x, minCover, maxIndep, done);
      }
    for (int i = 0; i < int(comp[0].size()); i++)
      if (!done[comp[0][i]]) {
        minCover.push_back(comp[0][i]);
```

```
        maxIndep.push_back(m[0][comp[0][i]]);
      }
    return isMinCover ? minCover : maxIndep;
  }
};
```

Convex Hull trick DP

```
// Maximize result for linear function Y = A[i] * X + B[i]
// Function add require Lines to be added in sorted slope
// To minimize the result use addSorted with reverse order
vector<long long> A, B;
bool cmp(int l1, int l2, int l3) {
  return (B[l3] - B[l1])*(A[l1]-A[l2]) < (B[l2]-B[l1])*(A[l1] - A[l3]);
}
void add(long long a, long long b) {
  A.push_back(a); B.push_back(b);
  while (A.size() >= 3 && cmp(A.size() - 3, A.size() - 2, A.size() - 1))
    A.erase(A.end() - 2), B.erase(B.end() - 2);
}
long long f(long long x, int l) {  return A[l] * x + B[l];}
long long query(long long x) {
  int lo = 0, hi = A.size() - 2, res = A.size() - 1;
  while (lo <= hi) {
    int md = (lo + hi) >> 1;
    if (f(x, md) < f(x, md + 1)) lo = md + 1;
    else res = md, hi = md - 1;
  }
  return f(x, res);
}
```

Minimum Cost Max Flow SPFA implementation

```
const int mxN = 110;
const int inf = 1000000010;
struct Edge {
  int to, cost, cap, flow, backEdge;
};
struct MCMF {
  int s, t, n;
  vector<Edge> g[mxN];
  MCMF(int _s, int _t, int _n) {
    s = _s, t = _t, n = _n;
  }
  void addEdge(int u, int v, int cost, int cap) {
    Edge e1 = { v, cost, cap, 0, g[v].size() };
    Edge e2 = { u, -cost, 0, 0, g[u].size() };
    g[u].push_back(e1); g[v].push_back(e2);
  }
  pair<int, int> minCostMaxFlow() {
    int flow = 0, cost = 0;
    vector<int> state(n), from(n), from_edge(n), d(n);
    deque<int> q;
    while (true) {
      for (int i = 0; i < n; i++)
        state[i] = 2, d[i] = inf, from[i] = -1;
      state[s] = 1; q.clear(); q.push_back(s); d[s] = 0;
      while (!q.empty()) {
        int v = q.front(); q.pop_front(); state[v] = 0;
        for (int i = 0; i < (int) g[v].size(); i++) {
          Edge e = g[v][i];
          if (e.flow >= e.cap || d[e.to] <= d[v] + e.cost)
            continue;
          int to = e.to; d[to] = d[v] + e.cost;
          from[to] = v; from_edge[to] = i;
          if (state[to] == 1)  continue;
          if (!state[to] || (!q.empty() && d[q.front()] > d[to]))
            q.push_front(to);
          else q.push_back(to);
          state[to] = 1;
        }
      }
      if (d[t] == inf) break;
      int it = t, addflow = inf;
      while (it != s) {
        addflow = min(addflow,
            g[from[it]][from_edge[it]].cap
                - g[from[it]][from_edge[it]].flow);
        it = from[it];
      }
      it = t;
      while (it != s) {
        g[from[it]][from_edge[it]].flow += addflow;
        g[it][g[from[it]][from_edge[it]].backEdge].flow -= addflow;
        cost += g[from[it]][from_edge[it]].cost * addflow;
        it = from[it];
      }
      flow += addflow;
    }
    return {cost,flow};
  }
};
```

Adjacency list implementation of Dinic's blocking flow algorithm.

```
// This is very fast in practice, and only loses to push-relabel flow.
// Running time:
//     O(min(|E|,flow) * Layers)
//     In general graphs: Layers = O(|V|^2)
//     In bibpartite graphs: Layers = O(sqrt(|V|))
// INPUT:
//     - graph, constructed using AddEdge()
```

```
//      - source and sink
// OUTPUT:
//      - maximum flow value
//      - To obtain actual flow values, look at edges with capacity > 0
//        (zero capacity edges are residual edges).
typedef long long LL;
struct Edge {
  int u, v;
  LL cap, flow;
  Edge() {}
  Edge(int u, int v, LL cap) : u(u), v(v), cap(cap), flow(0) {}
};
struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;
  Dinic(int N) : N(N), E(0), g(N), d(N), pt(N) {}
  void AddEdge(int u, int v, LL cap) {
    if (u != v) {
      E.push_back(Edge(u, v, cap));
      g[u].push_back(E.size() - 1);
      E.push_back(Edge(v, u, 0));
      g[v].push_back(E.size() - 1);
    }
  }
  bool BFS(int S, int T) {
    queue<int> q( { S });
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k : g[u]) {
        Edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.push(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }
  LL DFS(int u, int T, LL flow = -1) {
    if (u == T || flow == 0)
      return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      Edge &e = E[g[u][i]];
      Edge &oe = E[g[u][i] ^ 1];
      if (d[e.v] == d[e.u] + 1) {
        LL amt = e.cap - e.flow;
        if (flow != -1 && amt > flow)
          amt = flow;
        if (LL pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }
  LL MaxFlow(int S, int T) {
    LL total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (LL flow = DFS(S, T))
        total += flow;
    }
    return total; } };
```

GomoryHu

