THE WHOLE LIB USES dcmp() FOR FLOATING-POINT COMPARISON

1 Snippets 2
- 1.1 Makefile 2
- 1.2 C++ Template 2
- 1.3 Overflow 2
- 1.4 Policy 2
- 1.5 .vimrc 2

2 Data Structures 2
- 2.1 Sparse Table 2
- 2.2 Link-Cut Tree 2
- 2.3 Treap 3
- 2.4 Persistent Treap 4
- 2.5 Wavelet Tree 4

3 Graphs 5
- 3.1 Johnson 5
- 3.2 Connectivity 6
  - 3.2.1 SCC and condensation 6
  - 3.2.2 BCC and condensation 6
- 3.3 Trees 7
  - 3.3.1 Heavy-Light Decomposition 7
  - 3.3.2 LCA 8
  - 3.3.3 Centroid Decomposition 8
- 3.4 Dominator Tree 9
- 3.5 2-SAT 10
- 3.6 Misc 11
  - 3.6.1 Stable Marriage 11
- 3.7 Notes 11
  - 3.7.1 Euler Tour / Path 11

4 Dynamic Programming 12
- 4.1 Optimization Techniques 12
  - 4.1.1 Envelope for Convex-Hull Trick 12
  - 4.1.2 Dynamic Envelope for Convex-Hull Trick 12
  - 4.1.3 Knuth Optimization 12
  - 4.1.4 Divide and Conquer Optimization 13

5 Number Theory 13
- 5.1 Euclidean Modular Inverse 13
- 5.2 Garner CRT 13
- 5.3 Fraction Comparison 13
- 5.4 Extended Euclidean 13
- 5.5 Miller-Rabin Primality Test 14

6 Strings 14
- 6.1 KMP 14
- 6.2 Z-Algorithm 15
- 6.3 Power 15
- 6.4 Suffix Array 15
- 6.5 Aho-Corasick 15
- 6.6 Suffix Automaton 16
- 6.7 Manacher 16
- 6.8 Palindromic Tree 17

7 Flows and Matching 17
- 7.1 SPFA Min-cost Max-flow 17
- 7.2 Potentials Min-cost Max-flow 18
- 7.3 Mincost Circulation 20
- 7.4 Dinic's 21
- 7.5 Kuhn Algorithm 22
- 7.6 Hopcroft-Karp 22
- 7.7 Hungarian Algorithm 23
- 7.8 Matching in general graph 23
- 7.9 Global Min Cut (Stoer-Wagner) 24
- 7.10 Simplex 24

8 Geometry 25
- 8.1 2D Geometry 25
  - 8.1.1 Points, Lines and Segments 25
- 8.2 3D Geometry 26
- 8.3 Complex Geometry 26
- 8.4 Polygon Intersection 28
- 8.5 Barycentric Stuff 28
- 8.6 Rotating Caliper 28

9 Algebra 29
- 9.1 Matrix 29
- 9.2 Gaussian Elimination 29
- 9.3 Fast Fourier Transform 29
- 9.4 Xor FFT 30

10 Non-Indexed Codes 31
- 10.1 Matroid Intersection 31
- 10.2 Burunduk1 (bridges) 32
- 10.3 Burunduk1 Incremental (bridges) 34
- 10.4 Minimum Lex Rotation 34
- 10.5 Bitmasks 34
- 10.6 Caliper Usage 35
- 10.7 Notes 36

1. Snippets

1.1. Makefile

```
                                # make a     -> compiles a.cpp into a
.SUFFIXES:
%: %.cpp
    @g++ $< -o $@ -Ddebug -g -fsanitize=address -fsanitize=undefined -Wall -Wextra
```

1.2. C++ Template

```cpp
// clang-format off
#include <bits/stdc++.h>
using namespace std;

#ifndef debug        // Escreva:
#define debug if (0) //   debug cout << x << endl;
#endif

typedef long long ll;
typedef unsigned long long ull;
typedef vector<int> vi;
typedef pair<int, int> ii;

const double EPS = 1e-7;
int dcmp(double a, double b = 0.0) { return a <= b + EPS ? (a + EPS < b) ? -1 : 0 : 1; }
```

1.3. Overflow

```cpp
bool __builtin_add_overflow(type1 a, type2 b, type3* res);
bool __builtin_sub_overflow(type1 a, type2 b, type3* res);
bool __builtin_mul_overflow(type1 a, type2 b, type3* res);
```

1.4. Policy

```cpp
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/detail/standard_policies.hpp>
#include <ext/pb_ds/tree_policy.hpp>        // Including tree_order_statistics_node_update
using __gnu_pbds;                           // earlier it was called pb_ds
template <typename Key,                     // Key type
          typename Mapped,                  // Mapped-policy
          typename Cmp_Fn = std::less<Key>, // Key comparison functor
          typename Tag = rb_tree_tag,       // Specifies which underlying data structure to use
          template <typename Const_Node_Iterator, typename Node_Iterator, typename Cmp_Fn_,
                    typename Allocator_> class Node_Update = null_node_update, // A policy for updating node invariants
          typename Allocator = std::allocator<char> > // An allocator type
class tree;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
void sample() {
  ordered_set X;
  cout << *X.find_by_order(1) << endl; // 2
  cout << X.order_of_key(400) << endl; // 5
}
```

1.5. .vimrc

```
set ai si noet ts=4 sw=4 bs=2 so=1000
set number
map<Home> ^
imap<Home> <c-o>^
nmap <s-tab> <<
nmap <tab> >>
imap <s-tab> <c-o><<
vmap <tab> >>
vmap <s-tab> <<
inoremap {<CR> {<C-o>o}<C-o>O

nmap <F12> :! setxkbmap us<CR>
nmap <s-F12> :! setxkbmap br<CR>
```

2. Data Structures

2.1. Sparse Table

```cpp
int arr[MAXN];
int st[MAXN + 1][MAXLOGN];
int n;
int computeLog(int n) {
  int i = 1;
  for (; (1 << i) <= n; i++) {}
  return --i;
}
// store value instead of key
// it can be easily modified to store keys
void compute() {
  for (int i = 0; i < n; i++) st[i][0] = arr[i];
  for (int j = 1; (1 << j) < n; j++) {
    for (int i = 0; i + (1 << j) - 1 < n; i++) { st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]); }
  }
}
int query(int i, int j) {
  // works in O(logn) and O(1) with precomputed logs
  int k = computeLog(j - i + 1);
  return min(st[i][k], st[j - (1 << k) + 1][k]);
}
```

2.2. Link-Cut Tree

```cpp
#include <bits/stdc++.h>
using namespace std;
namespace LinkCut {
  struct Node {
    int id;
    Node *left, *right, *parent;
    bool evert;
    Node() {
      left = right = parent = 0;
      evert = false;
    }
    Node(int x) {
      left = right = parent = 0;
      evert = false;
      id = x;
    }
    bool is_root() {
      return parent == 0 ||
```

```
      (parent->left != this && parent->right != this); // return true if
    node is root of its aux tree
    }
    void update() { // lazy part
      // clear this function if tree is ROOTED
      if (evert) {
        evert = false;
        swap(left, right);
        if (left != 0) left->evert ^= 1;
        if (right != 0) right->evert ^= 1;
      }
    }
    void refresh() { // normal part
      // function to refresh values of a node
    }
  };
  // add node u as (is_left ? "left" : "right") child of p
  void add_edge(Node* p, Node* u, bool is_left) {
    if (u != 0) u->parent = p;
    if (is_left)
      p->left = u;
    else
      p->right = u;
  }
  void rotate(Node* u) {
    Node* p = u->parent;
    Node* q = p->parent; // q may be null
    bool proot = p->is_root();
    bool is_left = (u == p->left);
    // create edges [(p, mid), (u, p), (q, u)] where (parent, child)
    // if u is left child, mid node is right child of u and vice-versa
    add_edge(p, is_left ? u->right : u->left, is_left);
    add_edge(u, p, !is_left);
    if (!proot)
      add_edge(q, u, p == q->left);
    else
      u->parent = q;
    p->refresh();
  }
  void splay(Node* u) {
    while (!u->is_root()) {
      Node* p = u->parent;
      Node* q = p->parent;
      // top-down update
      if (!p->is_root()) q->update();
      p->update();
      u->update();
      if (!p->is_root()) {
        if ((p->left == u) != (q->left == p)) // zig zag
          rotate(u);
        else
          rotate(p); // zig zig
      }
      rotate(u); // zig
    }
    u->update();
    u->refresh();
  }
  Node* access(Node* v) {
    Node* prev = 0;
    for (Node* u = v; u != 0; u = u->parent) {
      splay(u);
      u->right = prev;
      prev = u;
    }
    splay(v);
    return prev; // endpoint node from last broken preferred link
  }
  void reroot(Node* v) {
    access(v);
    v->evert ^= 1;
  }
  Node* find_root(Node* u) {
    // find root of real tree containing u
    // on ROOTED tree
    access(u);
    while (u->left != 0) u = u->left;
    splay(u);
    return u;
  }
  bool connected(Node* u, Node* v) {
    if (u == v) return true;
    access(u);
    access(v);
    return u->parent != 0;
  }
  void link(Node* v, Node* u) {
    // assume u and v are not connected
    // in ROOTED tree, v is supposed to be u's parent
    // and u must be a root in its real tree
    reroot(u); // change to reroot(u) if its a UNROOTED tree or access(u) to
    ROOTED
    u->parent = v;
  }
  void cut(Node* u) {
    // cut operation for ROOTED tree
    access(u);
    // assume u is not real tree root
    u->left->parent = 0;
    u->left = 0;
  }
  void cut(Node* u, Node* v) {
    // cut operation for UNROOTED tree
    // assume there's a edge between u and v
    // v will become u child in rerooted tree at u
    // and then v can be cut as in a rooted tree
    reroot(u);
    cut(v);
  }
  Node* lca(Node* u, Node* v) {
    access(u);
    return access(v);
  }
};
```

2.3. Treap

```
#include <bits/stdc++.h>
using namespace std;
// it may be good to use random_shuffle to generate the keys
struct Treap {
  Treap *l, *r;
  int sz, val, y;
  int x; // for explicit
  Treap(int a, int b = 0) {
    val = a;
    y = b;
    sz = 1;
    l = r = 0;
```

```cpp
  }
  static Treap* ptr;
  static Treap* make(int a, int b = 0) { return new (ptr++) Treap(a, b); }
  static Treap* make(Treap* k) { return new (ptr++) Treap(*k); }
};
typedef Treap* PT;
int getsz(PT no) {
  if (!no) return 0;
  return no->sz;
}
void update(PT no) {
  if (!no) return;
  no->sz = getsz(no->l) + getsz(no->r) + 1;
}
void push(PT no) {}
void merge(PT& no, PT l, PT r) {
  push(l);
  push(r);
  if (!l)
    no = r;
  else if (!r)
    no = l;
  else if (l->y > r->y) {
    merge(l->r, l->r, r);
    no = l;
  } else {
    merge(r->l, l, r->l);
    no = r;
  }
  update(no);
}
void split(PT no, PT& l, PT& r, int x, int acc = 0) {
  push(no);
  if (!no) return void(l = r = 0);
  int k = acc + getsz(no->l); // change for explicit key
  // by default returns left tree <= x and right tree > x
  if (x < k) {
    split(no->l, l, no->l, x, acc);
    r = no;
  } else {
    split(no->r, no->r, r, x, acc + getsz(no->l) + 1);
    l = no;
  }
  update(no);
}
// slow operations for explicit treaps (in implicit you just need to merge)
void insert(PT& no, PT nw) {
  PT no2;
  split(no, no, no2, nw->x);
  if (no->x != nw->x) merge(no, no, nw);
  merge(no, no, no2);
}
void erase(PT& no, PT nw) {
  PT no2;
  split(no, no, no2, nw->x);
  if (no->x == nw->x) {
    PT old = no;
    merge(no, no->l, no->r);
    delete old; // if dynamically allocated
  }
  merge(no, no, no2);
}
```

2.4. Persistent Treap

```cpp
#include <bits/stdc++.h>
using namespace std;
// it may be good to use random_shuffle to generate the keys
struct Treap {
  Treap *l, *r;
  int sz, val;
  int x; // for explicit
  Treap(int a) {
    val = a;
    sz = 1;
    l = r = 0;
  }
  static Treap* ptr; // create a pool for this
  static Treap* make(int a) { return new (ptr++) Treap(a); }
  static Treap* make(Treap* k) { return new (ptr++) Treap(*k); }
};
typedef Treap* PT;
int getsz(PT no) {
  if (!no) return 0;
  return no->sz;
}
void update(PT no) {
  if (!no) return;
  no->sz = getsz(no->l) + getsz(no->r) + 1;
}
void push(PT no) {}
void merge(PT& no, PT l, PT r) {
  push(l);
  push(r);
  int k = rand() % (getsz(l) + getsz(r) + 1);
  if (!l)
    no = r;
  else if (!r)
    no = l;
  else if (k < getsz(l)) {
    no = Treap::make(l);
    merge(no->r, no->r, r);
  } else {
    no = Treap::make(r);
    merge(no->l, l, no->l);
  }
  update(no);
}
void split(PT no, PT& l, PT& r, int x, int acc = 0) {
  push(no);
  if (!no) return void(l = r = 0);
  int ls = getsz(no->l);
  int k = acc + ls; // change for explicit key
  // by default returns left tree <= x and right tree > x
  if (x < k) {
    no = Treap::make(no);
    split(no->l, l, no->l, x, acc);
    r = no;
  } else {
    no = Treap::make(no);
    split(no->r, no->r, r, x, acc + ls + 1);
    l = no;
  }
  update(no);
}
int main() {}
```

2.5. Wavelet Tree

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

template<typename T = int>
struct Wavelet{
  typedef typename vector<T>::iterator It;
  vector<T> a; // for swap
  vector<vector<int>> t;
  int sig;

  Wavelet(vector<T> a, T sig) : a(a), t(4*sig), sig(sig){
    build(1, a.begin(), a.end(), 0, sig-1);
  }

  void build(int no, It st, It nd, int L, int R){
    if(L == R) return;
    int M = (L+R)/2;
    t[no].pb(0);
    for(auto it = st; it != nd; it++)
      t[no].pb(t[no].back() + (*it <= M));

    It mid = stable_partition(st, nd, [M](T x){ return x <= M; });
    build(2*no, st, mid, L, M);
    build(2*no+1, mid, nd, M+1, R);
  }

  // all intervals are open [0, i)
  int mapL(int no, int i){ return t[no][i]; }
  int mapR(int no, int i){ return i-t[no][i]; }

  // find freq of x in interval [0, i)
  int rank(int x, int i){
    int L = 0, R = sig-1;
    int no = 1;

    while(L != R){
      int M = (L+R)/2;
      if(x <= M) i = mapL(no, i), no = 2*no, R = M;
      else i = mapR(no, i), no = 2*no+1, L = M+1;
    }

    return i;
  }

  // find k-th smallest in interval [i, j], with k>=1
  int quantile(int k, int i, int j){
    j++;
    int L = 0, R = sig-1;
    int no = 1;
    while(L != R){
      int c = mapL(no, j) - mapL(no, i);
      int M = (L+R)/2;
      if(k <= c)
        i = mapL(no, i), j = mapL(no, j), R = M, no = 2*no;
      else
        i = mapR(no, i), j = mapR(no, j), L = M+1, no = 2*no+1, k-=c;
    }

    return L;
  }

  // [i, j], [x, y]
  int x, y;
  int range(int x, int y, int i, int j){
    this->x = x, this->y = y;
    return _range(1, 0, sig-1, i, j+1);
  }

  // [i, j), [x, y]
  int _range(int no, int L, int R, int i, int j){
    if(y < L || x > R) return 0;
    if(x <= L && R <= y) return j-i;
    int M = (L+R)/2;
    return _range(2*no, L, M, mapL(no, i), mapL(no, j)) +
      _range(2*no+1, M+1, R, mapR(no, i), mapR(no, j));
  }

  // swap a[i], a[i+1]
  void swap(int i){
    int o = i++;

    int L = 0, R = sig-1;
    int no = 1;
    while(L != R){
      int M = (L+R)/2;
      if(a[o] <= M){
        if(a[o+1] > M){
          t[no][i]--;
          break;
        }
      } else if(a[o+1] <= M){
        t[no][i]++;
        break;
      }

      if(a[o+1] <= M) i = mapL(no, i), no = 2*no, R = M;
      else i = mapR(no, i), no = 2*no+1, L = M+1;
    }

    std::swap(a[o], a[o+1]);
  }

  // for toggle update queries, mantain BITs in nodes for
  // activeLeaf, activeLeft, activeRight, and modify queries accordingly
};
```

3. Graphs

3.1. Johnson

```
struct Edge {
  int next, cost;
  Edge(const int next, const int cost) : next(next), cost(cost) {}
};
struct State {
  int vert, cost;
  State(const int vert, const int cost) : vert(vert), cost(cost) {}
};
bool operator<(const State lhs, const State rhs) { return lhs.cost >
    rhs.cost; }
void johnson() {
  int n, m;
  scanf("%d%d", &n, &m);
  vector<Edge> adj[n + 1];
  while (m--) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
```

