```
/*Vertical Decomposition*/
typedef double dbl;
const dbl eps = 1e-9;

inline bool eq(dbl x, dbl y){
  return fabs(x - y) < eps;
}

inline bool lt(dbl x, dbl y){
  return x < y - eps;
}

inline bool gt(dbl x, dbl y){
  return x > y + eps;
}

inline bool le(dbl x, dbl y){
  return x < y + eps;
}

inline bool ge(dbl x, dbl y){
  return x > y - eps;
}

struct pt {
  dbl x, y;

  inline pt operator-(const pt &p) const {
    return pt{x - p.x, y - p.y};
  }

  inline pt operator+(const pt &p) const {
    return pt{x + p.x, y + p.y};
  }

  inline pt operator*(dbl a) const {
    return pt{x * a, y * a};
  }

  inline dbl cross(const pt &p) const {
    return x * p.y - y * p.x;
  }

  inline dbl dot(const pt &p) const {
    return x * p.x + y * p.y;
  }

  inline bool operator==(const pt &p) const {
    return eq(x, p.x) && eq(y, p.y);
  }
};

struct Line {
  pt p[2];

  Line(){}

  Line(pt a, pt b) : p{a, b} {}

  pt vec() const {
    return p[1] - p[0];
  }

  pt &operator[](size_t i){
    return p[i];
  }
};

inline bool lexComp(const pt &l, const pt &r){
  if (fabs(l.x - r.x) > eps){
    return l.x < r.x;
  } else return l.y < r.y;
}

vector <pt> interSegSeg(Line l1, Line l2){
  if (eq(l1.vec().cross(l2.vec()), 0)){
    if (!eq(l1.vec().cross(l2[0] - l1[0]), 0))
      return {};
    if (!lexComp(l1[0], l1[1]))
      swap(l1[0], l1[1]);
    if (!lexComp(l2[0], l2[1]))
      swap(l2[0], l2[1]);
    pt l = lexComp(l1[0], l2[0]) ? l2[0] : l1[0];
    pt r = lexComp(l1[1], l2[1]) ? l1[1] : l2[1];
    if (l == r)
      return {l};
    else return lexComp(l, r) ? vector < pt > {l, r} : vector<pt>();
  } else {
    dbl s = (l2[0] - l1[0]).cross(l2.vec()) / l1.vec().cross(l2.vec());
    pt inter = l1[0] + l1.vec() * s;
    if (ge(s, 0) && le(s, 1) && le((l2[0] - inter).dot(l2[1] - inter), 0))
      return {inter};
    else
      return {};
  }
}

inline char get_segtype(Line segment, pt other_point){
  if (eq(segment[0].x, segment[1].x))
    return 0;
  if (!lexComp(segment[0], segment[1]))
    swap(segment[0], segment[1]);
  return (segment[1] - segment[0]).cross(other_point - segment[0]) > 0 ? 1 : -1;
}

dbl union_area(vector <tuple<pt, pt, pt>> triangles){
  vector <Line> segments(3 * triangles.size());
  vector<char> segtype(segments.size());
  for (size_t i = 0; i < triangles.size(); i++){
    pt a, b, c;
    tie(a, b, c) = triangles[i];
    segments[3 * i] = lexComp(a, b) ? Line(a, b) : Line(b, a);
    segtype[3 * i] = get_segtype(segments[3 * i], c);
    segments[3 * i + 1] = lexComp(b, c) ? Line(b, c) : Line(c, b);
    segtype[3 * i + 1] = get_segtype(segments[3 * i + 1], a);
    segments[3 * i + 2] = lexComp(c, a) ? Line(c, a) : Line(a, c);
    segtype[3 * i + 2] = get_segtype(segments[3 * i + 2], b);
  }
  vector <dbl> k(segments.size()), b(segments.size());
  for (size_t i = 0; i < segments.size(); i++){
    if (segtype[i]){
      k[i] = (segments[i][1].y - segments[i][0].y) / (segments[i][1].x - segments[i][0].x);
      b[i] = segments[i][0].y - k[i] * segments[i][0].x;
    }
  }
  dbl ans = 0;
  for (size_t i = 0; i < segments.size(); i++){
    if (!segtype[i])
      continue;
    dbl l = segments[i][0].x, r = segments[i][1].x;
    vector <pair<dbl, int>> evts;
    for (size_t j = 0; j < segments.size(); j++){
      if (!segtype[j] || i == j)
        continue;
      dbl l1 = segments[j][0].x, r1 = segments[j][1].x;
      if (ge(l1, r) || ge(l,