```
/* Find min cut between every pair of vertices using N max_flow call
(instead of N^2)
 * Not tested with directed graph
 * Index start from 0
 */
const int mxN = 110, INF = 1000000010;
struct GomoryHu {
  int ok[mxN], cap[mxN][mxN];
  int answer[mxN][mxN], parent[mxN], n;
  Dinic flow;
  GomoryHu(int n) :
      n(n), flow(n) {
    for (int i = 0; i < n; ++i)  ok[i] = parent[i] = 0;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        cap[i][j] = 0, answer[i][j] = INF;
  }
  void addEdge(int u, int v, int c) { cap[u][v] += c; }
  void calc() {
    for (int i = 0; i < n; ++i) parent[i] = 0;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        answer[i][j] = 2000111000;
    for (int i = 1; i <= n - 1; ++i) {
      flow = Dinic(n);
```

```
      for (int u = 0; u < n; u++)
        for (int v = 0; v < n; v++)
          if (cap[u][v]) flow.AddEdge(u, v, cap[u][v]);
      int f = flow.MaxFlow(i, parent[i]);  bfs(i);
      for (int j = i + 1; j < n; ++j)
        if (ok[j] && parent[j] == parent[i]) parent[j] = i;
      answer[i][parent[i]] = answer[parent[i]][i] = f;
      for (int j = 0; j < i; ++j)
        answer[i][j] = answer[j][i] = min(f, answer[parent[i]][j]);
    }
  }
  void bfs(int start) {
    memset(ok, 0, sizeof ok);
    queue<int> qu; qu.push(start);
    while (!qu.empty()) {
      int u = qu.front();
      qu.pop();
      for (int xid = 0; xid < flow.g[u].size(); ++xid) {
        int id = flow.g[u][xid];
        int v = flow.E[id].v, fl = flow.E[id].flow,  cap =
flow.E[id].cap;
        if (!ok[v] && fl < cap) ok[v] = 1, qu.push(v);
      }
    }
  }
};
```

Theorems

```
/*  In any bipartite graph, the number of edges in a maximum matching
equals the number of vertices in a minimum vertex cover.
 *  In any graph,  maximum matching <= minimum vertex cover.
 *  In any graph, A set of vertices is a vertex cover if and only if its
complement is an independent set.
 *  In any graph, the number of vertices of a graph is equal to its
minimum vertex cover number plus the size of a maximum independent set.
 *  In any graph, Maximum matching + Minimum edge cover = V
 *  For a connected planar graph, the relationship between the number of
vertices V, the number of sides E, and the number of faces F is V - E + F
= 2.
 *  MaxFlow = MinCut [Minimum capacity required to make the graph
disconnected]
 *  In bipartite graph K_{n,m} , number of perfect matching is #P-
complete and can be found using DP[(1<<N)][M]. DP[S][j] = DP[S][j-1] +
DP[S/{v}][j-1] for each v connected to the j^{th} node.
 * Closure: set of vertices with no outgoing edges
 * Max closure = compliment of Min closure
 * Max closure = Min cut in H:
 - Add source s, sink t
 - if f(v) > 0 --> add edge (s, v) with capacity = f(v)
 - if f(v) < 0 --> add edge (v, t) with capacity = -f(v)
 - All edges in G have infinite capacity in H
 - Vertices in same side as s forms a closure. Weight(cut) = sum(f(v)
where f(v) > 0) - weight(closure) --> cut is minimum when closure is
maximum
 */
```

Hungarian Algorithm

```
/* w[i][j] = amount bidder j is willing to pay for item i (0 if he is not
bidding)
 * run time is O(nm^2) where n = #of items and m = #of bidders
 * resets negative bids in w to 0
 * returns a, where a[i] = j means ith item got assigned to bidder j
 * a[i] = -1 means item i did not get assigned
 * for minimizing set w[i][j] = max(w) - w[i][j]
 * for assigning all, w[i][j] = min(w) + w[i][j]
 */
const int INF = 1000000010;
vector<int> hungarianMethod(vector<vector<int> > w) {
  int n = w.size(), m = w[0].size(), PHI = -1, NOL = -2;
  int f = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      f = max(f, w[i][j]);
  vector<vector<bool> > x(n, vector<bool>(m));
  vector<bool> ss(n), st(m);
  vector<int> u(n, f), v(m), p(m,INF), ls(n, PHI), lt(m, NOL), a(n, -1);
  while (true) {
    f = -1;
    for (int i = 0; i < n && f == -1; i++)
      if (ls[i] != NOL && !ss[i])  f = i;
    if (f != -1) {
      ss[f] = true;
      for (int j = 0; j < m; j++)
        if (!x[f][j] && u[f] + v[j] - w[f][j] < p[j])
          lt[j] = f, p[j] = u[f] + v[j] - w[f][j];
    } else {
      for (int i = 0; i < m && f == -1; i++)
        if (lt[i] != NOL && !st[i] && p[i] == 0) f = i;
      if (f == -1) {
        int d1 = INF, d2 = INF, d;
        for (int i : u) d1 = min(d1, i);
        for (int i : p)  if (i > 0) d2 = min(d2, i);
        d = min(d1, d2);
        for (int i = 0; i < n; i++)  if (ls[i] != NOL) u[i] -= d;
        for (int i = 0; i < m; i++)
          if (p[i] == 0) v[i] += d;
          else if (p[i] > 0 && lt[i] != NOL) p[i] -= d;
        if (d2 >= d1)
```