```
    adj[u].emplace_back(v, w);
  }
  // pot[v] <= pot[u] + cost(u, v) ==> cost(u, v) + pot[u] - pot[v] >= 0
  int pot[n + 1];
  memset(pot, 0, sizeof pot);
  for (;;) {
    bool changed = false;
    for (int u = 1; u <= n; ++u)
      for (const Edge e : adj[u])
        if (pot[u] + e.cost < pot[e.next]) pot[e.next] = pot[u] + e.cost, changed = true;
    if (!changed) break;
  }
  int dist[n + 1][n + 1];
  memset(dist, 0x3f, sizeof dist);
  for (int s = 1; s <= n; ++s) {
    priority_queue<State> q;
    q.emplace(s, 0);
    while (!q.empty()) {
      const State us = q.top();
      q.pop();
      const int u = us.vert;
      const int uc = us.cost;
      if (dist[s][u] != 0x3f3f3f3f) continue;
      dist[s][u] = uc;
      for (const Edge e : adj[u]) {
        const int v = e.next;
        const int vc = uc + e.cost + pot[u] - pot[v];
        if (dist[s][v] == 0x3f3f3f3f) q.emplace(v, vc);
      }
    }
    for (int t = 1; t <= n; ++t)
      if (dist[s][t] != 0x3f3f3f3f) dist[s][t] += pot[t] - pot[s];
  }
}
```

3.2. Connectivity

3.2.1. SCC and condensation

```
// suppose list of adj. exists
// "" global index
// "" visit time
// "" stack s
// "" suppose result vector res
// "" suppose result array from
vector<int> adj[MAX_N];
int idx;
int tempo[MAX_N];
vector<vector<int> > comps;
stack<int> st;
void tarjan(int n) {
  idx = 0;
  s = stack<int>();
  comps.clear();
  for (int i = 0; i < n; i++) {
    if (!tempo[i]) scc(i);
  }
}
int scc(int i) {
  int lowlink = tempo[i] = ++idx;
  s.push(i);
  for (int k : adj[i]) {
    if (!tempo[k])
      lowlink = min(lowlink, scc(k));
    else if (tempo[k] > 0) // on stack
      lowlink = min(lowlink, tempo[k]);
  }
  if (lowlink == tempo[i]) {
    vector<int> comp;
    int c = comps.size();
    int k;
    do {
      k = s.pop();
      comp.push_back(k);
      in_comp[k] = c;
      tempo[k] = -1;
    } while (k != i);
    comps.push_back(comp);
  }
  return lowlink;
}
// suppose vector res exists and is populated
// suppose array from exists and is populated
// ""  cadj exists to be populated
void contract() {
  vector<bool> visited(res.size());
  for (int i = 0; i < res.size(); i++) {
    visited[i] = true;
    for (int u : res[i])
      for (int v : adj[u])
        if (!visited[from[v]]) cadj[i].push_back(from[v]), visited[from[v]] = true;
    for (int u : res[i])
      for (int v : adj[u]) visited[from[v]] = false;
    visited[i] = false;
  }
}
```

3.2.2. BCC and condensation

```

int tempo;
int lowlink[MAXN], visited[MAXN];
vector<vector<int> > comps;
vector<int> st;
void bcc(int u, int p) {
  visited[u] = lowlink[u] = ++tempo;
  st.push_back(u);
  for (int v : adj[u]) {
    if (v == p) {
      p = -1; // this allow multiple edges
      continue;
    }
    if (!visited[v]) {
      bcc(v, u);
      lowlink[u] = min(lowlink[u], lowlink[v]);
      // check for bridges
      if (lowlink[v] > visited[u]) {
        vector<int> aux;
        int x;
        do {
          x = st.back();
          aux.push_back(x);
          st.pop_back();
        } while (x != v);
```

```
        comps.push_back(aux);
      }
    } else
      lowlink[u] = min(lowlink[u], visited[v]);
  }
}
int main() {
  // iterate over all vertices
  // after each call, check if there are still some vertices in the stack
  for (int i = 0; i < n; i++) {
    if (!visited[i]) {
      bcc(i, -1);
      if (!st.empty()) comps.push_back(st);
      st.clear();
    }
  }
}
```

3.3. Trees

3.3.1. Heavy-Light Decomposition

```
///// Note: 0-indexed vertices /////
class HLD {
  private:
  int current_index;
  void hld0(int v);
  void hld1(int v);

  public:
  std::vector<int>* adj;
  int *parent, *subtree_size, *special_child, *depth;
  int *head, *tail, *idx, *rev, *subtree_end, *relative_idx;
  HLD(int n);
  ~HLD();
  void add_edge(int u, int v);
  void add_arc(int u, int v);
  void compute();
  int lca(int u, int v);
  int dist(int u, int v);
};
HLD::HLD(const int n) {
  adj = new vector<int>[n];
  parent = new int[n];
  subtree_size = new int[n];
  special_child = new int[n];
  depth = new int[n];
  head = new int[n];
  tail = new int[n];
  idx = new int[n];
  rev = new int[n];
  subtree_end = new int[n];
  relative_idx = new int[n];
}
HLD::~HLD() {
  delete[] adj;
  delete[] parent;
  delete[] subtree_size;
  delete[] special_child;
  delete[] depth;
  delete[] head;
  delete[] tail;
  delete[] idx;
  delete[] rev;
  delete[] subtree_end;
  delete[] relative_idx;
}
void HLD::add_edge(const int u, const int v) {
  adj[u].push_back(v);
  adj[v].push_back(u);
}
void HLD::add_arc(const int u, const int v) { adj[u].push_back(v); }
void HLD::compute() {
  parent[0] = -1;
  depth[0] = 0;
  hld0(0);
  current_index = 0;
  head[0] = 0;
  hld1(0);
}
void HLD::hld0(const int v) {
  subtree_size[v] = 1;
  special_child[v] = -1;
  for (int i = adj[v].size() - 1; i >= 0; --i) {
    const int w = adj[v][i];
    if (w == parent[v]) continue;
    depth[w] = 1 + depth[v];
    parent[w] = v;
    hld0(w);
    subtree_size[v] += subtree_size[w];
  }
  for (int i = adj[v].size() - 1; i >= 0; --i) {
    const int w = adj[v][i];
    if (w == parent[v]) continue;
    if (subtree_size[w] > subtree_size[v] / 2) special_child[v] = w;
  }
}
void HLD::hld1(const int v) {
  idx[v] = current_index;
  rev[current_index] = v;
  current_index += 1;
  relative_idx[v] = idx[v] - idx[head[v]];
  if (special_child[v] == -1) {
    tail[v] = v;
  } else {
    head[special_child[v]] = head[v];
    hld1(special_child[v]);
    tail[v] = tail[special_child[v]];
  }
  for (int i = adj[v].size() - 1; i >= 0; --i) {
    const int w = adj[v][i];
    if (w == parent[v]) continue;
    if (w == special_child[v]) continue;
    head[w] = w;
    hld1(w);
  }
  subtree_end[v] = current_index - 1;
}
int HLD::lca(int u, int v) {
  while (head[u] != head[v]) {
    if (depth[head[u]] < depth[head[v]])
      v = parent[head[v]];
    else
      u = parent[head[u]];
  }
  return depth[u] < depth[v] ? u : v;
}
int HLD::dist(const int u, const int v) { return depth[u] + depth[v] - 2 *
    depth[lca(u, v)]; }
```

```cpp
#include <vector>
using namespace std;
template <int N> struct HLD {
  vector<int> P, h, L, pos, ch;
  int n, root;
  vector<vector<int> >& adj;
  HLD(vector<vector<int> >& adj, int n, int r = 0) : adj(adj), n(n), root(r)
    {
    P.assign(n, -1);
    h.assign(n, -1);
    L.resize(n);
    pos.resize(n);
    ch.resize(n);
  }
  int dfs(int u) {
    int sz = 1, best = 0;
    for (int v : adj[u]) {
      if (v != P[u]) {
        P[v] = u;
        L[v] = L[u] + 1;
        int subsz = dfs(v);
        if (subsz > best) {
          best = subsz;
          h[u] = v;
        }
        sz += subsz;
      }
    }
    return sz;
  }
  void init() {
    for (int i = 0; i < n; i++) P[i] = -1, h[i] = -1;
    L[root] = 0;
    dfs(root); // 0 as root
    for (int i = 0, ptr = 0; i < n; i++) {
      if (P[i] == -1 || h[P[i]] != i) {
        for (int j = i; j != -1; j = h[j]) {
          ch[j] = i;
          pos[j] = ptr++;
        }
      }
    }
  }
  template <class Func> int lift(int u, int v, Func f, bool is_edge = false)
    {
    for (; ch[u] != ch[v]; u = P[ch[u]]) {
      if (L[ch[u]] < L[ch[v]]) swap(u, v);
      f(pos[ch[u]], pos[u]);
    }
    if (L[u] > L[v]) swap(u, v);
    // handle edges adding +1 and taking care with R>L case on interval
    if (is_edge && pos[u] + 1 <= pos[v])
      f(pos[u] + 1, pos[v]);
    else if (!is_edge)
      f(pos[u], pos[v]);
    return u;
  }
  // implement query and update functions
};
```

3.3.2. LCA

```cpp
#include <bits/stdc++.h>
using namespace std;
const int MAX_N = 1e5;
const int MAX_LOGN = 20;
vector<int> adj[MAX_N];
bool visited[MAX_N];
int level[MAX_N];
int P[MAX_N][MAX_LOGN];
int n;
int dfsLevel;
// tree must be 0-indexed
// call with p as -1 for the root
void dfs(int u, int p) {
  visited[u] = true;
  level[u] = dfsLevel;
  P[u][0] = p;
  for (int v : adj[u]) {
    if (!visited[v]) {
      dfsLevel++;
      dfs(v, u);
      dfsLevel--;
    }
  }
}
/////////////////
// LCA PART BELOW
int computeLog(int x) {
  int i;
  for (i = 1; (1 << i) <= x; i++)
    ;
  return --i;
}
void computeLCA() {
  for (int j = 1; (1 << j) < n; j++) {
    for (int i = 0; i < n; i++) {
      if (P[i][j - 1] != -1) { P[i][j] = P[P[i][j - 1]][j - 1]; }
    }
  }
}
int LCA(int p, int q) {
  if (level[p] < level[q]) swap(p, q);
  int logP = computeLog(level[p]);
  for (int i = logP; i >= 0; i--) {
    if (level[p] - (1 << i) >= level[q]) p = P[p][i];
  }
  if (p == q) return p;
  logP = computeLog(level[p]);
  for (int i = logP; i >= 0; i--) {
    if (P[p][i] != -1 && P[p][i] != P[q][i]) {
      p = P[p][i];
      q = P[q][i];
    }
  }
  return P[p][0];
}
```

3.3.3. Centroid Decomposition

```cpp
const int MAX_N = 100000;
```

```cpp
vector<int> adj[MAX_N + 1];
bool is_centroid[MAX_N + 1];
int subtree_size[MAX_N + 1];
vector<int> vcentroids[MAX_N + 1];
vector<int> vdistances[MAX_N + 1];
void compute_sizes(const int u, const int v) {
  subtree_size[v] = 1;
  for (const int w : adj[v])
    if (!is_centroid[v])
      if (w != u) {
        compute_sizes(v, w);
        subtree_size[v] += subtree_size[w];
      }
}
int find_centroid(const int r, const int u, const int v) {
  for (const int w : adj[v])
    if (!is_centroid[w])
      if (w != u)
        if (subtree_size[w] > subtree_size[r] / 2) return find_centroid(r, v, w);
  return v;
}
void process_layer(const int centroid, const int u, const int v, const int dist) {
  vcentroids[v].push_back(centroid);
  vdistances[v].push_back(dist);
  for (const int w : adj[v])
    if (!is_centroid[w])
      if (w != u) process_layer(centroid, v, w, dist + 1);
}
void centroid_decomposition(const int s) {
  compute_sizes(-1, s);
  const int v = find_centroid(s, -1, s);
  process_layer(v, -1, v, 0);
  is_centroid[v] = true;
  for (const int w : adj[v])
    if (!is_centroid[w]) centroid_decomposition(w);
}
```

```cpp
struct LevelComp{
    int level;
    LevelComp(){}
    LevelComp(int level) : level(level) {}
    bool operator()(const int a, const int b){
        return dist[a][level] < dist[b][level] || (dist[a][level] == dist[b][level] && a < b);
    }
};
set<int, LevelComp> st[100001];

int find_centroid(int u, int p, int sz, int & center){
    int res = 1;
    for(Edge e : adj[u]){
        int v = e.next;
        if(removed[v] || v == p) continue;
        res += find_centroid(v, u, sz, center);
    }
    if(center == -1 && 2*res >= sz) center = u;
    return res;
}

void precompute_dist(int u, int p, int level){
    for(Edge e : adj[u]){
        int v = e.next;
        if(removed[v] || v == p) continue;
        dist[v][level] = dist[u][level] + e.weight;
        precompute_dist(v, u, level);
    }
}

void decompose(int u, int level, int p){
    int center = 0;
    int sz = find_centroid(u, -1, 0, center);
    center = -1; find_centroid(u, -1, sz, center);

    //printf("center: %d size: %d\n", center, sz);
    if(level == 0) root = center;
    pi[center] = p;
    L[center] = level;

    st[center] = set<int, LevelComp>(LevelComp(level));
    dist[center][level] = 0;
    precompute_dist(center, -1, level);

    removed[center] = true;
    for(Edge e : adj[center]){
        int v = e.next;
        if(removed[v]) continue;
        decompose(v, level+1, center);
    }
    removed[center] = false;
}

void update(int x){
    for(int center = x; center != -1; center = pi[center]){
        if(black[x]) st[center].insert(x);
        else st[center].erase(x);
    }
    black[x] = !black[x];
}

int query(int x){
    int res = 0x3f3f3f3f;
    for(int center = x; center != -1; center = pi[center]){
        if(!st[center].empty()) {
            int v = *st[center].begin();
            res = min(res, dist[v][L[center]] + dist[x][L[center]]);
        }
    }

    return res;
}
```

3.4. Dominator Tree

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

// 0 -indexed, O(nlogn) because it does not use rank-heuristic
template<typename T = int>
struct LinkDsu{
  vector<int> r;
  vector<T> best;
  LinkDsu(int n = 0){
    r = vector<int>(n); iota(r.begin(), r.end(), 0);
    best = vector<T>(n);
  }

```

```
  int find(int u) {
    if (r[u] == u)
      return u;
    else {
      int v = find(r[u]);
      if (best[r[u]] < best[u]) best[u] = best[r[u]];
      return r[u] = v;
    }
  }

  T eval(int u){ find(u); return best[u]; }
  void link(int p, int u) { r[u] = p; }
  void set(int u, T x) { best[u] = x; }
};

struct DominatorTree{
  typedef vector<vector<int>> Graph;
  vector<int> semi, dom, parent, st, from;
  Graph succ, pred, bucket;
  int r, n, tempo;

  void dfs(int u, int p){
    semi[u] = u;
    from[st[u] = tempo++] = u;
    parent[u] = p;
    for (int v : succ[u]) {
      pred[v].push_back(u);
      if (semi[v] == -1) { dfs(v, u); }
    }
  }

  void build(){
    n = succ.size();
    dom.assign(n, -1);
    semi.assign(n, -1);
    parent.assign(n, -1);
    st.assign(n, 0);
    from.assign(n, -1);
    pred = Graph(n, vector<int>());
    bucket = Graph(n, vector<int>());
    LinkDsu<pair<int,int>> dsu(n);
    tempo = 0;

    dfs(r, r);
    for(int i = 0; i < n; i++) dsu.set(i, make_pair(st[i],  i));

    for (int i = tempo - 1; i; i--) {
      int u = from[i];
      for (int v : pred[u]) {
        int w = dsu.eval(v).second;
        if (st[semi[w]] < st[semi[u]]) { semi[u] = semi[w]; }
      }
      dsu.set(u, make_pair(st[semi[u]], u));
      bucket[semi[u]].push_back(u);
      dsu.link(parent[u], u);
      for(int v : bucket[parent[u]]) {
        int w = dsu.eval(v).second;
        dom[v] = semi[w] == parent[u] ? parent[u] : w;
      }
      bucket[parent[u]].clear();
    }
    for (int i = 1; i < tempo; i++) {
      int u = from[i];
      if (dom[u] != semi[u]) dom[u] = dom[dom[u]];
    }
  }

  DominatorTree(const Graph & g, int s) : succ(g), r(s) {
    build();
  }
};
```

3.5. 2-SAT

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

#define POS(x) (2*(x))
#define NEG(x) (2*(x)+1)

struct TwoSAT{
  int n, sz;
  vector<vector<int>> adj;

  int tempo, cnt;
  vector<int> low, vis, from;
  stack<int> st;
  vector<bool> res;

  TwoSAT(int n) : n(n), adj(2*n){}

  int add_dummy() {
    int res = adj.size();
    for(int i = 0; i < 2; i++)
      adj.push_back(vector<int>());
    return res;
  }

  int convert(int x) const { return 2*x; }
  void add_edge(int a, int b) { adj[a].push_back(b); }
  void or_clause(int a, int b){
    add_edge(a^1, b);
    add_edge(b^1, a);
  }

  void implication_clause(int a, int b){
    or_clause(a^1, b);
  }

  void literal_clause(int x) { or_clause(x, x); }
  void and_clause(int a, int b){
    literal_clause(a);
    literal_clause(b);
  }

  void xor_clause(int a, int b){
    or_clause(a, b);
    or_clause(a^1, b^1);
  }

  void nand_clause(int a, int b){
    or_clause(a^1, b^1);
  }

  void nor_clause(int a, int b){
    literal_clause(a^1);
    literal_clause(b^1);
  }

```