```

```
r1))
        continue;
      dbl common_l = max(l,
l1), common_r = min(r, r1);
      auto pts =
interSegSeg(segments[i],
segments[j]);
      if (pts.empty()){
        dbl yl1 = k[j] *
common_l + b[j];
        dbl yl = k[i] *
common_l + b[i];
        if (lt(yl1, yl) ==
(segtype[i] == 1)){
          int evt_type = -
segtype[i] * segtype[j];

evts.emplace_back(common_l,
evt_type);

evts.emplace_back(common_r, -
evt_type);
        }
      } else if (pts.size() ==
1u){
        dbl yl = k[i] *
common_l + b[i], yl1 = k[j] *
common_l + b[j];
        int evt_type = -
segtype[i] * segtype[j];
        if (lt(yl1, yl) ==
(segtype[i] == 1)){

evts.emplace_back(common_l,
evt_type);

evts.emplace_back(pts[0].x, -
evt_type);
        }
        yl = k[i] * common_r +
b[i], yl1 = k[j] * common_r +
b[j];
        if (lt(yl1, yl) ==
(segtype[i] == 1)){

evts.emplace_back(pts[0].x,
evt_type);

evts.emplace_back(common_r, -
evt_type);
        }
      } else {
        if (segtype[j] !=
segtype[i] || j > i){

evts.emplace_back(common_l, -
2);

evts.emplace_back(common_r,
2);
        }
      }
    }
    evts.emplace_back(l, 0);
    sort(evts.begin(),
evts.end());
    size_t j = 0;
    int balance = 0;
    while (j < evts.size()){
      size_t ptr = j;
      while (ptr < evts.size()
&& eq(evts[j].first,
evts[ptr].first)){
        balance +=
evts[ptr].second;
        ++ptr;
      }
      if (!balance &&
!eq(evts[j].first, r)){
        dbl next_x = ptr ==
evts.size() ? r :
evts[ptr].first;
        ans -= segtype[i] *
(k[i] * (next_x +
evts[j].first) + 2 * b[i]) *
(next_x - evts[j].first);
      }
      j = ptr;
    }
  }
  return ans / 2;
}
```

```
/*Treap*/
```

```
template<class T>
class treap {
  struct item {
    int prior, cnt;
    T key;
    item *l, *r;

    item(T v){
      key = v;
      l = NULL;
      r = NULL;
      cnt = 1;
      prior = rand();
    }
  } *root, *node;

  int cnt(item *it){
    return it ? it->cnt : 0;
  }

  void upd_cnt(item *it){
    if (it)
      it->cnt = cnt(it->l) +
cnt(it->r) + 1;
  }

  void split(item *t, T key,
item *&l, item *&r){
    if (!t)
      l = r = NULL;
    else if (key < t->key)
      split(t->l, key, l, t-
>l), r = t;
    else
      split(t->r, key, t->r,
r), l = t;
    upd_cnt(t);
  }

  void insert(item *&t, item
*it){
    if (!t)
      t = it;
    else if (it->prior > t-
>prior)
      split(t, it->key, it->l,
it->r), t = it;
    else
      insert(it->key < t->key
? t->l : t->r, it);
    upd_cnt(t);
  }

  void merge(item *&t, item
*l, item *r){
    if (!l || !r)
      t = l ? l : r;
    else if (l->prior > r-
>prior)
      merge(l->r, l->r, r), t
= l;
    else
      merge(r->l, l, r->l), t
= r;
    upd_cnt(t);
  }

  void erase(item *&t, T key){
    if (t->key == key)
      merge(t, t->l, t->r);
    else
      erase(key < t->key ? t-
>l : t->r, key);
    upd_cnt(t);
  }

  T elementAt(item *&t, int
key){
    T ans;
    if (cnt(t->l) == key) ans
= t->key;
    else if (cnt(t->l) > key)
ans = elementAt(t->l, key);
    else ans = elementAt(t->r,
key - 1 - cnt(t->l));
    upd_cnt(t);
    return ans;
  }

  item *unite(item *l, item
*r){
    if (!l || !r) return l ? l
: r;
    if (l->prior < r->prior)
swap(l, r);
    item *lt, *rt;
    split(r, l->key, lt, rt);
    l->l = unite(l->l, lt);
    l->r = unite(l->r, rt);
    upd_cnt(l);
    upd_cnt(r);
    return l;
  }

  void heapify(item *t){
    if (!t) return;