```
          break;
        } else {
          st[f] = true; int s = -1;
          for (int i = 0; i < n && s == -1; i++)
            if (x[i][f]) s = i;
          if (s == -1) {
            for (int l, r;; f = r) {
              r = f; l = lt[r];
              if (r >= 0 && l >= 0) x[l][r] = !x[l][r];
              else break;
              r = ls[l];
              if (r >= 0 && l >= 0) x[l][r] = !x[l][r];
              else break;
            }
            fill(p.begin(), p.end(), INF);
            fill(lt.begin(), lt.end(), NOL);
            fill(ls.begin(), ls.end(), NOL);
            fill(ss.begin(), ss.end(), false);
            fill(st.begin(), st.end(), false);
            for (int i = 0; i < n; i++) {
              bool ex = true;
              for (int j = 0; j < m && ex; j++) ex = !x[i][j];
              if (ex)  ls[i] = PHI;
            }
          } else
            ls[s] = f;
        }
      }
    }
    for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
        if (x[i][j]) a[j] = i;
    return a;
}
```

KD Tree

```
// construct vector<int,int> different Points and call buildTree(0,size
of points, points)
// Use find findNearestNeighbour(size of points, point_x, point_y)
typedef pair<int, int> pii;
typedef vector<pii> vpii;
const int maxn = 100000;
int tx[maxn], ty[maxn];
bool divX[maxn];
bool cmpX(const pii &a, const pii &b) { return a.first < b.first; }
bool cmpY(const pii &a, const pii &b) { return a.second < b.second; }
void buildTree(int left, int right, pii points[]) {
  if (left >= right) return;
  int mid = (left + right) >> 1;
  //sort(points + left, points + right + 1, divX ? cmpX : cmpY);
  int minx = INT_MAX, maxx = INT_MIN, miny = INT_MAX, maxy = INT_MIN;
  for (int i = left; i < right; i++) {
    min(minx, points[i].first); max(maxx, points[i].first);
    min(miny, points[i].second); max(maxy, points[i].second);
  }
  divX[mid] = (maxx - minx) >= (maxy - miny);
  nth_element(points + left, points + mid, points + right, divX[mid] ?
cmpX : cmpY);
  tx[mid] = points[mid].first;
  ty[mid] = points[mid].second;
  if (left + 1 == right) return;
  buildTree(left, mid, points);
  buildTree(mid + 1, right, points);
}


long long closestDist; int closestNode;
void findNearestNeighbour(int left, int right, int x, int y) {
  if (left >= right)
    return;
  int mid = (left + right) >> 1;
  int dx = x - tx[mid], dy = y - ty[mid];
  long long d = dx * (long long) dx + dy * (long long) dy;
  if (closestDist > d && d) closestDist = d, closestNode = mid;
  if (left + 1 == right) return;
  int delta = divX[mid] ? dx : dy;
  long long delta2 = delta * (long long) delta;
  int l1 = left, r1 = mid, l2 = mid + 1, r2 = right;
  if (delta > 0) swap(l1, l2), swap(r1, r2);
  findNearestNeighbour(l1, r1, x, y);
  if (delta2 < closestDist) findNearestNeighbour(l2, r2, x, y);
}

int findNearestNeighbour(int n, int x, int y) {
  closestDist = LLONG_MAX;
  findNearestNeighbour(0, n, x, y);
  return closestNode;
}
```

Laguerre's method of polynom roots finding

```
typedef complex<double> cdouble;
typedef vector<cdouble> poly;
pair<poly, cdouble> horner(const poly &a, cdouble x0) {
  int n = a.size();
  poly b = poly(max(1, n - 1));
  for (int i = n - 1; i > 0; i--)
    b[i - 1] = a[i] + (i < n - 1 ? b[i] * x0 : 0);
  return make_pair(b, a[0] + b[0] * x0);
```

```
}
cdouble eval(const poly &p, cdouble x) { return horner(p, x).second; }
poly derivative(const poly &p) {
  int n = p.size(); poly r = poly(max(1, n - 1));
  for (int i = 1; i < n; i++) r[i - 1] = p[i] * cdouble(i);
  return r;
}
const double EPS = 1e-9;
int cmp(cdouble x, cdouble y) {
  double diff = abs(x) - abs(y);
  return diff < -EPS ? -1 : (diff > EPS ? 1 : 0);
}
cdouble find_one_root(const poly &p0, cdouble x) {
  int n = p0.size() - 1;
  poly p1 = derivative(p0);
  poly p2 = derivative(p1);
  for (int step = 0; step < 10000; step++) {
    cdouble y0 = eval(p0, x);
    if (cmp(y0, 0) == 0) break;
    cdouble G = eval(p1, x) / y0, H = G * G - eval(p2, x) - y0;
    cdouble R = sqrt(cdouble(n - 1) * (H * cdouble(n) - G * G));
    cdouble D1 = G + R, D2 = G - R;
    cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
    x -= a;
    if (cmp(a, 0) == 0) break;
  }
  return x;
}
vector<cdouble> find_all_roots(const poly &p) {
  vector<cdouble> res; poly q = p;
  while (q.size() > 2) {
    cdouble z(rand() / double(RAND_MAX), rand() / double(RAND_MAX));
    z = find_one_root(q, z); z = find_one_root(p, z);
    q = horner(q, z).first; res.push_back(z);
  }
  res.push_back(-q[0] / q[1]); return res;
}
int main(int argc, char* argv[]) {
  poly p;
// x^3 - 8x^2 - 13x + 140 = (x+4)(x-5)(x-7)
  p.push_back(140); p.push_back(-13);
  p.push_back(-8); p.push_back(1);
  vector<cdouble> roots = find_all_roots(p);
  for (size_t i = 0; i < roots.size(); i++) {
    if (abs(roots[i].real()) < EPS)
      roots[i] -= cdouble(roots[i].real(), 0);
    if (abs(roots[i].imag()) < EPS)
      roots[i] -= cdouble(0, roots[i].imag());
    cout << setprecision(3) << roots[i] << endl; } return 0; }
```

Blossom Algorithm