```
  void equals(int a, int b){
    implication_clause(a, b);
    implication_clause(b, a);
  }

  void max_one_clause(const vector<int> & v){
    vector<int> p;
    for(int i = 0; i < v.size(); i++)
      p.push_back(add_dummy());

    for(int i = 0; i < v.size(); i++){
      implication_clause(v[i], p[i]);
      if(i+1 < v.size()){
        implication_clause(p[i], p[i+1]);
        implication_clause(p[i], v[i+1]^1);
      }
    }
  }

  void clear(){
    for(int i = 0; i < adj.size(); i++)
      adj[i].clear();
  }

  void tarjan(int u){
    low[u] = vis[u] = ++tempo;
    st.push(u);

    for(int v : adj[u]){
      if(!vis[v]){
        tarjan(v);
        low[u] = min(low[u], low[v]);
      } else if(vis[v] > 0)
        low[u] = min(low[u], vis[v]);
    }

    if(low[u] == vis[u]){
      int k;
      do{
        k = st.top();
        st.pop();
        from[k] = cnt;
        vis[k] = -1;
      } while(k != u);
      cnt++;
    }
  }

  bool solve(){
    sz = adj.size();
    assert(sz%2 == 0);

    low.assign(sz, 0);
    vis.assign(sz, 0);
    tempo = 0;
    cnt = 0;
    from.assign(sz, -1);
    st = stack<int>();

    res.assign(n, true);

    for(int i = 0; i < sz; i++)
      if(!vis[i])
        tarjan(i);

    for(int i = 0; i < sz; i += 2){
      if(from[i] == from[i^1]) return false;
      else if(from[i] > from[i^1] && (i>>1) < n)
        res[i>>1] = false;
    }

    return true;
  }

  bool get(int i) const { return res[i]; }
};
```

2

3.6. Misc

3.6.1. Stable Marriage

```
int wpref[n][n], mpref[n][n];
// ...
int pos[n];
memset(pos, 0, sizeof pos);
for (;;) {
  int suitors_count[n];
  int chosen[n];
  memset(suitors_count, 0, sizeof suitors_count);
  memset(chosen, -1, sizeof chosen);
  for (int m = 0; m < n; ++m) {
    const int w = mpref[m][pos[m]];
    ++suitors_count[w];
    if (chosen[w] == -1 || wpref[w][m] < wpref[w][chosen[w]]) chosen[w] = m;
  }
  for (int m = 0; m < n; ++m) {
    const int w = mpref[m][pos[m]];
    if (chosen[w] != m) ++pos[m];
  }
  const int max_suitors_count = *max_element(suitors_count, suitors_count + n);
  if (max_suitors_count == 1) break;
}
// ...
for (int m = 0; m < n; ++m) printf("%d %d\n", m + 1, mpref[m][pos[m]] + 1);
```

3.7. Notes

3.7.1. Euler Tour / Path

```
/*
UNDIRECTED GRAPH:
Check if every vertex has even degree and graph is connected (euler tour)
Check if two vertices have odd degree and graph is connected (euler path)

To find the edges, choose a start point and DFS all the edges, picking it
and removing it from the graph.
If you want an euler path, set one of the vertices with odd degree as the start point.

DIRECTED GRAPH:
Check if every vertex has indeg == outdeg and graph is strongly connected (euler tour)
Check if one vertex has indeg+1 == outdeg, another one has outdeg+1 == indeg and
graph is strongly connected (euler path)
```

```

Algorithm is the same to find euler tour/path, but now the starting point for euler
path is vertex on which indeg+1 == outdeg (it has more edges going out than coming in)
*/
```

4. Dynamic Programming

4.1. Optimization Techniques

4.1.1. Envelope for Convex-Hull Trick

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

struct LowerEnvelope { // min
  // change if number fits in 10^9
  typedef long long num;
  typedef long double num2;
  vector<num> M, B;
  int ptr = 0;
  bool bad(int a, int b, int c) {
    // change to insert inverted
    return (num2)(B[c] - B[a]) * (M[a] - M[b]) < (num2)(B[b] - B[a]) * (M[a] - M[c]);
  }
  void add(num m, num b) {
    if(!M.empty() && M.back() == m && B.back() <= b) // change for upper, care with float
      return;

    M.push_back(m);
    B.push_back(b);
    while (M.size() >= 3 && bad(M.size()-3, M.size()-2, M.size()-1)) {
      M.erase(M.end() - 2);
      B.erase(B.end() - 2);
    }
  }
  num eval(int i, num x) { return M[i] * x + B[i]; }
  num next(num x) {
    ptr = min(ptr, (int)M.size() - 1);
    while (ptr < M.size() - 1 && eval(ptr + 1, x) < eval(ptr, x)) // change for upper
      ptr++;
    return eval(ptr, x);
  }

  num best(num x){
    int sz = M.size();
    assert(sz > 0);

    int L = 0, R = sz-1;
```

4.1.2. Dynamic Envelope for Convex-Hull Trick

```cpp
      int mid = (L+R)/2;
      if(eval(mid+1, x) >= eval(mid, x)) // change for upper
        R = mid;
      else L = mid+1;
    }

    return eval(L, x);
  }
};
/////////////// DYNAMIC ENVELOPE
// change if number fits in 10^9
typedef long long num;
typedef long double num2;
const num is_query = -(1LL << 62);
struct Line {
  num 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;
    num x = rhs.m;
    return b - s->b > (s->m - m) * x; // modify for upper
  }
};
struct DynamicEnvelope : public multiset<Line> { // will maintain lower envelope by default
  // dont change things for dynamic envelope except for upper/lower comparator above
  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 add(num m, num 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));
  }
  num query(num x) {
    auto l = *lower_bound((Line){x, is_query});
    return l.m * x + l.b;
  }
};
```

4.1.3. Knuth Optimization

```
/*
dp[i][j] = min {dp[i][k] + dp[k][j]} + C[i][j]

minimizing condition
quadrangle inequality:
C[a][c] + C[b][d] <= C[a][d] + C[b][c]

monotonicity:
C[b][c] <= C[a][d]
```

```

sufficient minimizing condition
A[i][j-1] <= A[i][j] <= A[i+1][j]

*/
for (int s = 0; s <= k; s++)          // s - length(size) of substring
  for (int l = 0; l + s <= k; l++) { // l - left point
    int r = l + s;                    // r - right point
    if (s < 2) {
      res[l][r] = 0; // DP base - nothing to break
      mid[l][r] = l; // mid is equal to left border
      continue;
    }
    int mleft = mid[l][r - 1]; // Knuth's trick: getting bounds on m
    int mright = mid[l + 1][r];
    res[l][r] = 1000000000000000000LL;
    for (int m = mleft; m <= mright; m++) { // iterating for m in the bounds
    only
      int64 tres = res[l][m] + res[m][r] + (x[r] - x[l]);
      if (res[l][r] > tres) { // relax current solution
        res[l][r] = tres;
        mid[l][r] = m;
      }
    }
  }
int64 answer = res[0][k];
```

4.1.4. Divide and Conquer Optimization

```
/*
dp[i][j] = min{dp[i-1][k] + C[k][j]} *note that it can be C[k+1][j]*

sufficient minimizing condition
A[i][j] <= A[i][j+1]

cost condition
quadrangle inequalities
*/
void fill(int g, int l1, int l2, int p1, int p2) {
  if (l1 > l2) return;
  int lm = l1 + l2 >> 1;
  // compute dp[g][lm]
  int pm = -1;
  dp[g][lm] = infinity;
  for (int k = p1; k <= p2; k++) {
    ll new_cost = dp[g - 1][k] + cost(k, lm);
    if (dp[g][lm] > new_cost) {
      dp[g][lm] = new_cost;
      pm = k;
    }
  }
  // calculate both sides of lm
  fill(g, l1, lm - 1, p1, pm);
  fill(g, lm + 1, l2, pm, p2);
}

/*
 * if A[i-1][j] <= A[i][j] <= A[i][j+1] is true
 * then it can be solved in O(n^2 + nk)
 * for 1..K, N..1 compute dp[i][j] testing i \in [P[i-1][j], P[i][j+1]]
 */
```

5. Number Theory

5.1. Euclidean Modular Inverse

```
long long mul_inv(long long a, long long b) {
  long long b0 = b, t, q;
  long long x0 = 0, x1 = 1;
  if (b == 1) return 1;
  while (a > 1) {
    q = a / b;
    t = b, b = a % b, a = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
  }
  if (x1 < 0) x1 += b0;
  return x1;
}
```

5.2. Garner CRT

```
int x[];
int garner(int a[], int p[]) {
  int res = 0;
  int mult = 1;
  for (int i = 0; i < k; ++i) {
    x[i] = a[i];
    for (int j = 0; j < i; ++j) {
      x[i] = r[j][i] * (x[i] - x[j]);
      x[i] = x[i] % p[i];
      if (x[i] < 0) x[i] += p[i];
    }
    res += mult * x[i];
    mult *= p[i];
  }
  return res;
}
```

5.3. Fraction Comparison

```
// O(logn) til 10^18
bool comp(ll a, ll b, ll c, ll d) {
  if (a / b != c / d) return a / b < c / d;
  a %= b, c %= d;
  if (c == 0) return 0;
  if (a == 0) return 1;
  return comp(d, c, b, a);
}
```

5.4. Extended Euclidean

```
// (x, y) are found for ax+by = gcd(a,b)

// once (x, y) is found, solutions for ax+by = c
// will be in the form:
// ( (c*x + k*b)i / gcd(a, b), (c*y - k*a) / gcd(a, b) )
// or (x0 + k*b/g), (y0 - k*a/g) where x0 = xg*(c/g), y0 = yg*(c/g)

int _euclid(int a, int b, int & x, int & y){
  if(a == 0){
    x = 0, y = 1;
    return b;
  }
  int x1, y1;
```

```cpp
  int g = _euclid(b%a, a, x1, y1);
  x = y1 - b/a*x1;
  y = x1;
  return g;
}

// guarantees that returned gcd is positive
int euclid(int a, int b, int & x, int & y){
  int g = _euclid(a, b, x, y);
  if(g < 0) g = -g, x = -x, y = -y;
  return g;
}


// MORE OF
int gcd(int a, int b, int &x, int &y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd(b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}

void shift_solution (int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions (int a, int b, int c, int minx, int maxx, int miny, int maxy) {
    int x, y, g;
    if (! find_any_solution (a, b, c, x, y, g))
        return 0;
    a /= g;  b /= g;

    int sign_a = a>0 ? +1 : -1;
    int sign_b = b>0 ? +1 : -1;

    shift_solution (x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution (x, y, a, b, sign_b);
    if (x > maxx)
        return 0;
    int lx1 = x;

    shift_solution (x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution (x, y, a, b, -sign_b);
    int rx1 = x;

    shift_solution (x, y, a, b, - (miny - y) / a);
    if (y < miny)
        shift_solution (x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;

    shift_solution (x, y, a, b, - (maxy - y) / a);
    if (y > maxy)
        shift_solution (x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2)
        swap (lx2, rx2);
    int lx = max (lx1, lx2);
    int rx = min (rx1, rx2);

    return (rx - lx) / abs(b) + 1;
}
```

5.5. Miller-Rabin Primality Test

```cpp
bool isPrime(ll x) {
  if (x < 2) return false;
  if (x != 2 && x % 2 == 0) return false;
  ll p = x - 1;
  while (p % 2 == 0) p /= 2;
  for (int i = 0; i < 4; i++) {
    ll a = rand() % (x - 1) + (x - 1); // does this need to be long long or int?
    a %= (x - 1);
    a++;
    ll b = p;
    ll m = power(a, b, x);
    while (b != x - 1 && m != 1 && m != x - 1) {
      m = mul(m, m, x);
      b *= 2;
    }
    if (m != x - 1 && b % 2 == 0) { return false; }
  }
  return true;
}
```

6. Strings

6.1. KMP

```cpp
void KMP(string a, string b) {
  int n = b.size();
  int m = a.size();
  int f[m], kmp[n];
  f[0] = 0;
  for (int i = 1, k = -1; i < m; i++) {
    while (k >= 0 && a[k + 1] != a[i]) k = f[k] - 1;
    if (a[k + 1] == a[i]) k++;
    f[i] = k + 1;
  }
  for (int i = 0, k = -1; i < n; i++) {
    if (k == m - 1) k = f[k] - 1;
    while (k >= 0 && a[k + 1] != b[i]) k = f[k] - 1;
    if (a[k + 1] == b[i]) k++;
```

```
    kmp[i] = k + 1;
    if (kmp[i] == m) printf("%d\n", i);
  }
}
int main() {
  string s, t;
  cin >> s >> t;
  KMP(t, s);
}
```

6.2. Z-Algorithm

```
#include <bits/stdc++.h>
using namespace std;
vector<int> Z(string s) {
  int n = s.size();
  vector<int> z(n);
  // [L, R)
  for (int i = 1, L = 0, R = 0; i < n; i++) {
    if (i < R) z[i] = min(z[i - L], R - i);
    while (i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > R) L = i, R = i + z[i];
  }
  return z;
}
```

6.3. Power

```
// Given an string s, finds the smallest k such that s = x^k
const int len = f[n - 1];
const int period = len == 0 ? 1 : n % (n - len) ? 1 : n / (n - len);
printf("%d\n", period);
```

6.4. Suffix Array

```
char s[MAX_N];
bool suffixCmp(int i, int j) {
  if (rnk[i] != rnk[j]) return rnk[i] < rnk[j];
  i += block, j += block;
  return (i < n && j < n) ? rnk[i] < rnk[j] : i > j;
}
void build() {
  for (int i = 0; i < n; i++) sa[i] = i, rnk[i] = s[i], tmp[i] = 0;
  block = 1;
  while (tmp[n - 1] != n - 1) {
    sort(sa, sa + n, suffixCmp);
    tmp[0] = 0;
    for (int i = 0; i < n - 1; i++) tmp[i + 1] = tmp[i] + suffixCmp(sa[i],
      sa[i + 1]);
    for (int i = 0; i < n; i++) rnk[sa[i]] = tmp[i];
    block *= 2;
  }
}
void buildLCP() {
  for (int i = 0; i < n; i++) rnk[sa[i]] = i, lcp[i] = 0;
  int last = 0; // last LCP
  for (int i = 0; i < n; i++, last = max(lcp[rnk[i - 1]] - 1, 0)) {
    if (rnk[i] == n - 1) continue;
    int j = sa[rnk[i] + 1]; // next suffix pos in suffix array
    while (i + last < n && j + last < n && s[i + last] == s[j + last])
      last++;
    lcp[rnk[i]] = last;
  }
}
void solve() {
  scanf(" %s", s);
  n = strlen(s);
  build();
  for (int i = 0; i < n; i++) printf("%d\n", sa[i]);
}
```

```
int lcp[MAX_N];
char s[MAX_N];
bool suffixCmp(int i, int j) {
  if (rnk[i] != rnk[j]) return rnk[i] < rnk[j];
  i += block, j += block;
  if (i >= n) i -= n;
  if (j >= n) j -= n;
  return rnk[i] < rnk[j];
}
void suffixSort(int h) {
  for (int i = 0; i < n; i++) {
    aux[i] = sa[i] - block;
    if (aux[i] < 0) aux[i] += n;
  }
  for (int i = 0; i < h; i++) tmp[i] = 0;
  for (int i = 0; i < n; i++) tmp[rnk[aux[i]]]++;
  for (int i = 0; i < h - 1; i++) tmp[i + 1] += tmp[i];
  for (int i = n - 1; i >= 0; i--) sa[--tmp[rnk[aux[i]]]] = aux[i];
  tmp[0] = 0;
  for (int i = 0; i < n - 1; i++) tmp[i + 1] = tmp[i] + suffixCmp(sa[i],
    sa[i + 1]);
  for (int i = 0; i < n; i++) rnk[sa[i]] = tmp[i];
}
void build() {
  sa = suf;
  n++; // consider additional '\0' character
  for (int i = 0; i < n; i++) sa[i] = i, rnk[i] = s[i], tmp[i] = 0;
  block = 0;
  suffixSort(256);
  for (block = 1; tmp[n - 1] != n - 1; block *= 2) suffixSort(tmp[n - 1] +
    1);
  n--;
  sa = suf + 1;
}
void solve() {
  scanf(" %s", s);
  n = strlen(s);
  build();
  for (int i = 0; i < n; i++) printf("%d\n", sa[i]);
}
```

6.5. Aho-Corasick

```
vector<bool> aho_corasick(vector<string> keywords, string text) {
  // Compute the maximum possible number of states
  int max_states = 1;
  for (const string& keyword : keywords) max_states += keyword.size();
  int free_state = 1;
  // Build the success(goto) / failure / output functions
  map<char, int>* transitions = new map<char, int>[max_states];
  vector<int>* output = new vector<int>[max_states];
  int failure[max_states];
  for (int i = 0, sz = keywords.size(); i < sz; ++i) {
    const string& keyword = keywords[i];
```

```
    int state = 0;
    for (const char ch : keyword) {
      int& next_state = transitions[state][ch];
      if (next_state == 0) next_state = free_state++;
      state = next_state;
    }
    output[state].push_back(i);
  }
  queue<int> q;
  failure[0] = -1000000000;
  for (const auto x : transitions[0]) {
    const int v = x.second;
    q.push(v);
    failure[v] = 0;
  }
  while (!q.empty()) {
    const int u = q.front();
    q.pop();
    for (const auto x : transitions[u]) {
      const char ch = x.first;
      const int v = x.second;
      int f = failure[u];
      while (f != 0 && transitions[f].count(ch) == 0) f = failure[f];
      if (transitions[f].count(ch) == 0)
        failure[v] = 0;
      else
        failure[v] = transitions[f][ch];
      copy(output[failure[v]].begin(), output[failure[v]].end(), back_inserter(output[v]));
      q.push(v);
    }
  }
  // Find out which keywords the text contains
  vector<bool> ans(keywords.size());
  int state = 0;
  for (const char ch : text) {
    while (state != 0 && transitions[state].count(ch) == 0) state = failure[state];
    if (transitions[state].count(ch)) state = transitions[state][ch];
    for (int keyword_index : output[state]) ans[keyword_index] = true;
  }
  return ans;
}
```