    item *max = t;
```

```
    if (t->l != NULL && t->l-
>prior > max->prior)
      max = t->l;
    if (t->r != NULL && t->r-
>prior > max->prior)
      max = t->r;
    if (max != t){
      swap(t->prior, max-
>prior);
      heapify(max);
    }
  }

  item *build(T *a, int n){
    if (n == 0) return NULL;
    int mid = n / 2;
    item *t = new item(a[mid],
rand());
    t->l = build(a, mid);
    t->r = build(a + mid + 1,
n - mid - 1);
    heapify(t);
    return t;
  }

  void output(item *t, vector
<T> &arr){
    if (!t) return;
    output(t->l, arr);
    arr.push_back(t->key);
    output(t->r, arr);
  }

public:
  treap(){
    root = NULL;
  }

  treap(T *a, int n){
    build(a, n);
  }

  void insert(T value){
    node = new item(value);
    insert(root, node);
  }

  void erase(T value){
    erase(root, value);
  }

  T elementAt(int position){
    return elementAt(root,
position);
  }

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

  void output(vector <T>
&arr){
    output(root, arr);
  }

  int range_query(T l, T r)
//(l,r]
```

```
  {
    item *previous, *next,
*current;
    split(root, l, previous,
current);
    split(current, r, current,
next);
    int ans = cnt(current);
    merge(root, previous,
current);
    merge(root, root, next);
    previous = NULL;
    current = NULL;
    next = NULL;
    return ans;
  }
};
```

```
/*System of Linear Equations*/
const double EPS = 1e-9;
const int INF = 2; // it
doesn't have to be infinity or
a big number
int gauss(vector
<vector<double>> a,
vector<double> &ans){
  int n = (int) a.size();
  int m = (int) a[0].size() -
1;
  vector<int> where(m, -1);
  for (int col = 0, row = 0;
col < m && row < n; ++col){
    int sel = row;
    for (int i = row; i < n;
++i)
      if (abs(a[i][col]) >
abs(a[sel][col]))
        sel = i;
    if (abs(a[sel][col]) <
EPS)
      continue;
    for (int i = col; i <= m;
++i)
      swap(a[sel][i],
a[row][i]);
    where[col] = row;
    for (int i = 0; i < n;
++i)
      if (i != row){
        double c = a[i][col] /
a[row][col];
        for (int j = col; j <=
m; ++j)
          a[i][j] -= a[row][j]
* c;
      }
    ++row;
  }
  ans.assign(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1)
      ans[i] = a[where[i]][m]
/ a[where[i]][i];
  for (int i = 0; i < n; ++i){
    double sum = 0;
    for (int j = 0; j < m;
++j)
      sum += ans[j] * a[i][j];
    if (abs(sum - a[i][m]) >
EPS)
      return 0;
  }
  for (int i = 0; i < m; ++i)
    if (where[i] == -1)
    return INF;
  return 1;
}
```

```
/*Suffix Automaton*/
class SuffixAutomaton {
  bool complete;
  int last;
  set<char> alphabet;

  struct state {
    int len, link, endpos,
first_pos,
shortest_non_appearing_string,
height;
    long long substrings,
length_of_substrings;
    bool is_clone;
    map<char, int> next;
    vector<int> inv_link;

    state(int leng = 0, int li
= 0){
      len = leng;
      link = li;
      first_pos = -1;
      substrings = 0;
      length_of_substrings =
0;
      endpos = 1;

shortest_non_appearing_string
= 0;
      is_clone = false;
      height = 0;
    }
  };

  vector <state> st;

  void process(int node){
    map<char, int>::iterator
mit;
    st[node].substrings = 1;

st[node].shortest_non_appearin
g_string = st.size();
    if ((int)
st[node].next.size() < (int)
alphabet.size())
st[node].shortest_non_appearin
g_string = 1;
    for (mit =
st[node].next.begin(); mit !=
st[node].next.end(); ++mit){