```
int lca(vector<int> &match, vector<int> &base, vector<int> &p, int a, int
b) {
  vector<bool> used(match.size());
  while (true) {
    a = base[a]; used[a] = true;
    if (match[a] == -1) break;
    a = p[match[a]];
  }
  while (true) {
    b = base[b];
    if (used[b]) return b;
    b = p[match[b]];
  }
}
void markPath(vector<int> &match, vector<int> &base, vector<bool>
&blossom, vector<int> &p,
    int v, int b, int children) {
  for (; base[v] != b; v = p[match[v]]) {
    blossom[base[v]] = blossom[base[match[v]]] = true;
    p[v] = children; children = match[v];
  }
}
int findPath(vector<vector<int> > &graph, vector<int> &match, vector<int>
&p, int root) {
  int n = graph.size(); vector<bool> used(n);
  fill(p.begin(), p.end(), -1); vector<int> base(n);
  for (int i = 0; i < n; i++) base[i] = i;
  used[root] = true; queue<int> q;
  q.push(root);
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int to : graph[v]) {
      if (base[v] == base[to] || match[v] == to)
        continue;
      if (to == root || (match[to] != -1 && p[match[to]] != -1)) {
        int curbase = lca(match, base, p, v, to);
        vector<bool> blossom(n);
        markPath(match, base, blossom, p, v, curbase, to);
        markPath(match, base, blossom, p, to, curbase, v);
        for (int i = 0; i < n; i++)
          if (blossom[base[i]]) {
            base[i] = curbase;
            if (!used[i])
              used[i] = true, q.push(i);
          }
```

```
      } else if (p[to] == -1) {
        p[to] = v; if (match[to] == -1) return to;
        to = match[to];  used[to] = true; q.push(to);
      }
    }
  }
  return -1;
}
int maxMatching(vector<vector<int> > &graph, vector<int> & match) {
  int n = graph.size();
  match = vector<int>(n, -1);
  vector<int> p(n);
  for (int i = 0; i < n; i++) {
    if (match[i] == -1) {
      int v = findPath(graph, match, p, i);
      while (v != -1) {
        int pv = p[v], ppv = match[pv];
        match[v] = pv; match[pv] = v; v = ppv;
      }
    }
  }
  int matches = 0;
  for (int i = 0; i < n; i++) if (match[i] != -1)  ++matches;
  return (matches >> 1);
}
```

Isomorphic Trees

```
/*
 * usage:
for(int i=0; i<MAX_N; i++) shaker[i] = rand();
read g[0] and g[1]
call isomorphic(0, 1);
 */
typedef unsigned long long hashh;
const int MAX_N = 1e5;
int N, V; vector<int> graph[2][MAX_N + 1];
int dist[MAX_N], dist2[MAX_N]; bool visited[MAX_N + 1];
hashh shaker[MAX_N];
//flood fill
void dfs(int k, int v) {
  visited[v] = true;
  for (int i = 0; i < (int) graph[k][v].size(); i++) {
    const int u = graph[k][v][i];
    if (!visited[u]) dist[u] = dist[v] + 1, dfs(k, u);
  }
}
pair<int, int> find_center(int k) {
  memset(visited, false, sizeof(visited));
  dist[0] = 0; dfs(k, 0);
  int e = max_element(dist, dist + N) - dist;
  memset(visited, false, sizeof(visited));
  memcpy(dist2, dist, sizeof(dist));
  dist[e] = 0; dfs(k, e);
  memset(visited, false, sizeof(visited));
  memcpy(dist2, dist, sizeof(dist));
  e = max_element(dist, dist + N) - dist; dist[e] = 0; dfs(k, e);
  int diameter = *max_element(dist, dist + N);
  pair<int, int> ret(-1, -1);
  for (int i = 0; i < N; i++) {
    if ((dist[i] == diameter / 2 || dist2[i] == diameter / 2)
        && dist[i] + dist2[i] == diameter) {
      if (ret.first == -1) ret.first = i;
      else ret.second = i;
    }
  }
  return ret;
}
hashh rec(int k, int v) {
  hashh ret = 1; visited[v] = true;
  vector<hashh> hs;
  for (int i = 0; i < (int) graph[k][v].size(); i++) {
    const int u = graph[k][v][i];
    if (!visited[u]) hs.push_back(rec(k, u));
  }
  sort(hs.begin(), hs.end());
  for (int i = 0; i < (int) hs.size(); i++) ret += hs[i] * shaker[i];
  return ret;
}
hashh calc_hash(int k) {
  pair<int, int> center = find_center(k); int root = center.first;
  if (center.second != -1) {
    root = N; const int v = center.first, u = center.second;
    graph[k][root].push_back(v); graph[k][root].push_back(u);
    *find(graph[k][v].begin(), graph[k][v].end(), u) = root;
    *find(graph[k][u].begin(), graph[k][u].end(), v) = root;
  }
  memset(visited, false, sizeof(visited)); return rec(k, root);
}
bool is_isomorphic() { return calc_hash(0) == calc_hash(1); }
```

Gauss Elimination

```
// a11x1+a12x2+···+a1mxm=b1
// last column in vvd a is the matrix b
int gauss(vector<vector<double> > a, vector<double> & ans) {
  int n = (int) a.size(), m = (int) a[0].size() - 1;
  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
```