6.6. Suffix Automaton

```
#include <bits/stdc++.h>
using namespace std;
inline int chr(char c) { return c - 'a'; }
struct State {
  int len, link;
  vector<int> t;
  State() {
    len = 0, link = -1;
    t.assign(26, -1);
  }
};
int last, cur, cnt;
vector<State> st;
void append(char c) {
  cur = ++cnt;
  st.push_back(State());
  st[cur].len = st[last].len + 1;
  int ptr = last;
  for (; ptr != -1 && st[ptr].t[chr(c)] == -1; ptr = st[ptr].link) st[ptr].t[chr(c)] = cur;
  if (ptr == -1) {
    st[cur].link = 0;
    return;
  }
  // complicated case
  int q = st[ptr].t[chr(c)];
  if (st[ptr].len + 1 == st[q].len) {
    st[cur].link = q;
  } else {
    // we have to break endpos class q
    int clone = ++cnt;
    st.emplace_back(st[q]);
    st[clone].len = st[ptr].len + 1;
    for (; ptr != -1 && st[ptr].t[chr(c)] == q; ptr = st[ptr].link) st[ptr].t[chr(c)] = clone;
    st[q].link = clone;
    st[cur].link = clone;
  }
}
void build(char* s, int n) {
  last = cur = cnt = 0;
  st.push_back(State());
  for (int i = 0; i < n; i++) {
    append(s[i]);
    last = cur;
  }
}
char str[100001], str2[100001];
int main() {
  scanf(" %s", str);
  int n = strlen(str);
  build(str, n);
  scanf(" %s", str2);
  int m = strlen(str2);
  int res = 0;
  int p = 0;
  for (; res < m && p != -1; p = st[p].t[chr(str2[res])]) res++;
  printf("%d\n", res);
}
```

6.7. Manacher

```

// online manacher
template <int delta> struct ManacherBase {
  private:
  static const int maxn = 1e5 + 1;
  int r[maxn];
  char s[maxn];
  int mid, n, i;
  public:
  ManacherBase() : mid(0), i(0), n(1) {
    memset(r, -1, sizeof(int) * maxn);
    s[0] = '$';
    r[0] = 0;
  }
  int get(int pos) {
    pos++;
    if (pos <= mid)
      return r[pos];
    else
      return min(r[mid - (pos - mid)], n - pos - 1);
```

```
  }
  void addLetter(char c) {
    s[n] = s[n + 1] = c;
    while (s[i - r[i] - 1 + delta] != s[i + r[i] + 1]) r[++i] = get(i - 1);
    r[mid = i]++, n++;
  }
  int maxPal() { return (n - mid - 1) * 2 + 1 - delta; }
};
struct Manacher {
  private:
  ManacherBase<1> manacherEven;
  ManacherBase<0> manacherOdd;

  public:
  void addLetter(char c) {
    manacherEven.addLetter(c);
    manacherOdd.addLetter(c);
  }
  int maxPal() { return max(manacherEven.maxPal(), manacherOdd.maxPal()); }
  int getRad(int type, int pos) {
    if (type)
      return manacherOdd.get(pos);
    else
      return manacherEven.get(pos);
  }
};
// short offline Manacher (copied)
vector<vector<int> > p(2, vector<int>(n, 0));
for (int z = 0, l = 0, r = 0; z < 2; z++, l = 0, r = 0)
  for (int i = 0; i < n; i++) {
    if (i < r) p[z][i] = min(r - i + !z, p[z][l + r - i + !z]);
    int L = i - p[z][i], R = i + p[z][i] - !z;
    while (L - 1 >= 0 && R + 1 < n && s[L - 1] == s[R + 1]) p[z][i]++, L--,
    R++;
    if (R > r) l = L, r = R;
  }
```

6.8. Palindromic Tree

```
int node_cnt;
struct Node{
  int cnt;
  int pos, len;
  Node * suf;
  map<char, Node*> t;

  int hs;
  vector<Node*> sg;
} nodes[N], * last, * r0, *r1;

Node * make(){
  return &nodes[node_cnt++];
}

void initPal(){
  r1 = make();
  r1->suf = r1;
  r1->len = -1;

  r0 = make();
  r0->suf = r1;
  r0->len = 0;
  r1->sg.pb(r0);

  last = r0;
}

void add(char c, int pos){
  Node * cur = last;
  while(1){
    int len = cur->len;
    if(pos-len-1 >= 0 && s[pos-len-1] == s[pos])
      break;
    cur = cur->suf;
  }

  // check if node already exists
  if(cur->t.count(c)){
    last = cur->t[c];
    last->cnt++;
    return;
  }

  last = make();
  last->len = cur->len+2;
  last->pos = pos;
  last->cnt++;
  cur->t[c] = last;

  // handle len==1 case
  if(last->len == 1){
    last->suf = r0;
    return;
  }

  // get suffix link
  while(1){
    cur = cur->suf;
    int len = cur->len;
    if(pos-len-1 >= 0 && s[pos-len-1] == s[pos])
      break;
  }

  last->suf = cur->t[c];
}

int propagate(){
  int tot = 0;
  for(int i = node_cnt-1; i >= 0; i--){
    if(nodes[i].len > 0){
      nodes[i].suf->cnt += nodes[i].cnt;
      tot += nodes[i].cnt;
    }
  }

  return tot;
}
```

7. Flows and Matching

7.1. SPFA Min-cost Max-flow

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


struct Edge {
  int next, cost, flow;
```

```cpp
};

struct Mincost{
  const int oo = 1e9;

  int s, t;
  int n;
  vector<vector<int>> adj;
  vector<Edge> e;
  vector<int> p, in_queue, pe, dist;

  Mincost(int n) : n(n), adj(n), p(n), in_queue(n), pe(n), dist(n){
    s = n-1;
    t = s-1;
  }

  void add_edge(int u, int v, int flow, int cost) {
    adj[u].push_back(e.size());
    e.push_back({v, cost, flow});
    adj[v].push_back(e.size());
    e.push_back({u, -cost, 0});
  }

  bool augment() {
    for (int i = 0; i < n; i++) p[i] = -1, dist[i] = oo;
    queue<int> q;
    q.push(s);
    in_queue[s] = 1;
    dist[s] = 0;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      in_queue[u] = 0;
      for (int k : adj[u]) {
        Edge& ed = e[k];
        if (ed.flow > 0 && dist[u]+ed.cost < dist[ed.next]) {
          dist[ed.next] = dist[u]+ed.cost;
          p[ed.next] = u;
          pe[ed.next] = k;
          if (!in_queue[ed.next]) {
            in_queue[ed.next] = 1;
            q.push(ed.next);
          }
        }
      }
    }

    return p[t] != -1;
  }
  pair<int, int> mincost() {
    int maxflow = 0, mincost = 0;
    while (augment()) {
      int cf = oo;
      for (int v = t; v != s; v = p[v]) { cf = min(cf, e[pe[v]].flow); }
      maxflow += cf;
      for (int v = t; v != s; v = p[v]) {
        e[pe[v]].flow -= cf;
        e[pe[v] ^ 1].flow += cf;
        mincost += cf * e[pe[v]].cost;
      }
    }
    return {maxflow, mincost};
  }
};
```

7.2. Potentials Min-cost Max-flow

```cpp
//////////////////////////////////////////////////////// Note: 0-indexed vertices
////////////////////////////////////////////////////////
struct Edge {
  int next;
  long long capacity;
  long long cost;
  Edge(int next, long long capacity, long long cost) : next(next),
    capacity(capacity), cost(cost) {}
};
struct State {
  int pos;
  long long cost;
};
bool operator<(const State lhs, const State rhs) { return lhs.cost >
    rhs.cost; }
struct MinCostFlow {
  int n;
  vector<vector<Edge> > adj;
  vector<vector<int> > rev;
  MinCostFlow(const int n) : n(n), adj(n), rev(n) {}
  void add_arc(const int u, const int v, const long long capacity, const
    long long cost) {
    if (u != v && capacity != 0) {
      rev[u].push_back(adj[v].size()), rev[v].push_back(adj[u].size());
      adj[u].emplace_back(v, capacity, cost), adj[v].emplace_back(u, 0,
    -cost);
    }
  }
  void add_edge(const int u, const int v, const long long capacity, const
    long long cost) {
    assert(cost >= 0);
    if (u != v && capacity != 0) {
      rev[u].push_back(adj[v].size()), rev[v].push_back(adj[u].size());
      adj[u].emplace_back(v, capacity, cost), adj[v].emplace_back(u,
    capacity, cost);
    }
  }
  pair<long long, long long> solve(const int s, const int t) {
    // Perform the initial cost transformation
    long long pot[n];
    memset(pot, 0, sizeof pot);
    for (;;) {
      bool changed = false;
      for (int u = 0; u < n; ++u)
        for (const Edge e : adj[u])
          if (e.capacity > 0 && pot[u] + e.cost < pot[e.next]) changed =
    true, pot[e.next] = pot[u] + e.cost;
      if (!changed) break;
    }
    // Compute the min-cost flow using Dijkstra
    long long total_cost = 0;
    long long total_flow = 0;
    for (;;) {
      bool visited[n];
      memset(visited, 0, sizeof visited);
      int parent[n];
      memset(parent, -1, sizeof parent);
      long long dist[n];
      memset(dist, 0x3f, sizeof dist);
      priority_queue<State> q;
      parent[s] = -1;
      dist[s] = 0;
      q.push({s, 0});
```

```cpp
      vector<int> vec;
      while (!q.empty()) {
        const State su = q.top();
        q.pop();
        if (visited[su.pos]) continue;
        visited[su.pos] = true;
        vec.push_back(su.pos);
        if (su.pos == t) break;
        for (int i = 0, sz = adj[su.pos].size(); i < sz; ++i) {
          const Edge e = adj[su.pos][i];
          const long long edge_adjusted_cost = e.cost + pot[su.pos] - pot[e.next];
          if (e.capacity > 0 && su.cost + edge_adjusted_cost < dist[e.next]) {
            parent[e.next] = rev[su.pos][i];
            dist[e.next] = su.cost + edge_adjusted_cost;
            q.push({e.next, su.cost + edge_adjusted_cost});
          }
        }
      }
      if (parent[t] == -1) break;
      long long bottleneck = std::numeric_limits<long long>::max();
      int v = t;
      while (v != s) {
        int idx = parent[v];
        const int u = adj[v][idx].next;
        const int xdi = rev[v][idx];
        bottleneck = min(bottleneck, adj[u][xdi].capacity);
        v = u;
      }
      total_flow += bottleneck;
      total_cost += bottleneck * (dist[t] - pot[s] + pot[t]);
      v = t;
      while (v != s) {
        int idx = parent[v];
        const int u = adj[v][idx].next;
        const int xdi = rev[v][idx];
        adj[u][xdi].capacity -= bottleneck;
        adj[v][idx].capacity += bottleneck;
        // total_cost += bottleneck * adj[u][xdi].cost;
        v = u;
      }
      reverse(vec.begin(), vec.end());
      for (const int x : vec) {
        pot[x] += dist[x];
        pot[x] = min(pot[x], 0x3f3f3f3f3f3f3f3fLL);
      }
    }
    return make_pair(total_flow, total_cost);
  }
};
```

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

struct Edge {
  int next, cost, flow;
};

struct Mincost{
  const int oo = 1e9;
  typedef pair<int, int> ii;
  struct State{
    int dist, vert;
    bool operator<(const State & rhs) const {
      return dist > rhs.dist;
    }
  };

  int s, t;
  int n;
  vector<vector<int>> adj;
  vector<Edge> e;
  vector<int> p, pot, pe, dist;

  Mincost(int n) : n(n), adj(n), p(n), pot(n, oo), pe(n), dist(n){
    s = n-1;
    t = s-1;
  }

  void add_edge(int u, int v, int flow, int cost) {
    adj[u].push_back(e.size());
    e.push_back({v, cost, flow});
    adj[v].push_back(e.size());
    e.push_back({u, -cost, 0});
  }

  void update(){
    for(int i = 0; i < n; i++) pot[i] = min(pot[i]+dist[i], oo);
  }

  bool augment() {
    int u, d;
    for (int i = 0; i < n; i++) p[i] = -1, dist[i] = oo;
    priority_queue<State> pq;
    pq.push({0, s});
    dist[s] = 0;
    while(!pq.empty()){
      State st = pq.top(); pq.pop();
      u = st.vert, d = st.dist;
      if(d != dist[u]) continue;
      for(int k : adj[u]){
        Edge & ed = e[k];
        int nd = d+pot[u]-pot[ed.next]+ed.cost;
        if(ed.flow > 0 && nd < dist[ed.next]){
          dist[ed.next] = nd;
          p[ed.next] = u;
          pe[ed.next] = k;
          pq.push({nd, ed.next});
        }
      }
    }

    return p[t] != -1;
  }
  pair<int, int> mincost() {
    pot[s] = 0;
    int changed = 1;
    while(!(changed ^= 1))
      for(int i = 0; i < n; i++)
        if(pot[i] != oo)
          for(int k : adj[i])
            if(e[k].flow > 0 && pot[e[k].next] > pot[i]+e[k].cost)
              pot[e[k].next] = pot[i]+e[k].cost, changed=1;

    int maxflow = 0, mincost = 0;
    while (augment()) {
      int cf = oo;
      for (int v = t; v != s; v = p[v]) { cf = min(cf, e[pe[v]].flow); }
```

```
      maxflow += cf;
      for (int v = t; v != s; v = p[v]) {
        e[pe[v]].flow -= cf;
        e[pe[v] ^ 1].flow += cf;
        mincost += cf * e[pe[v]].cost;
      }
      update();
    }
    return {maxflow, mincost};
  }
};
```

7.3. Mincost Circulation

```
////////////////////////////////////////// BEGIN OF MIN-COST-CIRCULATION ////////////////////////////////////////////
// Note: 0-indexed vertices
// Complexity: O(UE*ElogV), where U = flow value -- TODO: confirm, probably better -- I think it might be U+E instead of UE
// ---------- Unit capacities, no pararallel arcs =>? O(VE*ElogV) -- TODO: confirm, it's probably better too
// TODO: prove that this "fix" cannot cause overflow
// TODO: This algorithm does not work when adding an edge with capacity greater than one and negative cost -- see comment below
// TODO: the resulting flow value is always zero... I should remove the "total_flow" variable -- not realy!
struct Edge {
    int next;
    long long capacity;
    long long cost;
    Edge(int next, long long capacity, long long cost) : next(next), capacity(capacity), cost(cost) {}
};
struct State {
    int pos;
    long long cost;
};
bool operator<(const State lhs, const State rhs) {
    return lhs.cost > rhs.cost;
}
struct MinCostCirculation {
    int n;
    vector<vector<Edge>> adj;
    vector<vector<int>> rev;
    vector<long long> price;
    long long total_flow, total_cost;
    MinCostCirculation(const int n)
        : n(n)
        , adj(n)
        , rev(n)
        , price(n)
        , total_flow(0)
        , total_cost(0)
    {
    }
    void add_arc(const int u, const int v, const long long capacity, const long long cost) {
        // Check capacity
        if (capacity == 0) return;
        assert(capacity > 0);
        // If the arc already has nonnegative reduced cost, we can just add it
        if (cost + price[u] - price[v] >= 0) {
            rev[u].push_back(adj[v].size());
            rev[v].push_back(adj[u].size());
            adj[u].emplace_back(v, capacity, cost);
            adj[v].emplace_back(u, 0, -cost);
            return;
        }
        // Otherwise, we should fix potentials (always) and update the current circulation (only if the new arc introduces a negative cycle)
        long long flow = 0;
        while (flow < capacity) {
            // Compute shortest paths from v
            int parent[n];
            memset(parent, -1, sizeof parent);
            long long dist[n];
            memset(dist, 0x3f, sizeof dist);
            const long long INF = dist[0];
            priority_queue<State> q;
            dist[v] = 0;
            q.push({v, 0});
            while (!q.empty()) {
                const State su = q.top();
                q.pop();
                if (su.cost != dist[su.pos]) continue;
                for (int i = 0, sz = adj[su.pos].size(); i < sz; ++i) {
                    const Edge& e = adj[su.pos][i];
                    const long long edge_adjusted_cost = e.cost + price[su.pos] - price[e.next];
                    const long long new_distance = su.cost + edge_adjusted_cost;
                    if (e.capacity > 0 && new_distance < dist[e.next]) {
                        parent[e.next] = rev[su.pos][i];
                        dist[e.next] = new_distance;
                        q.push({e.next, new_distance});
                    }
                }
            }
            // Save the real cost of the cycle u--v (using the shortest path v-u + the new arc)
            // -- Well-defined if u is reachable from v
            // -- Otherwise, approximately +inf (but we won't use its value anyway)
            const long long real_cycle_cost = cost + (dist[u] - price[v] + price[u]);
            // If u is unreachable from v, we can just update potentials and stop
            if (dist[u] == INF) {
                // Update potentials for vertices unreachable from v
                const long long fix = -(cost + price[u] - price[v]);
                assert(fix >= 0);
                for (int w = 0; w < n; ++w)
                    if (dist[w] == INF)
                        price[w] += fix;
                assert(cost + price[u] - price[v] == 0);
                // Stop
                break;
            }
            // Update potentials for vertices reachable from v
            if (dist[u] != INF) {
                for (int w = 0; w < n; ++w)
                    if (dist[w] != INF)
                        price[w] += dist[w];
            }
            // Update potentials for vertices unreachable from v
            long long fix = 0;
            for (int w = 0; w < n; ++w)
                if (dist[w] != INF)
```

```cpp
                fix = max(fix, dist[w]);
            for (int w = 0; w < n; ++w)
                if (dist[w] == INF)
                    price[w] += fix;
            // If the new edge does not introduce a negative cycle, we should stop (but this causes problems if capacity > 1)
            if (real_cycle_cost >= 0) {
                //assert(cost + price[u] - price[v] >= 0);
                /* TODO: The potential recalculation does not work if capacity > 1
                 * In particular, the above assert might fail in this case because we could end up
                 *   with both the original arc and the reverse arc in the residual network.
                 * And the reduced cost of the original arc becomes real_cycle_cost, which means
                 *   that it is nonnegative (and might be positive). If it is indeed positive, the
                 *   reverse arc will be negative, which is a problem.
                 * Indeed, the only way to end up with a valid potential function would be if
                 *   reduced_cost(u, v) = reduced_cost(v, u) = 0
                 *   (if any of costs is positive, the other must be negative)
                 *   TODO: try to come up with an strategy that accomplishes this
                 *         alternatively, I could try to prove that this bad situation never happens -- but I'm pretty sure that it does
                 * On the other hand, correctness is guaranteed if capacity = 1 because there are only two cases:
                 *    1. there is a negative cycle, and one unit of flow is pushed
                 *         - In this case, the reduced cost of (u, v) remains negative, because
                 *           it is set to the value of the (negative) cycle
                 *         - Consequently, the reduced cost of (v, u) is positive
                 *    2. there is no cycle, because there is no path from v to u
                 *        - This case has already been correctly handled above
                 *    3. there are only nonnegative cycles, and no flow is pushed
                 *         - In this case, the reduced cost of (u, v) remains nonnegative, because
                 *           it is set to the value of the (nonnegative) cycle.
                 * */
                break;
            }
            // Send flow along the cycle
            long long bottleneck = capacity - flow;
            int vert = u;
            while (vert != v) {
                int idx = parent[vert];
                const int prev = adj[vert][idx].next;
                const int xdi = rev[vert][idx];
                bottleneck = min(bottleneck, adj[prev][xdi].capacity);
                vert = prev;
            }
            flow += bottleneck;
            total_cost += bottleneck * real_cycle_cost;
            vert = u;
            while (vert != v) {
                int idx = parent[vert];
                const int prev = adj[vert][idx].next;
                const int xdi = rev[vert][idx];
                adj[prev][xdi].capacity -= bottleneck;
                adj[vert][idx].capacity += bottleneck;
                vert = prev;
            }
        }
        //if (flow != capacity)
            //assert(cost + price[u] - price[v] >= 0);
        //if (flow != 0)
            //assert(-cost + price[v] - price[u] >= 0);
        // And finally add the arc to the network
        rev[u].push_back(adj[v].size());
        rev[v].push_back(adj[u].size());
        adj[u].emplace_back(v, capacity - flow, cost);
        adj[v].emplace_back(u, flow, -cost);
        // Update total flow
        total_flow += flow;
    }
};
////////////////////////////////////////// END OF MIN-COST-CIRCULATION ////////////////////////////////////////////
```

7.4. Dinic's

```cpp

template <class NumT = int> struct Edge {
  int a, b;
  NumT w;
  Edge(int _a, int _b) : a(_a), b(_b), w(1) {}
  Edge(int _a, int _b, NumT _w) : a(_a), b(_b), w(_w) {}
};
template <class NumT = int> struct Flow {
  int size, source, sink;
  vector<vector<NumT> > adj;
  vector<Edge<NumT> > e;
  vector<bool> visited; // used for bfs/dfs
  vector<int> dist;     // used for layered graph / dinic's
  vector<int> used;     // edges used in dinic's blocking flow
  Flow(int sz) : size(sz) {
    adj = vector<vector<NumT> >(sz);
    source = sz-1, sink = sz-2;
  }
  void setup(const int s, const int t) {
    source = s;
    sink = t;
  }
  void add_edge(const int u, const int v, const NumT weight) {
    adj[u].push_back(e.size());
    e.push_back(Edge<NumT>(u, v, weight));
    adj[v].push_back(e.size());
    e.push_back(Edge<NumT>(v, u, 0));
  }
  bool layered_bfs() {
    dist.assign(size, -1);
    dist[source] = 0;
    vector<int> q;
    q.push_back(source);
    for (int i = 0; i < q.size(); i++) {
      if (dist[sink] != -1) break;
      int u = q[i];
      for (int x : adj[u]) {
        if (dist[e[x].b] == -1 && e[x].w > 0) {
          dist[e[x].b] = dist[u] + 1;
```

```
          q.push_back(e[x].b);
        }
      }
    }
    return dist[sink] != -1;
  }
  NumT augmenting(const int u, const NumT bottle) {
    if (!bottle) return 0;
    if (u == sink) return bottle;
    for (int& i = used[u]; i < adj[u].size(); i++) {
      int x = adj[u][i];
      if (dist[e[x].b] != dist[u] + 1) continue; // only use edges of layered graph
      NumT cf = augmenting(e[x].b, min(bottle, e[x].w));
      e[x].w -= cf;
      e[x ^ 1].w += cf;
      if (cf) return cf;
    }
    return 0;
  }
  NumT blocking_flow() {
    if (!layered_bfs()) return 0;
    used.assign(size, 0);
    NumT aug, flow = 0;
    while ((aug = augmenting(source, numeric_limits<NumT>::max()))) flow += aug;
    return flow;
  }
  NumT maxflow() {
    NumT aug, flow = 0;
    while ((aug = blocking_flow())) flow += aug;
    return flow;
  }
};
```

7.5. Kuhn Algorithm

```
vector<vector<int> > adj;
vector<int> l, r;
vector<bool> vis, cl, cr;
int augmenting(int i) {
  if (vis[i]) return 0;
  vis[i] = true;
  for (int x : adj[i]) {
    if (r[x] == -1 || augmenting(r[x])) {
      r[x] = i;
      l[i] = x;
      return 1;
    }
  }
  return 0;
}
int matching(int n, int m) {
  l.assign(n, -1);
  r.assign(m, -1);
  int flow = 0;
  for (int i = 0; i < n; i++) {
    vis.assign(n, false);
    flow += augmenting(i);
  }
  return flow;
}
void alternating(int i) {
  if (vis[i]) return;
  vis[i] = true;
  cl[i] = false;
  for (int x : adj[i]) {
    cr[x] = true;
    if (r[x] != -1) alternating(r[x]);
  }
}
void covering(int n, int m) {
  cl.assign(n, true);
  cr.assign(m, false);
  vis.assign(n, false);
  for (int i = 0; i < n; i++) {
    if (l[i] == -1) alternating(i);
  }
}
```

7.6. Hopcroft-Karp

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 50005;
// store dist to L[u_in_L] vertices in the form
// 2*dist[u]+1
int dist[MAXN];
int L[MAXN], R[MAXN];
bool visited[MAXN];
vector<int> adj[MAXN]; // store directed edges from L to R
bool bfs(int n) {
  memset(dist, 0x3f, sizeof dist);
  queue<int> q;
  for (int i = 0; i < n; i++)
    if (L[i] == -1) {
      q.push(i);
      dist[i] = 0;
    }
  bool has = false;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : adj[u]) {
      has |= R[v] == -1;
      if (R[v] != -1 && dist[u] + 1 < dist[R[v]]) {
        dist[R[v]] = dist[u] + 1;
        q.push(R[v]);
      }
    }
  }
  return has;
}
bool augment(int u) {
  if (visited[u]) return 0;
  visited[u] = 1;
  for (int v : adj[u]) {
    if (R[v] == -1 || (dist[R[v]] == dist[u] + 1 && augment(R[v]))) {
      R[v] = u;
      L[u] = v;
      return 1;
    }
  }
  return 0;
}
int matching(int n) {
  memset(L, -1, sizeof L);
  memset(R, -1, sizeof R);
```

```
  int ans = 0;
  while (bfs(n)) {
    memset(visited, 0, sizeof visited);
    for (int i = 0; i < n; i++)
      if (L[i] == -1) ans += augment(i);
  }
  return ans;
}
int main() {
  int n, m, p;
  scanf("%d%d%d", &n, &m, &p);
  while (p--) {
    int a, b;
    scanf("%d%d", &a, &b);
    --a, --b;
    adj[a].push_back(b);
  }
  printf("%d\n", matching(n));
}
```

7.7. Hungarian Algorithm

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 0x3f3f3f3f;
// potentials a[i][j] >= u[i] + v[j]
int hungarian(vector<vector<int> >& a, int n, int m) {
  vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
  for (int i = 1; i <= n; ++i) {
    p[0] = i;
    int j0 = 0;
    vector<int> minv(m + 1, INF);
    vector<char> used(m + 1, false);
    do {
      used[j0] = true;
      int i0 = p[j0], delta = INF, j1;
      for (int j = 1; j <= m; ++j)
        if (!used[j]) {
          int cur = a[i0][j] - u[i0] - v[j];
          if (cur < minv[j]) minv[j] = cur, way[j] = j0;
          if (minv[j] < delta) delta = minv[j], j1 = j;
        }
      for (int j = 0; j <= m; ++j)
        if (used[j])
          u[p[j]] += delta, v[j] -= delta;
        else
          minv[j] -= delta;
      j0 = j1;
    } while (p[j0] != 0);
    do {
      int j1 = way[j0];
      p[j0] = p[j1];
      j0 = j1;
    } while (j0);
  }
  return -v[0];
}
```

7.8. Matching in general graph

```
#include <stdio.h>
#include <string.h>
using namespace std;
#define MAX_V 223
#define MAX_E MAX_V* MAX_V
long nV, nE, Match[MAX_V];
long Last[MAX_V], Next[MAX_E], To[MAX_E];
long eI;
long q[MAX_V], Pre[MAX_V], Base[MAX_V];
bool Hash[MAX_V], Blossom[MAX_V], Path[MAX_V];
void Insert(long u, long v) {
  To[eI] = v, Next[eI] = Last[u], Last[u] = eI++;
  To[eI] = u, Next[eI] = Last[v], Last[v] = eI++;
}
long Find_Base(long u, long v) {
  memset(Path, 0, sizeof(Path));
  for (;;) {
    Path[u] = 1;
    if (Match[u] == -1) break;
    u = Base[Pre[Match[u]]];
  }
  while (Path[v] == 0) v = Base[Pre[Match[v]]];
  return v;
}
void Change_Blossom(long b, long u) {
  while (Base[u] != b) {
    long v = Match[u];
    Blossom[Base[u]] = Blossom[Base[v]] = 1;
    u = Pre[v];
    if (Base[u] != b) Pre[u] = v;
  }
}
long Contract(long u, long v) {
  memset(Blossom, 0, sizeof(Blossom));
  long b = Find_Base(Base[u], Base[v]);
  Change_Blossom(b, u);
  Change_Blossom(b, v);
  if (Base[u] != b) Pre[u] = v;
  if (Base[v] != b) Pre[v] = u;
  return b;
}
void Augment(long u) {
  while (u != -1) {
    long v = Pre[u];
    long k = Match[v];
    Match[u] = v;
    Match[v] = u;
    u = k;
  }
}
long Bfs(long p) {
  memset(Pre, -1, sizeof(Pre));
  memset(Hash, 0, sizeof(Hash));
  long i;
  for (i = 1; i <= nV; i++) Base[i] = i;
  q[1] = p, Hash[p] = 1;
  for (long head = 1, rear = 1; head <= rear; head++) {
    long u = q[head];
    for (long e = Last[u]; e != -1; e = Next[e]) {
      long v = To[e];
      if (Base[u] != Base[v] and v != Match[u]) {
        if (v == p or (Match[v] != -1 and Pre[Match[v]] != -1)) {
          long b = Contract(u, v);
          for (i = 1; i <= nV; i++)
            if (Blossom[Base[i]] == 1) {
              Base[i] = b;
              if (!Hash[i]) {
                Hash[i] = 1;
```

```
              q[++rear] = i;
            }
          }
        } else if (Pre[v] == -1) {
          Pre[v] = u;
          if (Match[v] == -1) {
            Augment(v);
            return 1;
          } else {
            q[++rear] = Match[v];
            Hash[Match[v]] = 1;
          }
        }
      }
    }
  }
  return 0;
}
long Edmonds_Blossom(void) {
  long i, Ans = 0;
  memset(Match, -1, sizeof(Match));
  for (i = 1; i <= nV; i++)
    if (Match[i] == -1) Ans += Bfs(i);
  return Ans;
}
int main(void) {
  int n;
  scanf("%d", &n);
  memset(Last, -1, sizeof(Last));
  nV = n;
  eI = 0;
  for (int u, v; scanf("%d%d", &u, &v) > 0;) {
    Insert(u, v);
    nE += 1;
  }
  int x = Edmonds_Blossom();
  printf("%d\n", x * 2);
  for (int i = 1; i <= n; ++i)
    if (Match[i] > i) printf("%d %d\n", i, Match[i]);
}
```

7.9. Global Min Cut (Stoer-Wagner)

```
/* Initialization */
int cost[n + 1][n + 1];
memset(cost, 0, sizeof cost);
while (m--) {
  int u, v, c;
  u = input.next();
  v = input.next();
  c = input.next();
  cost[u][v] = c;
  cost[v][u] = c;
}
/* Stoer-Wagner: global minimum cut in undirected graphs */
int min_cut = 1000000000;
bool added[n + 1];
int vertex_cost[n + 1];
for (int vertices_count = n; vertices_count > 1; --vertices_count) {
  memset(added, 0, sizeof(added[0]) * (vertices_count + 1));
  memset(vertex_cost, 0, sizeof(vertex_cost[0]) * (vertices_count + 1));
  int s = -1, t = -1;
  for (int i = 1; i <= vertices_count; ++i) {
    int vert = 1;
    while (added[vert]) ++vert;
    for (int j = 1; j <= vertices_count; ++j)
      if (!added[j] && vertex_cost[j] > vertex_cost[vert]) vert = j;
    if (i == vertices_count - 1)
      s = vert;
    else if (i == vertices_count) {
      t = vert;
      min_cut = min(min_cut, vertex_cost[vert]);
    }
    added[vert] = 1;
    for (int j = 1; j <= vertices_count; ++j) vertex_cost[j] +=
    cost[vert][j];
  }
  for (int i = 1; i <= vertices_count; ++i) {
    cost[s][i] += cost[t][i];
    cost[i][s] += cost[i][t];
  }
  for (int i = 1; i <= vertices_count; ++i) {
    cost[t][i] = cost[vertices_count][i];
    cost[i][t] = cost[i][vertices_count];
  }
}
printf("%d\n", min_cut);
```

7.10. Simplex

```
typedef 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) {
    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] / D[r][s];
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    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;
```

```cpp
        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];
  }
};
```