```
    for (int i = row; i < n; ++i)
      if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;
    for (int i = col; i <= m; ++i) swap(a[sel][i], a[row][i]);
    where[col] = row;
    for (int i = 0; i < n; ++i)
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j = col; j <= m; ++j) a[i][j] -= a[row][j] * c;
      }
    ++row;
  }
  ans.assign(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1)  ans[i] = a[where[i]][m] / a[where[i]][i];
  for (int i = 0; i < n; ++i) {
    double sum = 0;
    for (int j = 0; j < m; ++j)  sum += ans[j] * a[i][j];
    if (abs(sum - a[i][m]) > EPS) return 0;
  }
  for (int i = 0; i < m; ++i) if (where[i] == -1) return INF;
  return 1;
}
```

Mod 2 Gauss

```
int gauss(vector<bitset<N> > a, int n, int m, bitset<N> & ans) {
  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    for (int i = row; i < n; ++i)
      if (a[i][col]) { swap(a[i], a[row]); break; }
    if (!a[row][col]) continue;
    where[col] = row;
    for (int i = 0; i < n; ++i)
      if (i != row && a[i][col])
        a[i] ^= a[row];
    ++row;
  }
  // The rest of implementation is the same as above
}
```

Numerical integration

```
//Adaptive Simpson works if there is no horizontal lines in the curve
double adaptiveSimpsonsAux(double (*f)(const double&), double a, double
b,
    double epsilon, double S, double fa, double fb, double fc, int
bottom) {
  double c = (a + b) / 2, h = b - a;
  double d = (a + c) / 2, e = (c + b) / 2;
  double fd = f(d), fe = f(e);
  double Sleft = (h / 12) * (fa + 4 * fd + fc);
  double Sright = (h / 12) * (fc + 4 * fe + fb);
  double S2 = Sleft + Sright;
  if (bottom <= 0 || fabs(S2 - S) <= 15 * epsilon)
    return S2 + (S2 - S) / 15;
  return adaptiveSimpsonsAux(f, a, c, epsilon / 2, Sleft, fa, fc, fd,
      bottom - 1)
      + adaptiveSimpsonsAux(f, c, b, epsilon / 2, Sright, fc, fb, fe,
          bottom - 1);
}
// Adaptive Simpson's Rule
double adaptiveSimpsons(double (*f)(const double&), // ptr to function
double a, double b, // interval [a,b]
    double epsilon, // error tolerance
    int maxRecursionDepth) { // recursion cap
  double c = (a + b) / 2, h = b - a;
  double fa = f(a), fb = f(b), fc = f(c);
  double S = (h / 6) * (fa + 4 * fc + fb);
  return adaptiveSimpsonsAux(f,a,b,epsilon,S,fa,fb,fc,maxRecursionDepth);
}
```

Pollard rho

```
class PollardRho {
  private final static BigInteger ZERO = new BigInteger("0");
  private final static BigInteger ONE = new BigInteger("1");
  private final static BigInteger TWO = new BigInteger("2");
  private final static SecureRandom random = new SecureRandom();

  public static BigInteger rho(BigInteger N) {
    BigInteger divisor;
    BigInteger c = new BigInteger(N.bitLength(), random);
    BigInteger x = new BigInteger(N.bitLength(), random);
    BigInteger xx = x;
    // check divisibility by 2
    if (N.mod(TWO).compareTo(ZERO) == 0)
      return TWO;
    do {
      x = x.multiply(x).mod(N).add(c).mod(N);
      xx = xx.multiply(xx).mod(N).add(c).mod(N);
      xx = xx.multiply(xx).mod(N).add(c).mod(N);
      divisor = x.subtract(xx).gcd(N);
    } while ((divisor.compareTo(ONE)) == 0);
    return divisor;
  }

  public static void factor(BigInteger N) {
    if (N.compareTo(ONE) == 0)
      return;
    if (N.isProbablePrime(20)) {
      System.out.println(N);
```

```
      return;
    }
    BigInteger divisor = rho(N);
    factor(divisor);
    factor(N.divide(divisor));
  }
}
```

Miller Rabin

```
typedef unsigned long long ll;
ll mul(ll b, ll e, ll m){
  ll r = 0; b%=m,e%=m;
  for(; e; e >>= 1) {
    if(e & 1) r = (r+b)%m;
    b = (b+b)%m;
  }
  return r;
}
ll power(ll b, ll e, ll m){ ll r = 1; for(; e; e >>= 1) {
    if(e & 1) r = mul(r, b, m); b = mul(b, b, m);
  } return r;
}
bool witness(ll n, ll s, ll d, ll a){
  ll x = power(a, d, n),y;
  while (s) {
    y = mul(x,x,n);
    if (y == 1 && x != 1 && x != n-1) return false;
    x = y; --s;
  }
  if (y != 1) return false;
  return true;
}
bool is_prime(ll n){
  if ((((!(n & 1)) && n != 2 ) || (n < 2) || (n % 3 == 0 && n != 3))
return false;
  if (n <= 3) return true;
  ll d = n / 2,s = 1;
  while (!(d & 1)) d /= 2, ++s;
  vector<ll> v = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 325, 9375,
28178, 450775, 9780504, 1795265022};
  for(auto i : v){
    if(i > n)break;
    if(n % i == 0)return false;
    if(!witness(n,s,d,i))return false;
  } return true; }
```

Chinese Remainder theorem

```
long long prod = 1;
for(int i = 0;i < int)primes.size();i++)
  prod = prod * primes[i];
for(int i = 0;i < (int)primes.size();i++){
  long long p = prod / primes[i];
  solution += mod[i] * p * pow_mod(p,primes[i]-2,primes[i]);
  solution %= prod;
}
```

Extended Euclidean

```
typedef long long ll;
//ax+by=gcd(a,b)
// a,b inputs, x,y outputs
// There will be many solution to the equation above on the form of:
// { (x+(k*b)/GCD(a,b)  , y-(k*a)/GCD(a,b)) | k belongs to Z} according
to Beziout's Identity
int eGCD(int a, int b, int &x, int &y) {
  x = 1;  y = 0;
  int nx = 0, ny = 1;
  int t, r;
  while (b) {
    r = a / b;t = a - r * b;a = b;  b = t;  t = x - r * nx;
    x = nx; nx = t;t = y - r * ny;y = ny; ny = t; }
  return a;  }
//ax+by=c
bool solveLDE(int a, int b, int c, int &x, int &y, int &g) {
  g = eGCD(a, b, x, y);x *= c / g; y *= c / g;  return (c % g) == 0; }
int modInv(int a, int m) {  //(a*mi)%m=1
  int mi, r;eGCD(a, m, mi, r);  return (mi + m) % m; }
```

Optimized Seive

```
// This is the famous "Yarin sieve", for use when memory is tight.
#define MAXSIEVE 100000000 // All prime numbers up to this
#define MAXSIEVEHALF (MAXSIEVE/2)
#define MAXSQRT 5000 // sqrt(MAXSIEVE)/2
char a[MAXSIEVE / 16 + 2];
#define isprime(n) (a[(n)>>4]&(1<<(((n)>>1)&7))) // Works when n is odd

int i, j;
memset(a,255,sizeof(a));
a[0]=0xFE;
for(i=1;i<MAXSQRT;i++)
if (a[i>>3]&(1<<(i&7)))
for(j=i+i+i+1;j<MAXSIEVEHALF;j+=i+i+1)
a[j>>3]&=~(1<<(j&7));
```

Solve linear congruences equation:

```
// - a[i] * x = b[i] MOD m[i] (mi don't need to be co-prime)
typedef long long ll;
bool linearCongruences(const vector<ll> &a, const vector<ll> &b,
    const vector<ll> &m, ll &x, ll &M) {
  ll n = a.size(); x = 0; M = 1;
  REP(i, n) {
    ll a_ = a[i] * M, b_ = b[i] - a[i] * x, m_ = m[i];
```

```
    ll y, t, g = extgcd(a_, m_, y, t);
    if (b_ % g) return false;
    b_ /= g; m_ /= g; x += M * (y * b_ % m_); M *= m_;
  }
  x = (x + M) % M; return true;
}
```

Geaometry

```
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  ); }
  bool operator<(const PT &p)  const { return
make_pair(x,y)<make_pair(p.x,p.y); }
  bool operator==(const PT &p)  const { return !(*this < p) && !(p <
*this); }
};
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; }
PT norm(PT x, double l)    { return x * sqrt(l*l / dot(x,x));}
istream &operator>>(istream &is, PT &p) {return is >> p.x >> p.y; }
ostream &operator<<(ostream &os, const PT &p) {return os << "(" << p.x <<
"," << p.y << ")";}
/*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 (assuming a != b)
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(c-a, b-a)/dot(b-a,b-a);
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}
PT reflectAroundLine(PT p, PT a, PT b) {
  return ProjectPointLine(p,a,b) * 2.0 - p;
}
// DISTANCES:
// DISTANCE point to segment or line = distance between the point and its
projection
// DISTNACE line to parallel line = distance between on end point to the
other line
// DISTANCE line to parallel segment = distance between on end point of
the line to the segment
// DISTANCE segment to segment = minimum distance of the four end points
to the other segment
// DISTANCE point to polygon = minimum distance to any edge if it's
outside
// DISTNACE Line\segment to polygon = minimum distance to any edge
// DISTANCE point to circle = distance point to center - R
// DISTANCE line to circle = distance between center to the line - R
// DISTANCE polygon to circle = minimum distance between circle and each
edge (if the center isn't inside)
// DISTANCE polygon to polygon = minimum distance between any pair of
edges
// DISTANCE circle to circle = distance between centers - sum of the two
Rs
// CIRCLES states: (D = distance between circles, r first radius, R
second radius)
// - Outside : D > r + R
// - Touch : D == r + R
// - Inside : D < R - r
// - Touch inside : D == R - r
//
// INTERSECTIONS:
// 1 = RIGTH, 0 = collinear, -1 = LEFT
int isLeft(PT o,PT a,PT b) {
    double isLeft = cross(a - o, b - o);
    return isLeft < -EPS ? -1 : (isLeft > EPS ? 1 : 0 );
}
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 a to b intersects with 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;
}
// ST Line ab intersect ST Line cd assuming unique intersection exists
// for line segments, 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);
}
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) { // st
line and r > 0
  vector<PT> ret;
  b = b-a, a = a-c;
  double A = dot(b, b), B = dot(a, b), C = dot(a, a) - r*r, 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 (a,r) and (b,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;
}
//return the common area of two circle
double cirAreaCut(double a, double r) {
    double s1 = a * r * r / 2;
    double s2 = sin(a) * r * r / 2;
    return s1 - s2;
}
double commonCircleArea(PT c1, double r, PT c2, double R) {
  if (r < R) swap(c1, c2),swap(r,R);
  double d = sqrt(dist2(c1,c2));
  if (d + R <= r + EPS) return R*R*acos(-1.0);
  if (d >= r + R - EPS) return 0.0;
  double a1 = acos((d*d + r*r - R*R) / 2 / d / r);
  double a2 = acos((d*d + R*R - r*r) / 2 / d / R);
  return cirAreaCut(a1*2, r) + cirAreaCut(a2*2, R);
}
pair<PT,double> getCircumcircle(PT a, PT b, PT c) {
  double d = 2.0 * (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y -
b.y));
  assert(fabs(d) > EPS);
  double x = (dot(a,a) * (b.y - c.y) + dot(b,b) * (c.y - a.y) + dot(c,c)
* (a.y - b.y)) / d;
  double y = (dot(a,a) * (c.x - b.x) + dot(b,b) * (a.x - c.x) + dot(c,c)
* (b.x - a.x)) / d;
  PT p(x, y);
  return {p, sqrt(dist2(p,a))};
}
pair<PT,double> getEnclosingCircle(vector<PT> &p) {
  random_shuffle(p.begin(), p.end());
  PT c(0,0);
  double r = 0;
  int n = p.size();
  for (int i = 1; i < n; i++)
    if (dist2(p[i],c) > r * r + EPS) {
      c = p[i],r = 0;
      for (int j = 0; j < i; j++)
        if (dist2(p[j],c) > r * r + EPS) {
          c = (p[i] + p[j]) / 2, r = sqrt(dist2(p[i], p[j])) / 2;
          for (int k = 0; k < j; k++)
            if (dist2(p[k],c) > r * r + EPS) {
              auto cir = getCircumcircle(p[i], p[j], p[k]);
              c = cir.first;
              r = cir.second;
            }
        }
    }
  return {c,r};
}
vector<PT> pointCircleTangent(const PT &x, double r, const PT &a) {
  double dist = dist2(x,a);
  vector<PT> res;
  if(fabs(dist) < EPS)
    res.push_back(a);
  else if(dist > r * 1LL * r) {
    PT v = a - x;
    v = v * (r * 1.0 / sqrt(dist));
    double theta = acos(sqrt(dist) / r);
    res.push_back(x + RotateCCW(v, theta));
```