8. Geometry

8.1. 2D Geometry

8.1.1. Points, Lines and Segments

```cpp
#include <algorithm>
#include <cmath>
using namespace std;
//////////
// Basic
//////////
double PI = acos(-1);
// convert radians to degs
double r2d(double a) { return a * 180 / PI; }
// convert degs to radians
double d2r(double a) { return a * PI / 180; }
////////////
//// Points
////////////
struct Vec2 {
  double x, y;
  Vec2(double x = 0, double y = 0) : x(x), y(y) {}
  Vec2(Vec2 a, Vec2 b) {
    x = b.x - a.x;
    y = b.y - a.y;
  }
  bool operator<(Vec2 b) const {
    int c = dcmp(x, b.x);
    return c == 0 ? (dcmp(y, b.y) < 0) : c < 0;
  }
  bool operator==(Vec2 b) const { return dcmp(x, b.x) == 0 && dcmp(y, b.y) == 0; }
  double norm() { return hypot(x, y); }
  double norm_sq() { return x * x + y * y; }
  Vec2 operator+(const Vec2& b) const { return Vec2(x + b.x, y + b.y); }
  Vec2 operator-(const Vec2& b) const { return Vec2(x - b.x, y - b.y); }
  Vec2 operator*(double b) const { return Vec2(x * b, y * b); }
  Vec2 operator/(double b) const { return Vec2(x / b, y / b); }
};
double dist(Vec2 a, Vec2 b) { return hypot(a.x - b.x, a.y - b.y); }
// //////////////////
// Primitive operations over Points and Vectors
/////////////////////
// Rotate CCW with 'a' rads
Vec2 rotate(Vec2 p, double a) { return {p.x * cos(a) - p.y * sin(a), p.x * sin(a) + p.y * cos(a)}; }
double dot(Vec2 a, Vec2 b) { return a.x * b.x + a.y * b.y; }
double cross(Vec2 a, Vec2 b) { return a.x * b.y - a.y * b.x; }
// returns 1 if vectors are ccw (ob is to the left of oa)
// returns -1 if vectores are cw (ob
int ccw(Vec2 o, Vec2 a, Vec2 b) { return dcmp(cross(Vec2(o, a), Vec2(o, b)), 0); }
// Smallest angle between oa and ob
double angle(Vec2 oa, Vec2 ob) { return atan2(fabs(cross(oa, ob)), dot(oa, ob)); }
// Smallest angle AOB
double angle(Vec2 o, Vec2 a, Vec2 b) { return angle(Vec2(o, a), Vec2(o, b)); }
// Clockwise angle between oa and ob
double clockwise_angle(Vec2 oa, Vec2 ob) {
  double val = atan2(cross(oa, ob), dot(oa, ob));
  if (dcmp(val, 0) < 0) val += 2 * PI;
  return val;
}
bool between(Vec2 p, Vec2 q, Vec2 r) { return ccw(p, q, r) == 0 && dcmp(dot((p - q), (r - q))) <= 0; }
// Clockwise angle AOB
double clockwise_angle(Vec2 o, Vec2 a, Vec2 b) { return clockwise_angle(Vec2(o, a), Vec2(o, b)); }
/////////////////////
////// Lines
/////////////////////
struct Line2 {
  // (a, b) is a vector orthogonal to this line
  // (-b, a) is a vector parallel to this line
  double a, b, c; // ax + by = c
  Line2() {}
  Line2(double a, double b, double c) : a(a), b(b), c(c) {}
  Line2(Vec2 v1, Vec2 v2) {
    a = v2.y - v1.y;
    b = v1.x - v2.x;
    c = a * v1.x + b * v1.y;
  }
  // orthogonal containg point p
  Line2 orthogonal(Vec2 p) { return {-b, a, -b * p.x + a * p.y}; }
  Vec2 orthogonal_vector() { return {a, b}; }
};
bool parallel(Line2 n1, Line2 n2) { return dcmp(n1.a * n2.b, n2.a * n1.b) == 0; }
```

```
bool operator==(Line2 l1, Line2 l2) {
  return dcmp(l1.a * l2.c, l2.a * l1.c) == 0 && dcmp(l1.b * l2.c, l2.b *
     l1.c) == 0;
}
bool operator<(Line2 l1, Line2 l2) {
  if (l1 == l2) return false; // Se precisar de velocidade, cachear o angulo
  int f = dcmp(atan2(l1.b, l1.a), atan2(l2.b, l2.a));
  return f < 0 || (f == 0 && dcmp(l1.a * l2.c, l2.a * l1.c) < 0);
}
Vec2 inter(Line2 n1, Line2 n2, bool& exists) {
  double det = n1.a * n2.b - n2.a * n1.b;
  if (dcmp(det, 0) == 0) {
    exists = false;
    return {0, 0};
  } else {
    exists = true;
    double x = (n2.b * n1.c - n1.b * n2.c) / det;
    double y = (n1.a * n2.c - n2.a * n1.c) / det;
    return {x, y};
  }
}
// ////////////
// Segment
/////////////
struct Segment2 {
  Vec2 a, b;
  double left_x() { return min(a.x, b.x); }
  double right_x() { return max(a.x, b.x); }
  bool contains(Vec2 q) { return between(a, q, b); }
  double len() { return dist(a, b); }
};
bool parallel(Segment2 n1, Segment2 n2) { return parallel(Line2(n1.a, n1.b),
     Line2(n2.a, n2.b)); }
Vec2 inter(Line2 l, Segment2 s, bool& exists) {
  Line2 ls(s.a, s.b);
  Vec2 p = inter(l, ls, exists);
  exists &= s.contains(p);
  return p;
}
Vec2 inter(Segment2 s1, Segment2 s2, bool& exists) {
  Line2 l1(s1.a, s1.b);
  Line2 l2(s2.a, s2.b);
  Vec2 p = inter(l1, l2, exists);
  exists &= s1.contains(p) && s2.contains(p);
  return p;
}
// /////////
// Distances
// /////////
double dist(Line2 n, Vec2 v) { return abs(n.a * v.x + n.b * v.y - n.c) /
     sqrt(n.a * n.a + n.b * n.b); }
double dist(Line2 n1, Line2 n2) {
  if (parallel(n1, n2)) {
    double faq = n1.a / n2.a;
    return abs(n1.c - n2.c * faq) / sqrt(n1.a * n1.a + n1.b * n1.b);
  } else {
    return 0;
  }
}
double dist(Segment2 s, Vec2 p) {
  bool ex;
  Line2 l = Line2(s.a, s.b).orthogonal(p);
  Vec2 x = inter(l, s, ex);
  if (!ex)
    return min(dist(s.a, p), dist(s.b, p));
  else
    return dist(x, p);
}
double dist(Segment2 s1, Segment2 s2) {
  bool ex;
  inter(s1, s2, ex);
  if (ex) return 0;
  double d = dist(s1, s2.a);
  d = min(d, dist(s1, s2.b));
  d = min(d, dist(s2, s1.a));
  d = min(d, dist(s2, s1.b));
  return d;
}
```

8.2. 3D Geometry

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

struct Vec3d{
  double x,y,z;
  Vec3d operator+(const Vec3d & rhs) const{
    return {x+rhs.x, y+rhs.y, z+rhs.z};
  }
  Vec3d operator*(const double k) const {
    return {k*x, k*y, k*z};
  }
  Vec3d operator-(const Vec3d & rhs) const{
    return *this + rhs*-1;
  }
  Vec3d operator/(const double k) const {
    return {x/k, y/k, z/k};
  }
  double operator*(const Vec3d & rhs) const{
    return x*rhs.x+y*rhs.y+z*rhs.z;
  }
  double norm_sq() { return (*this)*(*this); }
  double norm(){ return sqrt(norm_sq()); }
};

Vec3d rotate(Vec3d p, Vec3d u /*unit vector*/, double ang){
  double dot = p*u;
  double co = cos(ang);
  double si = sin(ang);
  double x = u.x*dot*(1-co) + p.x*co + (u.y*p.z-u.z*p.y)*si;
  double y = u.y*dot*(1-co) + p.y*co + (u.z*p.x-u.x*p.z)*si;
  double z = u.z*dot*(1-co) + p.z*co + (u.x*p.y-u.y*p.x)*si;
  return {x, y, z};
}

int main(){
  Vec3d axis = {1.5, 0.5, 0};
  axis = axis/axis.norm();
  Vec3d pt = rotate({1.5, 2.5, 19}, axis, 45.*acos(-1)/180);
  cout << pt.x << " " << pt.y << " " << pt.z << endl;
}
```

8.3. Complex Geometry

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

typedef complex<double> point;
```

```cpp
#define x real()
#define y imag()
#define EX(p) p.first,p.second
#define PT(a) const point & a
#define LINE(a,b) PT(a),PT(b)
#define SEG(a,b) LINE(a,b)
#define CIRCLE(a,b) PT(a), double b

const double PI = acos(-1);
const double EPS = 1e-9;

int dcmp(double a, double b = 0){
  if(a+EPS < b) return -1;
  else if(b+EPS < a) return 1;
  return 0;
}

// basics
// returned angles are in [0, 2pi] for absolute or [-pi, pi] for relative
// arg(v) returns angle of the vector
// norm(v) returns squared norm of the vector
// abs(v) returns norm of the vector
// polar to cartesian polar(norm, arg)
// cartesian to polar point(abs(v), arg(v))
double dot(PT(a), PT(b)){ return (conj(a)*b).x; }
double cross(PT(a), PT(b)) { return (conj(a)*b).y; }
int ccw(PT(o), PT(a), PT(b)) { return dcmp(cross(a-o, b-o)); }
point rotate(PT(a), double theta) {
  return a*polar(1.0, theta);
}
point ortho(PT(v)){ return point(-v.y, v.x); }
point normalize(PT(p), double k=1.0){
  return !dcmp(abs(p)) ? point(0,0) : p/abs(p)*k;
}

bool collinear(PT(a), PT(b), PT(c)){
  return !ccw(a,b,c);
}

point proj(PT(p), PT(v)){
  return v*dot(p,v) / norm(v);
}

point reflect(PT(p), LINE(a, b)){
  return a + conj((p - a) / (b - a)) * (b - a);
}

point sig_angle(PT(a), PT(o), PT(b)){ // signed smallest angle
  return remainder(arg(a-o) - arg(b-o), 2.0 * PI);
}

point angle(PT(a), PT(o), PT(b)){ // abs smallest angle
  return abs(sig_angle(a,o,b));
}

bool between(PT(a), PT(b), PT(c)){
  return collinear(a,b,c) && dcmp(dot(a-b, c-b)) <= 0;
}

// lines
point proj(PT(p), LINE(a, b)){
  return a+proj(p-a, b-a);
}

bool collinear(LINE(a,b), LINE(p,q)){
  return collinear(a,b,p) && collinear(a,b,q);
}

bool parallel(LINE(a,b), LINE(p,q)){
  return !dcmp(cross(b-a, q-p));
}

// intersections and distances
point inter(LINE(a,b), LINE(p,q)) {
  double c1 = cross(p - a, b - a), c2 = cross(q - a, b - a);
  return (c1 * q - c2 * p) / (c1 - c2); // undefined if parallel
}

double dist_point_line(PT(p), LINE(a,b)){
  return abs(cross(b-a, p-a)) / abs(b-a);
}

double dist_point_segment(PT(p), SEG(a,b)){
  if(dot(b-a, p-b) > 0) return abs(p-b);
  if(dot(a-b, p-a) > 0) return abs(p-a);
  return dist_point_line(p,a,b);
}

double dist_segment(SEG(a,b), SEG(p,q)){
  return min({dist_point_segment(a, p, q),
      dist_point_segment(b, p, q),
      dist_point_segment(p, a, b),
      dist_point_segment(q, a, b)});
}

point closest_point_segment(PT(p), SEG(a,b)){
  if(dot(b-a, p-b) > 0) return b;
  if(dot(a-b, p-a) > 0) return a;
  return proj(p,a,b);
}

// barycentric stuff
point bary(PT(A), PT(B), PT(C), double a, double b, double c){
  return (A*a + B*b + C*c) / (a+b+c);
}
point centroid(PT(A), PT(B), PT(C)){
  return bary(A, B, C, 1, 1, 1);
}
point circumcenter(PT(A), PT(B), PT(C)){
  double a = norm(B-C), b = norm(C-A), c = norm(A-B);
  return bary(A, B, C, a*(b+c-a), b*(c+a-b), c*(a+b-c));
}
point incenter(PT(A), PT(B), PT(C)){
  return bary(A, B, C, abs(B-C), abs(A-C), abs(A-B));
}
point orthocenter(PT(A), PT(B), PT(C)){
  double a = norm(B-C), b = norm(C-A), c = norm(A-B);
  return bary(A, B, C, (a+b-c)*(c+a-b), (b+c-a)*(a+b-c), (c+a-b)*(b+c-a));
}
point excenter(PT(A), PT(B), PT(C)){
  return bary(A, B, C, -abs(B-C), abs(A-C), abs(A-B));
}

// circles
int inter_circle(CIRCLE(a, ra), CIRCLE(b, rb), point & r1, point & r2){
  double d = abs(a-b);
  if(dcmp(ra+rb, d) < 0 || dcmp(d, abs(ra-rb)) < 0) return 0;
  double ax = (ra*ra-rb*rb+d*d)/2/d;
  double h = sqrt(ra*ra-ax*ax);
  point v = normalize(b-a, ax), u = normalize(rotate(b-a, PI/2), h);
```

```
  r1 = a+v+u, r2 = a+v-u;
  return 1 + (dcmp(abs(u)) > 0);
}

int inter_line_circle(LINE(a, b), CIRCLE(O, r), point & r1, point & r2){
  point H = proj(b-a, O-a) + a; double h = abs(H-O);
  if (r < h - EPS) return 0;
  point v = normalize(b-a, sqrt(r*r - h*h));
  r1 = H + v, r2 = H - v;
  return 1 + (dcmp(abs(v)) > 0);
}

int tangent(PT(a), CIRCLE(O, r), point &r1, point &r2) {
    point v = O - a; double d = abs(v);
    if (d < r - EPS) return 0;
    double alpha = asin(r / d), L = sqrt(d*d - r*r);
    v = normalize(v, L);
    r1 = a + rotate(v, alpha), r2 = a + rotate(v, -alpha);
    return 1 + (dcmp(abs(v)) > 0);
}


// NO TEST, doesnt even compile :D
void tangent_outer(point A, double rA, point B, double rB, PP(P), PP(Q)) {
    if (rA - rB > EPS) { swap(rA, rB); swap(A, B); }
    double theta = asin((rB - rA)/abs(A - B));
    point v = rotate(B - A, theta + pi/2), u = rotate(B - A, -(theta + pi/2));
    u = normalize(u, rA);
    P.first = A + normalize(v, rA); P.second = B + normalize(v, rB);
    Q.first = A + normalize(u, rA); Q.second = B + normalize(u, rB); }
```

8.4. Polygon Intersection

```
#include "Geo2D.cpp" // ccw, line inter
// suppose poly has repeated first point
struct HalfPlane {
  Vec2 a, b;
  HalfPlane(Vec2 a, Vec2 b) : a(a), b(b) {}
  bool contains(Vec2 p) { return ccw(b, a, p) < 0; }
};
vector<Vec2> polycut(vector<Vec2> poly, HalfPlane plane) {
  int n = poly.size() - 1;
  if (n < 3) return {};
  vector<Vec2> out;
  bool inside = plane.contains(poly[0]);
  if (inside) out.push_back(poly[0]);
  Vec2 prev = poly[0];
  for (int i = 1; i <= n; ++i) {
    Vec2 p = poly[i % n];
    if (plane.contains(p)) {
      if (inside) {
        out.push_back(p);
      } else {
        bool ok;
        out.push_back(inter(Line2(plane.a, plane.b), Line2(prev, p), ok));
        out.push_back(p);
      }
      inside = true;
    } else {
      if (inside) {
        bool ok;
        out.push_back(inter(Line2(plane.a, plane.b), Line2(prev, p), ok));
      }
      inside = false;
    }
    prev = p;
  }
  return out;
}
const vector<Vec2> HUGEPOLY{{-1e10, 1e10}, {1e10, 1e10}, {1e10, -1e10}, {-1e10, -1e10}, {-1e10, 1e10}};
void polyfix(vector<Vec2>& p) {
  if (!HalfPlane(p[0], p[1]).contains(p[2])) reverse(p.begin(), p.end());
}
vector<Vec2> polyinter(vector<Vec2> p1, vector<Vec2> p2) {
  if (p1.size() < p2.size()) swap(p1, p2);
  vector<Vec2> out = p1;
  for (int i = 0; i < (int)p2.size() - 1; ++i) out = polycut(out, HalfPlane(p2[i], p2[i + 1]));
  return out;
}
```

8.5. Barycentric Stuff

```
Vec2 bary(Vec2 A, Vec2 B, Vec2 C, double a, double b, double c) { return (A * a + B * b + C * c) / (a + b + c); }
Vec2 centroid(Vec2 A, Vec2 B, Vec2 C) {
  // geometric center of mass
  return bary(A, B, C, 1, 1, 1);
}
Vec2 circumcenter(Vec2 A, Vec2 B, Vec2 C) {
  // intersection of perpendicular bisectors
  double a = (B - C).norm_sq(), b = (C - A).norm_sq(), c = (A - B).norm_sq();
  return bary(A, B, C, a * (b + c - a), b * (c + a - b), c * (a + b - c));
}
Vec2 incenter(Vec2 A, Vec2 B, Vec2 C) {
  // intersection of internal angle bisectors
  return bary(A, B, C, (B - C).norm(), (A - C).norm(), (A - B).norm());
}
Vec2 orthocenter(Vec2 A, Vec2 B, Vec2 C) {
  // intersection of altitudes
  double a = (B - C).norm_sq(), b = (C - A).norm_sq(), c = (A - B).norm_sq();
  return bary(A, B, C, (a + b - c) * (c + a - b), (b + c - a) * (a + b - c), (c + a - b) * (b + c - a));
}
Vec2 excenter(Vec2 A, Vec2 B, Vec2 C) {
  // intersection of two external angle bisectors
  double a = (B - C).norm(), b = (A - C).norm(), c = (A - B).norm();
  return bary(A, B, C, -a, b, c);
  //// NOTE: there are three excenters
  // return bary(A, B, C, a, -b, c);
  // return bary(A, B, C, a, b, -c);
}
```

8.6. Rotating Caliper

```
#include <bits/stdc++.h>
#include "GeoComplex.cpp"

// counterclockwise rotation
struct Caliper{
  point p;
  double ang; // absolute angle [0, 2pi]
  Caliper(point a, double b){
    p=a;
    ang = remainder(b, 2*PI);
```

```
    while(ang<0) ang += 2*PI;
  }
  double angle_to(point v){ // relative angle [-pi, pi] (ccw is positive)
    double a = remainder(arg(v-p) - ang, 2*PI);
    return a;
  }
  void rotate(double theta){
    ang += theta;
    while(ang<0) ang += 2*PI;
    while(ang>2*PI) ang -= 2*PI;
  }
  void move(point v) { p = v; }
  point versor(){ return polar(1.0, ang); }
  double dist(const Caliper & c){
    point v = p+versor()*100.0;
    return dist_point_line(c.p, p, v);
  }
};
```

9. Algebra

9.1. Matrix

```
  Matrix(int n, int m) {
    rows = n;
    cols = m;
    this->assign(n, vector<T>(m));
  }
};
template <typename T> Matrix<T> operator*(const Matrix<T>& a, const
    Matrix<T>& b) {
  // assert(a.cols == b.rows);
  Matrix<T> res(a.rows, b.cols);
  int mid = a.cols;
  for (int i = 0; i < a.rows; i++) {
    for (int k = 0; k < mid; k++) {
      {
        for (int j = 0; j < b.cols; j++) { res[i][j] += a[i][k] * b[k][j]; }
      }
    }
  }
  return res;
}
  // GAUSSIAN ELIMINATION
  // outputs implicit row echelon form matrix, modification ratio of
    determinant and rank
```

9.2. Gaussian Elimination

```
    int m = matrix.cols;
    int det = 1;  // modification ratio of determinant of (augmented) matrix
    int rank = 0; // rank of (augmented) matrix
    // forward elimination
    for (int j = 0; j < m; j++) {
      int p = j;
      for (int i = j + 1; i < n; i++) {
        if (dcmp(fabs(matrix[i][j]), fabs(matrix[p][j])) > 0) p = i;
      }
      // line swap
      if (p != j) {
        det = -det;
        for (int k = j; k < m; k++) swap(matrix[j][k], matrix[p][k]);
      }
      if (dcmp(matrix[j][j], 0.0) == 0) continue; // null line
      rank++;
      // change k >= 0 if lower triangle is important
      for (int i = j + 1; i < n; i++)
        for (int k = m - 1; k >= j; k--) { matrix[i][k] -= matrix[j][k] *
    matrix[i][j] / matrix[j][j]; }
    }
  }
  vector<double> gauss_backward(const Matrix<double>& matrix) {
    int n = matrix.rows;
    int m = matrix.cols - 1; // m is augmenting column index
    vector<double> res(m);
    for (int j = m - 1; j >= 0; j--) {
      assert(dcmp(matrix[j][j], 0.0) != 0);
      double t = 0.0;
      for (int k = j + 1; k < m; k++) t += matrix[j][k] * res[k];
      res[j] = (matrix[j][m] - t) / matrix[j][j];
    }
    return move(res);
  }
```

9.3. Fast Fourier Transform

```
#include <bits/stdc++.h>
using namespace std;
template <typename T> struct Complex {
  T re, im;
  Complex(T a = T(), T b = T()) : re(a), im(b) {}
  T real() const { return re; }
  void operator+=(const Complex<T>& rhs) { re += rhs.re, im += rhs.im; }
  void operator-=(const Complex<T>& rhs) { re -= rhs.re, im -= rhs.im; }
  void operator*=(const Complex<T>& rhs) {
    tie(re, im) = make_pair(re * rhs.re - im * rhs.im, re * rhs.im + im *
    rhs.re);
  }
  Complex<T> operator+(const Complex<T>& rhs) {
    Complex<T> res = *this;
    res += rhs;
    return res;
  }
  Complex<T> operator-(const Complex<T>& rhs) {
    Complex<T> res = *this;
    res -= rhs;
    return res;
  }
  Complex<T> operator*(const Complex<T>& rhs) {
    Complex<T> res = *this;
    res *= rhs;
    return res;
  }
  void operator/=(const T x) { re /= x, im /= x; }
};
typedef long long ll;
namespace NTT {
const ll MOD = 5 * (1 << 25) + 1;
const ll root = 243;
const ll root_inv = 114609789;
const ll root_sz = (1 << 25);
const ll inv2 = (MOD + 1) / 2;
};
typedef Complex<double> cd;
typedef vector<cd> vcd;
// make it bigger if needed
const int LBN = 22;
```

```
const int BN = (1 << LBN);
const double PI = acos(-1);
cd root[BN];
int rev[BN], bn;
// functions used to precompute and get kth roots for size n
cd get_root(int k, int n) { return root[k * (bn / n)]; }
void dft_roots(int n) {
  bn = n;
  root[0] = cd(cosl(2.0 * PI / bn), sinl(2.0 * PI / bn));
  for (int i = 0; i < bn; i++) { root[i] = root[i - 1] * root[0]; }
}
// function used to precompute rev for fixed size fft (n is a power of two)
void dft_rev(int n) {
  int lbn = __builtin_ctz(n);
  int h = -1;
  for (int i = 1; i < n; i++) {
    if ((i & (i - 1)) == 0) h++;
    rev[i] = rev[i ^ (1 << h)] | (1 << (lbn - h - 1));
  }
}
void dft_iter(cd* p, int n) {
  for (int L = 2; L <= n; L <<= 1) {
    double ang = PI * 2 / L;
    cd step = cd(cos(ang), sin(ang)); // root
    // for (int i = L; i < root_sz; i <<= 1) NTT HERE
    //   step = step * step % MOD;
    for (int i = 0; i < n; i += L) {
      cd w = 1;
      for (int j = 0; j < L / 2; j++) {
        cd x = p[i + j];
        cd y = p[i + j + L / 2] * w;
        p[i + j] = x + y;
        p[i + j + L / 2] = x - y;
        w *= step;
      }
    }
  }
}
void dft(vcd& p) {
  int n = p.size();
  for (int i = 0; i < n; i++)
    if (i < rev[i]) swap(p[i], p[rev[i]]);
  dft_iter(&p[0], n);
}
void idft(vcd& p) {
  int n = p.size();
  dft(p);
  reverse(p.begin() + 1, p.end());
  for (int i = 0; i < n; i++) p[i] /= n;
}
template <typename T> vcd fft(const vector<T>& a, const vector<T>& b) {
  vcd fa = a, fb = b;
  int n = max(a.size(), b.size());
  n = 1 << (32 - __builtin_clz(n) + ((n & (n - 1)) != 0));
  fa.resize(n), fb.resize(n);
  vcd res(n);
  dft_rev(n);
  dft(fa), dft(fb);
  for (int i = 0; i < n; i++) res[i] = fa[i] * fb[i];
  idft(res);
  return move(res);
}
// karatsuba-like
// only reference, not working code
vi multmod(const vi& a, const vi& b) {
  // a = a0 + sqrt(MOD) * a1
  // a = a0 + base * a1
  int base = (int)sqrtl(MOD);
  vi a0(sz(a)), a1(sz(a));
  forn(i, sz(a)) {
    a0[i] = a[i] % base;
    a1[i] = a[i] / base;
    assert(a[i] == a0[i] + base * a1[i]);
  }
  vi b0(sz(b)), b1(sz(b));
  forn(i, sz(b)) {
    b0[i] = b[i] % base;
    b1[i] = b[i] / base;
    assert(b[i] == b0[i] + base * b1[i]);
  }
  vi a01 = a0;
  forn(i, sz(a)) { addmod(a01[i], a1[i], MOD); }
  vi b01 = b0;
  forn(i, sz(b)) { addmod(b01[i], b1[i], MOD); }
  vi C = mult(a01, b01);  // 1
  vi a0b0 = mult(a0, b0); // 2
  vi a1b1 = mult(a1, b1); // 3
  vi mid = C;
  forn(i, N) {
    addmod(mid[i], -a0b0[i] + MOD, MOD);
    addmod(mid[i], -a1b1[i] + MOD, MOD);
  }
  vi res = a0b0;
  forn(i, N) { addmod(res[i], mulmod(base, mid[i], MOD), MOD); }
  base = mulmod(base, base, MOD);
  forn(i, N) { addmod(res[i], mulmod(base, a1b1[i], MOD), MOD); }
  return res;
}
```

9.4. Xor FFT

```
// For all transforms here, n = 2^k, k >= 0

// Lewin XorFFT

void transform(int x, int y) {
  if (x == y - 1) { return; }
  int l2 = (y - x) / 2;
  int z = x + l2;
  transform(x, z);
  transform(z, y);
  for (int i = x; i < z; i++) {
    int x1 = a[i];
    int x2 = a[i + l2];
    a[i] = (x1 - x2 + MOD) % MOD;
    a[i + l2] = (x1 + x2) % MOD;
  }
}
void untransform(int x, int y) {
  if (x == y - 1) { return; }
  int l2 = (y - x) / 2;
  int z = x + l2;
  for (int i = x; i < z; i++) {
    long long y1 = a[i];
    long long y2 = a[i + l2];
    a[i] = (int)(((y1 + y2) * INV2) % MOD);
    a[i + l2] = (int)(((y2 - y1 + MOD) * INV2) % MOD);
  }
  untransform(x, z);
```

```
  untransform(z, y);
}

// Fast Hadamard Transform
// T = 1/sqrt(2)[1  1] ; T^(-1) = T
//              [1 -1]
//
// T(a,b) = (a+b, a-b), T-1(a,b) = ((a+b)/2, (a-b)/2)
// For NXOR
// T(a,b) = (a-b, a+b), T-1(a,b) = ((a+b)/2, (b-a)/2)
void fht(int * p, int n, bool inverse = false){
  for(int L = 2; L <= n; L <<= 1){
    for(int i = 0; i < n; i += L){
      for(int j = 0; j < L/2; j++){
        int u = p[i+j];
        int v = p[i+j+L/2];
        p[i+j] = u+v;
        p[i+j+L/2] = u-v;
      }
    }
  }

  if(inverse){
    for(int i = 0; i < n; i++){
      p[i] /= n;
    }
  }
}

// And Closure Transform
// T = [0  1]; T^(-1) = [-1  1]
//     [1  1]           [ 1  0]
//
// T(a,b) = (b, a+b), T-1(a,b) = (b-a, a)
// For NAND
// T(a,b) = (a+b, b), T-1(a,b) = (a-b, b)
void act(int * p, int n, bool inverse= false){
  for(int L = 2; L <= n; L <<= 1){
    for(int i = 0; i < n; i += L){
      for(int j = 0; j < L/2; j++){
        int u = p[i+j];
        int v = p[i+j+L/2];
        if(inverse)
          p[i+j] = v-u, p[i+j+L/2] = u;
        else
          p[i+j] = v, p[i+j+L/2] = u+v;
      }
    }
  }
}

// Or Closure Transform
// T(a,b) = (a+b, a), T-1(a,b) = (b, a-b)
// For NOR
// T(a,b) = (a, a+b), T-1(a,b) = (a, b-a)
void oct(int * p, int n, bool inverse=false){
  for(int L = 2; L <= n; L <<= 1){
    for(int i = 0; i < n; i += L){
      for(int j = 0; j < L/2; j++){
        int u = p[i+j];
        int v = p[i+j+L/2];
        if(inverse)
          p[i+j] = v, p[i+j+L/2] = u-v;
        else
          p[i+j] = u+v, p[i+j+L/2] = u;
      }
    }
  }
}
```

10. Non-Indexed Codes

10.1. Matroid Intersection

```

#include <bits/stdc++.h>
using namespace std;
const int source = 10001;
const int sink = 10002;
const int INF = 0x3f3f3f3f;
struct Edge {
  int id;
  int u, v, color;
};
int n, m, k;
vector<Edge> edges;
vector<Edge> adj[101];
bool colored[202], I[10001], nd[10001];
int in_comp[101];
vector<int> vI;
int gn[10005];
int g[10005][10005];
int dist[10005];
int q[10005], L, R;
bool augment(int u) {
  int cur = dist[u];
  dist[u] = INF;
  if (nd[u]) {
    I[u] ^= 1;
    return 1;
  }
  for (int k = 0; k < gn[u]; k++) {
    int v = g[u][k];
    if (dist[v] == cur + 1 && augment(v)) {
      I[u] ^= 1;
      return 1;
    }
  }
  return 0;
}
void dfs(int u, int cnt) {
  in_comp[u] = cnt;
  for (Edge e : adj[u]) {
    if (in_comp[e.v] == -1 && I[e.id]) dfs(e.v, cnt);
  }
}
void connected_components() {
  int cnt = 0;
  for (int i = 0; i < n; i++) in_comp[i] = -1;
  for (int i = 0; i < n; i++)
    if (in_comp[i] == -1) dfs(i, cnt++);
}
void init() {
  for (int i = 0; i < n; i++) adj[i].clear();
  edges.clear();
  for (int i = 0; i < m; i++) I[i] = false;
}
int main() {
  int tn = 1;
```

```
  while (scanf("%d%d%d", &n, &m, &k) != EOF) {
    init();
    for (int i = 0; i < m; i++) {
      int a, b, c;
      scanf("%d%d%d", &a, &b, &c);
      --a, --b, --c;
      edges.push_back({i, a, b, c});
      adj[a].push_back({i, a, b, c});
      adj[b].push_back({i, b, a, c});
    }
    while (true) {
      // init
      vI.clear();
      for (int i = 0; i < k; i++) colored[i] = false;
      for (int i = 0; i < m; i++) {
        gn[i] = 0;
        dist[i] = INF;
        nd[i] = false;
        if (I[i]) vI.push_back(i), colored[edges[i].color] = true;
      }
      if (vI.size() == n - 1) break;
      // build bipartite border graph
      connected_components();
      int both = -1;
      for (int i = 0; i < m; i++) {
        if (!I[i]) {
          int same = 0;
          if (!colored[edges[i].color]) same++;
          if (in_comp[edges[i].u] != in_comp[edges[i].v]) same++, nd[i] =
    true;
          if (same == 2) {
            both = i;
            break;
          }
        }
      }
      if (both != -1) {
        I[both] = true;
        continue;
      }
      // build directed graph
      for (int i : vI) {
        // set temp
        colored[edges[i].color] = false;
        I[i] = false;
        connected_components();
        for (int j = 0; j < m; j++) {
          if (!colored[edges[j].color]) g[i][gn[i]++] = j;
          if (in_comp[edges[j].u] != in_comp[edges[j].v]) g[j][gn[j]++] = i;
        }
        // reset
        colored[edges[i].color] = true;
        I[i] = true;
      }
      // find augmenting sequence
      L = 0, R = 0;
      for (int i = 0; i < n; i++)
        if (!I[i] && !colored[edges[i].color]) q[R++] = i, dist[i] = 0;
      bool has = false;
      while (L != R) {
        int u = q[L++];
        if (nd[u]) {
          has = true;
          break;
        }
        for (int k = 0; k < gn[u]; k++) {
          int v = g[u][k];
          if (dist[v] > dist[u] + 1) {
            dist[v] = dist[u] + 1;
            q[R++] = v;
          }
        }
      }
      if (!has) break;
      for (int i = 0; i < n; i++)
        if (!I[i] && !colored[edges[i].color]) augment(i);
    }
    // get ans
    int ans = 0;
    for (int i = 0; i < m; i++) ans += I[i];
    printf("Instancia %d\n", tn++);
    if (ans == n - 1)
      puts("sim");
    else
      puts("nao");
    puts("");
  }
}
```

10.2. Burunduk1 (bridges)

```
#include <bits/stdc++.h>
using namespace std;
const int N = 300005;
typedef vector<vector<ii> > Graph;
struct Edge {
  int a, b, l, r, id;
};
vector<int> st;
int ans[2 * N];
int color[N];
int mark[N], subsz[N], last[N], dist[N];
// bcc stuff
int comps;
int tempo, lowlink[N], visited[N];
void bcc(const Graph& g, int u, int p) {
  visited[u] = lowlink[u] = ++tempo;
  // printf("visiting %d\n", u);
  st.push_back(u);
  for (const ii& e : g[u]) {
    int v = e.first;
    if (v == p) {
      p = -1;
      continue;
    }
    if (!visited[v]) {
      bcc(g, v, u);
      lowlink[u] = min(lowlink[u], lowlink[v]);
      // check for bridges
      if (lowlink[v] > visited[u]) {
        vector<int> aux;
        int x;
        do {
          x = st.back();
          color[x] = comps;
          st.pop_back();
        } while (x != v);
        comps++;
      }
```

```
    } else
      lowlink[u] = min(lowlink[u], lowlink[v]);
  }
}
void add(Graph& g, int u, int v, int w) {
  if (u == v) return;
  g[u].push_back({v, w});
  g[v].push_back({u, w});
}
int compress(const Graph& g, Graph& h) {
  int gn = g.size();
  h = Graph(comps);
  int sum = 0;
  for (int i = 0; i < gn; i++) {
    for (const ii& e : g[i]) {
      if (color[e.first] != color[i]) {
    h[color[i]].push_back({color[e.first], e.second}), sum += e.second; }
    }
  }
  // printf("sum %d\n", sum);
  return sum / 2;
}
void reduce(const Graph& g, Graph& h, int u, int p, int& w) {
  if (mark[u]) {
    last[u] = u;
    dist[u] = 0;
    color[u] = h.size();
    h.emplace_back();
  } else {
    last[u] = -1;
  }
  visited[u] = 1;
  for (const ii& e : g[u]) {
    int v = e.first;
    if (v == p) continue;
    reduce(g, h, v, u, w);
    if (last[v] != -1) {
      if (last[u] != -1) {
        if (last[u] != u) {
          color[u] = h.size();
          h.emplace_back();
          add(h, color[u], color[last[u]], dist[u]);
          w += dist[u];
          last[u] = u;
          dist[u] = 0;
        }
        add(h, color[u], color[last[v]], dist[v] + e.second);
        w += dist[v] + e.second;
      } else {
        last[u] = last[v];
        dist[u] = dist[v] + e.second;
      }
    }
  }
}
void go(Graph g, const vector<Edge>& q, int l, int r, int gn, int hidden) {
  if (r < l) return;
  // puts("=====");
  // printf("on l = %d, r = %d, gn = %d, hidden = %d\n", l,r,gn,hidden);
  vector<Edge> q2;
  // add edges
  for (const Edge& e : q) {
    if (e.l <= l && r < e.r) {
      add(g, e.a, e.b, 1);
      // printf("adding edge %d (%d, %d)\n", e.id, e.a, e.b);
    } else if (e.r >= l && e.l <= r)
      q2.push_back(e);
  }
  // bcc
  tempo = comps = 0;
  for (int i = 0; i < gn; i++) visited[i] = 0;
  assert(!g.empty());
  for (int i = 0; i < gn; i++)
    if (!visited[i]) {
      bcc(g, i, -1);
      if (!st.empty()) {
        for (int u : st) color[u] = comps;
        comps++;
        st.clear();
      }
    }
  // compress bcc
  Graph h;
  int w1 = compress(g, h);
  gn = h.size();
  // mark interesting vertices
  for (int i = 0; i < gn; i++) mark[i] = 0;
  for (Edge& e : q2) {
    e.a = color[e.a];
    e.b = color[e.b];
    mark[e.a] = 1;
    mark[e.b] = 1;
  }
  // reduce bcc tree
  Graph g2;
  int w2 = 0;
  for (int i = 0; i < gn; i++) visited[i] = 0;
  for (int i = 0; i < gn; i++) {
    if (!visited[i]) reduce(h, g2, i, -1, w2);
  }
  int cn = g2.size();
  for (Edge& e : q2) e.a = color[e.a], e.b = color[e.b];
  // printf("w1: %d, w2: %d\n", w1, w2);
  // recur
  if (l != r) {
    hidden += w1 - w2;
    int mid = (l + r) / 2;
    go(g2, q2, l, mid, cn, hidden);
    go(g2, q2, mid + 1, r, cn, hidden);
  } else {
    // printf("ans[%d] = %d + %d\n", l, w1, hidden);
    ans[l] = w1 + hidden;
  }
}
char s[10];
int main() {
  int n = RI;
  int k = RI;
  map<ii, int> cc;
  vector<Edge> q;
  int f = 0;
  int tempo = 0;
  for (int i = 0; i < k; i++) {
    scanf(" %s", s);
    char c = s[0];
    int a = RI - 1;
    int b = RI - 1;
    if (a > b) swap(a, b);
    if (c == 'A') {
      cc[ii(a, b)] = f;
```

```
      q.push_back({a, b, tempo++, 10000000, f});
      f++;
    } else {
      q[cc[ii(a, b)]].r = tempo++;
    }
  }
  go(Graph(n), q, 0, tempo - 1, n, 0);
  for (int i = 0; i < tempo; i++) printf("%d\n", ans[i]);
}
```