```
    res.push_back(x + RotateCCW(v, -theta));
  }
  return res;
}
vector<PT> circleCircleTangent(PT &a, int r1, PT &b, int r2) {
  vector<PT> res;
  if(r1 == r2) {
    PT v = (b - a) / sqrt(dist2(a,b));
    PT p1 = PT(-v.y, v.x) * r1;
    PT p2 = PT(v.y, -v.x) * r1;
    res.push_back(p1 + a);
    res.push_back(p2 + a);
    res.push_back(p1 + b);
    res.push_back(p2 + b);
    return res;
  }
  if(r1 < r2) {
    swap(a, b);
    swap(r1, r2);
  }
  double r = r1 - r2;
  res = pointCircleTangent(a, r, b);
  PT t1 = res[0], t2 = res[1];
  PT p1 = (t1 - a) / sqrt(dist2(a,t1));
  PT p2 = (t2 - a)  / sqrt(dist2(a,t1));
  res.push_back(p1 * r1 + a);
  res.push_back(p2 * r1 + a);
  res.push_back(p1 * r2 + b);
  res.push_back(p2 * r2 + b);
  return res;
}
// To line by segment and validate the intersection lie inside the
segment
// INTERSECTION LINE->POLYGON: intersection beteween the line and any
edge
// intersect every edge with the line or cirlce
//
// POLYGON FUNCTIONS
// possibly non-convex polygon
// strict in -> 1 .. strict out -> 0 ... else -> random
bool PointInPolygon(const vector<PT> &p, PT q) {
  int cnt = 0;
  for (int i = 0; i < (int)p.size(); i++) {
    PT p1 = p[i], p2 = p[(i + 1)%p.size()];
    if (fabs(p1.y-p2.y) < EPS) continue;
    if (p1.y > p2.y) swap(p1, p2);
    if (p1.y <= q.y && q.y < p2.y) {
      double dx = p2.x - p1.x, dy = p2.y - p1.y;
      if (dy * p1.x + dx * (q.y - p1.y) >= q.x * dy)
        cnt++;
    }
  }
  return cnt % 2 == 1;
}
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < (int)p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}
// possibly non convex polygon, PT sorted CW or CCW
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  if(p.size() < 3) {return 0;}
  area += p[0].x * (p[1].y - p.back().y);
  for(int i = 1; i < (int)p.size(); i++)
    area += p[i].x * (p[i+1].y - p[i-1].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 < (int)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;
}
bool PolygonIsSimple(const vector<PT> &p) {
  for (int i = 0; i < (int)p.size(); i++) {
    for (int k = i+1; k < (int)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;
}
bool PolygonIsConvex(const vector<PT> &P) {
    int n = (int) P.size();
```

```
    if (n <= 2) return false;
    int start = isLeft(P[0],P[1],P[2]);
    for (int i = 1; i < n; i++)
        if (isLeft(P[i], P[(i+1) % n], P[(i+2) % n]) * start < 0)
            return false;
    return true;
}
vector<PT> convexHull(vector<PT> & poly) {
  sort(poly.begin(),poly.end());
  auto it = unique(poly.begin(),poly.end());
  poly.erase(it,poly.end());
  if(poly.size() <= 3)
    return poly;
  vector<PT> res;
  for(int i = 0;i < (int)poly.size();i++) {
    while(res.size() > 1 && cross(res.back() - res[res.size()-2],poly[i]
- res[res.size()-2]) < 0)
      res.pop_back();
    res.push_back(poly[i]);
  }
  int t = res.size();
  for(int i = (int)poly.size() - 1;i >= 0;i--) {
    while((int)res.size() > t && cross(res.back() - res[res.size()-
2],poly[i] - res[res.size()-2]) < 0)
      res.pop_back();
    res.push_back(poly[i]);
  }
  res.pop_back();
  poly = res;
  return res;
}
vector<PT> polygon_cut(const vector<PT> &v, const PT &a, const PT &b) {
    vector<PT> res;
    int n = v.size();
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        bool in1 = isLeft(a, b, v[i]) > 0;
        bool in2 = isLeft(a, b, v[j]) > 0;
        if (in1) res.push_back(v[i]);
        if (in1 ^ in2) {
            PT r = ComputeLineIntersection(a, b, v[i], v[j]);
            res.push_back(r);
        }
    }
    return res;
}
vector<PT> areaPolygonIntersection(const vector<PT> &a, const vector<PT>
&b) {
  int n = a.size();
  int m = b.size();
  vector<PT> res;
  for (int i = 0; i < n; i++)
    if (PointInPolygon(b, a[i]))
      res.push_back(a[i]);
  for (int i = 0; i < m; i++)
    if (PointInPolygon(a, b[i]))
      res.push_back(b[i]);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      if (SegmentsIntersect(a[i], a[(i + 1) % n], b[j], b[(j + 1) % m]))
        res.push_back(ComputeLineIntersection(a[i], a[(i + 1) % n], b[j],
b[(j + 1) % m]));
  return convexHull(res);
}
// CIRCLE CONSTUCTION
PT ComputeCircleCenter(PT a, PT b, PT c) { // given 3 points
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-
c));
}
//
// RULE:
// SINE RULE: A/sin(a) = B/sin(b) = c/sin(c)
// COSINE RULE: C^2=A^2+B^2-2AB*cos(c)
// TANGENT RULE: (A-B)/(A+B) = tan(1/2(a-b)) / tan(1/2(a+b))
// COTAN RULE: A/cot(a/2) = B/cot(b/2) = C/cot(c/2)
// INDENTICAL CASES: SSS, SAS, AAS
// SIMILIAR CASE: AAA
// Mollweide's formula: (A+B)/C = cos((a-b)/2)/sin(c/2), (A-B)/C =
sin((a-b)/2)/cos(c/2)
// AREA of triangle = (B * H)/2, SQRT(S*(S-A)*(S-B)*(S-C)), S = 1/2 *
PERIMETER
// AREA of circle = R^2 * PI, PERIMETER = 2 * PI * R, AREA of sector =
(r^2 * theta)/2
// AREA of trapzoid = (B1 + B2)/2 * H
// Manhattan -> KING: (x,y) -> (x + y,x - y)
// KING -> Manhattan: (x,y) -> ((x+y)/2,(x-y)/2)
//
// EXTEREME POINT:
long long mxDot(vector<PT> & poly,PT & p) {
  int N = poly.size();
  if(N <= 10) {
    long long res = LLONG_MIN;
    for(int i = 0;i < N;i++) res = fmax(res,dot(p,poly[i]));
    return res;
```

```
  }
  if(dot(p,poly[0]-poly[1]) > 0 && dot(p,poly[0]-poly[N-1]) > 0)
    return dot(p,poly[0]);
  poly.push_back(poly[0]);
  int lo = 0,hi = N;
  long long res = -1;
  while(true) {
    int md = (lo + hi)>>1;
    if(dot(p,poly[md]-poly[md+1]) > 0 && dot(p,poly[md]-poly[md-1]) > 0)
{
      res = dot(p,poly[md]);
      break;
    }
    if(dot(p,poly[lo+1]-poly[lo]) > 0) {
      if(dot(p,poly[md+1]-poly[md]) <= 0)  hi = md;
      else {
        if(dot(p,poly[lo]-poly[md]) > 0) hi = md;
        else lo = md;
      }
    } else {
      if(dot(p,poly[md+1]-poly[md]) > 0) lo = md;
      else {
        if(dot(p,poly[lo]-poly[md]) <= 0) hi = md;
        else lo = md;
      }
    }
  }
  poly.pop_back();
  return res;
}
// ROTATING CALIPERS: assume p is set of points
double rotatingCalipers(vector<PT> p) {
  if(p.size() <= 1) return 0;
  if(p.size() == 2) return sqrt(dist2(p[0],p[1]));
  sort(p.begin(),p.end());
  vector<PT> U,L;
  for(int i = 0;i < (int)p.size();i++) {
    while(L.size() > 1 && cross(L.back() - L[L.size()-2],p[i] -
L[L.size()-2]) < 0)
      L.pop_back();
    L.push_back(p[i]);
  }
  for(int i = (int)p.size() - 1;i >= 0;i--) {
    while(U.size() > 1 && cross(U.back() - U[U.size()-2],p[i] -
U[U.size()-2]) < 0)
      U.pop_back();
    U.push_back(p[i]);
  }
  int i = 0,j = L.size(); double res = 0.0;
  reverse(L.begin(),L.end());
  while(i < (int)U.size() && j >= 0) {
    res = fmax(res,dist2(u[i],l[j])); // yield
    if(i == (int)U.size()) j--;
    else if(j == 0)  i++;
    else if(cross(L[j] - L[j-1],U[i+1]-U[i]) > 0) i++;
    else j--;
  }
  return sqrt(res);
}
```