10.3. Burunduk1 Incremental (bridges)

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100005;
int r[N];
int x[2 * N], y[2 * N];
bool mark[2 * N];
vector<int> adj[N];
void init(int n) {
  for (int i = 0; i < n; i++) r[i] = i;
}
int get(int x) { return r[x] == x ? x : (r[x] = get(r[x])); }
void join(int a, int b) { r[get(b)] = get(a); }
int p[N], level[N];
void dfs(int u) {
  for (int v : adj[u]) {
    if (p[u] == v) continue;
    p[v] = u;
    level[v] = level[u] + 1;
    dfs(v);
  }
}
#undef int
int main() {
#define int long long
  freopen("bridges.in", "r", stdin);
  freopen("bridges.out", "w", stdout);
  int n = RI;
  int m = RI;
  for (int i = 0; i < m; i++) {
    x[i] = RI;
    y[i] = RI;
  }
  int k = RI;
  for (int i = 0; i < k; i++) {
    x[i + m] = RI;
    y[i + m] = RI;
  }
  m += k;
  init(n + 1);
  for (int i = 0; i < m; i++) {
    if (get(x[i]) != get(y[i])) {
      join(x[i], y[i]);
      adj[x[i]].push_back(y[i]);
      adj[y[i]].push_back(x[i]);
      mark[i] = 1;
    }
  }
  init(n + 1);
  for (int i = 1; i <= n; i++)
    if (!p[i]) dfs(i);
  int ans = 0;
  for (int i = 0; i < m; i++) {
    if (mark[i])
      ans++;
    else {
      int a = get(x[i]);
      int b = get(y[i]);
      for (; a != b; a = get(a)) {
        if (level[b] > level[a]) swap(a, b);
        ans--;
        join(p[a], a);
      }
    }
    if (i >= m - k) printf("%lld\n", ans);
  }
}
```

10.4. Minimum Lex Rotation

```
int min_lex_rotation(string s) {
  s = s + s;
  int len = s.size(), i = 0, j = 1, k = 0;
  while (i + k < len && j + k < len) {
    if (s[i + k] == s[j + k])
      k++;
    else if (s[i + k] > s[j + k]) {
      i = i + k + 1;
      if (i <= j) i = j + 1;
      k = 0;
    } else if (s[i + k] < s[j + k]) {
      j = j + k + 1;
      if (j <= i) j = i + 1;
      k = 0;
    }
  }
  return min(i, j);
}
```

10.5. Bitmasks

```
// subsets of size k trick
void nCr() {
  inline int next_bit_perm(int v) {
    int t = v | (v - 1);
    return (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v) + 1));
  }
  // ...
  // handle 0 bits separately
  for (int k = 1; k <= n; ++k) {
    for (int w = (1 << k) - 1; w < (1 << n); w = next_bit_perm(w)) {
      // do whatever you want with w
    }
  }
}
// Iterate all subsets of a bitmask in increasing order
for (int sub = 0; sub = sub - bitmask & bitmask;) {
  // do something
}
// Iterate all subsets of a bitmask in decreasing order
for (int sub = (mask - 1) & bitmask; sub > 0; sub = (sub - 1) & bitmask) {
  // do something
}
```

10.6. Caliper Usage

```
#include <bits/stdc++.h>

using namespace std;
#define int long long
#define all(a) (a).begin(), (a).end()
#define ms(a,v) memset(a, v, sizeof(a))
#define sz(v) ((int)(v).size())
#define mp make_pair
#define pb push_back
#define prev biiiiirl_sai_de_casa_comi_pra_caralho
#define next trapezio_descendente
#define index eh_ele_que_nos_vai_buscar
#define left aqui_eh_37_anos_porra
#define R32 ({i32 x; scanf("%d", &x); x;})
#define RL ({long long x; scanf("%lld", &x); x;})
#define RC ({char x; scanf(" %c", &x); x;})
#define RI RL
#define ff first
#define ss second
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;
typedef long long ll;

#define SQ(x) ((x)*(x))

const int N = 1e6;
ii p[N];

int cross(ii a, ii b){
  return a.ff*b.ss - a.ss*b.ff;
}

int cross(ii o, ii a, ii b){
  return cross(ii(a.ff-o.ff, a.ss-o.ss), ii(b.ff-o.ff, b.ss-o.ss));
}

int dist2(ii a, ii b){
  return SQ(a.ff-b.ff)+SQ(a.ss-b.ss);
}

int compute_diameter2(ii * p, int n){
  int res = 0;

  // let k be the start corresponding point
  int i = n-1, j = 0, k = 1;
  while(abs(cross(p[i], p[j], p[k+1])) > abs(cross(p[i], p[j], p[k])))
    k++;
  i=0, j=k;
  while(i <= k && j < n){
    res = max(res, dist2(p[i], p[j]));
    while(j+1 < n && abs(cross(p[i], p[i+1], p[j+1]) > abs(cross(p[i],
    p[i+1], p[j])))){
      j++;
      res = max(res, dist2(p[i], p[j]));
    }
    i++;
  }

  return res;
}

void solve(int n){
  for(int i = 0; i < n; i++){
    p[i].ff = RI;
    p[i].ss = RI;
  }

  if(n==1) return void(cout << 0 << endl);

  for(int i = 1; i < n; i++)
    if(p[0].ss > p[i].ss || (p[0].ss== p[i].ss && p[0].ff > p[i].ff))
      swap(p[0], p[i]);

  sort(p, p+n, [](ii a, ii b){
    int cc = cross(p[0], a, b);
    return cc > 0 || (cc == 0 && dist2(p[0], a) < dist2(p[0], b));
  });

  int m = 2;
  for(int i = 2; i < n; i++){
    while(m > 1 && cross(p[m-2], p[m-1], p[i]) <= 0)
      m--;
    swap(p[i], p[m++]);
  }

  cout << fixed << setprecision(11);
  if(m==2){
    cout << dist2(p[0], p[1]) << endl;
    return;
  }

  cout << compute_diameter2(p, m) << endl;
}

int32_t main(){
  int T = RI;
  while(T--) solve(RI);
}
```

```
#include <bits/stdc++.h>
#include "/home/rsalesc/CP/cp-library/codes/geometry/PolyComplex.cpp"
#undef x
#undef y

using namespace std;
#define int long long
#define all(a) (a).begin(), (a).end()
#define ms(a,v) memset(a, v, sizeof(a))
#define sz(v) ((int)(v).size())
#define mp make_pair
#define pb push_back
#define prev biiiiirl_sai_de_casa_comi_pra_caralho
#define next trapezio_descendente
#define index eh_ele_que_nos_vai_buscar
#define left aqui_eh_37_anos_porra
#define R32 ({i32 x; scanf("%d", &x); x;})
#define RL ({long long x; scanf("%lld", &x); x;})
#define RC ({char x; scanf(" %c", &x); x;})
#define RI RL
#define ff real()
#define ss imag()
typedef pair<int, int> ii;
typedef vector<int> vi;
```

```
typedef vector<vi> vvi;
typedef vector<ii> vii;
typedef long long ll;

const int N = 1e6;
point p[N];

int hull(int n){
  for(int i = 1; i < n; i++)
    if(p[i].ss < p[0].ss || (p[i].ss == p[0].ss && p[i].ff < p[0].ff))
      swap(p[i], p[0]);

  sort(p+1, p+n, [](point a, point b){
      double cc = cross(a-p[0], b-p[0]);
      if(!dcmp(cc)) return norm(a-p[0]) < norm(b-p[0]);
      return dcmp(cc) > 0;
  });

  int m = 2;
  for(int i = 2; i < n; i++){
    while(m >= 2 && ccw(p[m-2], p[m-1], p[i]) <= 0)
      m--;
    swap(p[i], p[m++]);
  }

  return m;
}

void solve(int n){
  for(int i = 0; i < n; i++){
    int a = RI;
    int b = RI;
    p[i] = point(a, b);
  }

  int m = hull(n);
  p[m] = p[0];

  double done = 0, best = numeric_limits<double>::max();
  int a = 0, b = 0, c = 0, d = 0;

  for(int i = 0; i < m; i++){
    if(p[i].imag() < p[a].imag())
      a = i;
    if(p[i].imag() > p[c].imag())
      c = i;
    if(p[i].real() > p[b].real())
      b = i;
    if(p[i].real() < p[d].real())
      d = i;
  }

  Caliper ca(p[a], 0), cb(p[b], PI/2), cc(p[c], PI), cd(p[d], 3.*PI/2);

  cerr << fixed << setprecision(2);
  //cerr << ca.ang << " " << cb.ang << " " << cc.ang << " " << cd.ang << " "
    << m << endl;

  int steps = 0;
  while(done < 2*PI){
    double ra = ca.angle_to(p[a+1]);
    double rb = cb.angle_to(p[b+1]);
    double rc = cc.angle_to(p[c+1]);
    double rd = cd.angle_to(p[d+1]);
    double mn = min({ra, rb, rc, rd});

    //if(steps < 10) cerr << mn << " " << p[a] << " " << p[b] << " " << p[c]
    << " " << p[d] << endl;

    ca.rotate(mn), cb.rotate(mn), cc.rotate(mn), cd.rotate(mn);

    if(ra == mn){
      steps++;
      point u = p[a];
      point v = p[a+1];

      double d1 = ca.dist(cc);
      double d2 = cb.dist(cd);
      if(d1+d2 < best) best = d1+d2;

      a = (a+1)%m;
      ca.move(p[a]);
    } else if(rb == mn){
      b = (b+1)%m;
      cb.move(p[b]);
    } else if(rc == mn){
      c = (c+1)%m;
      cc.move(p[c]);
    } else if(rd == mn){
      d = (d+1)%m;
      cd.move(p[d]);
    } else assert(0);

    done += mn;
  }

  cout << fixed << setprecision(11);
  cout << 2.*best << endl;
}

int32_t main(){
  int n;
  while(~scanf("%lld", &n)){
    solve(n);
  }
}
```

10.7. Notes

```
=== Lagrange Interpolation
y are the evaluations at x

P(x) = sum y_i product (x-x_j) / (x_i - x_j)

can be made a lot faster (n^2 to n in some cases)

=== DAG/Order cool theorems
dilworth => minimum number of chains (partition) = maximum size of antichain
mirsky => maximum size of chain = minimum number of antichains (partition)

=== Weighted Gcd-Sum
P(n) = sum[d|n] phi(d) sum[j=1..n/d] w(dj , n)  (you can use mobius to
    consider only coprimes)

=== Lcm sum
L(n) = ((sum[d|n] d*phi(d)) +1) * (n/2)

=== Derangements
!0 = 1 and !1 = 0
```

```
!n = (n-1)(!(n-1) + !(n-2)) = n * !(n-1) + (-1)^n

=== Erdos-Gallai
degree sum is even
(sum[1..k] d_i) <= k(k-1) + (sum[k+1..n] min(d_i, k))
for every k in [1,n]

=== Havel-Hakimi
keep adding edges (v_1, v_2), (v_1, v_3), ..., (v_1, v_{d_1 + 1})
and removing vertex v_1, where v_1 is the minimum degree vertex

=== Vandermonde's Identity
C(m+n, r) = sum[0..r] C(m, i) * (n, r-i)

=== Rencontres numbers
count permutations by number of fixed points (partial derangement)
D(n, k) = !(n-k) * C(n, k)

=== Count permutations with a_k cycles of size k
n! / (product a_i! * k^(a_i))

=== Separable Permutation
avoid patterns 2413, 3142;

=== Catalan numbers
Cat(n) = C(2n, n) - C(2n, n+1) = 1/(n+1)  * C(2n, n)
C(2n, n) = sum[0..n] C(n,i)^2

number of full binary trees with n+1 leaves
number of non-isomorphic ordered tree with n vertices
polygon cutting in triangles (n+2 sides)
number of permutations that avoid fixed pattern of length 3 (231 por stack sortable sequences)
number of ways of non-intersect pairing up vertices of a convex 2n-gon

=== Bertrand's ballot
Ties allowed: C(p+q, q) - C(p+q, q-1) = C(p+q, p) - C(p+q, p+1)
  Probability P >= Q: (p+1-q) / (p+1)

Ties not allowed: C(p+q-1, q) - C(p+q-1, q-1) = C(p+q-1, p-1) - C(p+q-1, p)
  Probability P > Q: (p-q) / (p+q)

=== Bell numbers
number of (non-ordered) partitions of a set
number of factorizations of a square free number with n prime factors
number of rhymes schemes (D cant appear right after a B, for example)

B(n+1) = sum[0..n] C(n, i) * B(i)
B(n) = sum[0..n] S{n, i}

=== Ordered Bell numbers
count number of weak orderings (cayley trees/permutations)

a(n) = sum[1..n] C(n, i) * a(n-i)
a(n) = sum[0..n] i! * S{n, i}

=== Double factorials / semifactorials
number of Stirling permutations of size k = (n+1)/2 (permutation of the multiset 2x{1..k})
=> heap ordered trees with k+1 nodes (every child has an order and has number greater than parent)
perfect matchings in complete graph K_{n+1}

=== Telephone numbers
number of matchings in K_n
number of involutions (permutation that doesnt have cycle length > 2)

T(n) = T(n-1) + (n-1)*T(n-2)

=== Eulerian numbers
A(n, 0) = 1, A(n, k) = 0 qnd k > n
A(n, m) = (n-m) * A(n-1, m-1) + (m+1)*A(n-1, m)

number of permutations of size n with m ascents

=== Eulerian numbers of second kind
number of stirling permutations of size n with m ascents
A'(0, m) = [m = 0]
A'(n, m) = (2n - m - 1) * A'(n-1, m-1) + (m+1) * A'(n-1, m)

=== Stirling numbers of second kind
number of ways to partition a set of n elements into k non-empty subsets
rhyme schemes with k distinct characters

S{n+1, k} = k * S{n, k} + S{n, k-1}
S{0, 0} = 1, S{n, 0} = S{0, n} = 0
parity: [(n-k)&((k-1)/2)) == 0]

sum[0..n] S{n, i} fall(x, i) = x^n

=== Associated Stirling numbers of second kind
each subset must have at least r elements
Sr{n+1, k} = k * Sr{n, k} + C(n, r-1) * Sr{n-r+1, k-1}

=== Reduced Stirling numbers of the second kind
each element in subset have pairwise distance >= d
Sd{n, k} = S{n-d+1, k-d+1}

=== Signed Stirling numbers of the first kind
fall(x, n) = sum[0..n] s(n, i) * x^i
s(n+1, k) = -n * s(n, k) + s(n, k-1)

=== Unsigned Stirling numbers of the first kind
count number of permutations of size n with k disjoint cycles
raise(x, n) = sum[0..n] |s[n, i]| * x^i

=== Lucas' theorem
C(m, n) == product[0..k] C(m_i, n_i) (mod p)
k is the number of digits of m, n in base p

p^k divides C(m, n) if the number of carries in add of n and m-n in base p is >= k

=== K Disjoint Spanning Trees
exists iff E(P) >= k(|P|-1) for every partition P of the vertices of the graph

=== Matroid min-max
max |I| = min[u in S] (r_1(U) + r_2(S-U))

=== Dance with mobius

Dirichlet products
(I o mi)(n) = e(n)
(I o e)(n) = I(n)
(I o I)(n) = d(n)
(I o id)(n) = dsum(n)
(I o phi)(n) = id(n)

=== Legendre's formula
```

```
gives the p-adic evaluation of n!
sum_{i=1..} floor(n/p^i) = (n-sp(n))/(p-1)

=== Pick
A = i+B/2-1

=== Tutte's Theorem
A graph, G = (V, E), has a perfect matching if and only if for every subset
    U of V, the subgraph induced by V - U has at most |U| connected
    components with an odd number of vertices.

=== Multipartite Prufer
cnt(n0, n1, ..., n_k-1; n) = n^(k-2) product {n - n_i}^(n_i-1)

=== Tutte Matrix
Aij = x_ij if i < j, -x_ij if i > j, 0 if edge does not exist

=== BEST Theorem
ec(G) = tw(G) product {deg(v)-1}!
tw(G) = is the number of arborescences rooted at w for some w
Directed kirchoff is INDEGREE - ADJ
```