      if (st[mit-
>second].substrings == 0)
process(mit->second);
      st[node].height =
max(st[node].height, 1 +
```

```
st[mit->second].height);
      st[node].substrings =
st[node].substrings + st[mit-
>second].substrings;

st[node].length_of_substrings
=

st[node].length_of_substrings
+ st[mit-
>second].length_of_substrings
+ st[mit->second].substrings;

st[node].shortest_non_appearin
g_string =
min(st[node].shortest_non_appe
aring_string,
                             1
+ st[mit-
>second].shortest_non_appearin
g_string);
    }
    if (st[node].link != -1){

st[st[node].link].inv_link.pus
h_back(node);
    }
  }

  void set_suffix_links(int
node){
    int i;
    for (i = 0; i <
st[node].inv_link.size();
i++){

set_suffix_links(st[node].inv_
link[i]);
      st[node].endpos =
st[node].endpos +
st[st[node].inv_link[i]].endpo
s;
    }
  }

  void
output_all_occurrences(int v,
int P_length, vector<int>
&pos){
    if (!st[v].is_clone)

pos.push_back(st[v].first_pos
- P_length + 1);
    for (int u :
st[v].inv_link)

output_all_occurrences(u,
P_length, pos);
  }

  void kth_smallest(int node,
int k, vector<char> &str){
    if (k == 0) return;
    map<char, int>::iterator
mit;
    for (mit =
st[node].next.begin(); mit !=
st[node].next.end(); ++mit){
      if (st[mit-
>second].substrings < k) k = k
- st[mit->second].substrings;
      else {
        str.push_back(mit-
>first);
        kth_smallest(mit-
>second, k - 1, str);
        return;
      }
    }
  }

  int
find_occurrence_index(int
node, int index, vector<char>
&str){
    if (index == str.size())
return node;
    if
(!st[node].next.count(str[inde
x])) return -1;
    else return
find_occurrence_index(st[node]
.next[str[index]], index + 1,
str);
  }

  void klen_smallest(int node,
int k, vector<char> &str){
    if (k == 0) return;
    map<char, int>::iterator
mit;
    for (mit =
st[node].next.begin(); mit !=
st[node].next.end(); ++mit){
      if (st[mit-
>second].height >= k - 1){
        str.push_back(mit-
>first);
        klen_smallest(mit-
>second, k - 1, str);
        return;
      }
    }
  }

  void
minimum_non_existing_string(in
t node, vector<char> &str){
    map<char, int>::iterator
mit;
    set<char>::iterator sit;
    for (mit =
st[node].next.begin(), sit =
alphabet.begin(); sit !=
alphabet.end(); ++sit, ++mit){
      if (mit ==
st[node].next.end() || mit-
>first != (*sit)){
        str.push_back(*sit);
        return;
      } else if
(st[node].shortest_non_appeari
ng_string == 1 + st[mit-
>second].shortest_non_appearin
g_string){
        str.push_back(*sit);

minimum_non_existing_string(mi
t->second, str);
        return;
      }
    }
  }

  void find_substrings(int
node, int index, vector<char>
&str, vector <pair<long long,
long long>> &sub_info){

sub_info.push_back(make_pair(s
t[node].substrings,
st[node].length_of_substrings
+ st[node].substrings *
index));
    if (index == str.size())
return;
    if
(st[node].next.count(str[index
])){

find_substrings(st[node].next[
str[index]], index + 1, str,
sub_info);
      return;
    } else {

sub_info.push_back(make_pair(0
, 0));
    }
  }

  void check(){
    if (!complete){
      process(0);
      set_suffix_links(0);
      int i;
      complete = true;
    }
  }

public:
  SuffixAutomaton(set<char>
&alpha){
    st.push_back(state(0, -
1));
    last = 0;
    complete = false;
    set<char>::iterator sit;
    for (sit = alpha.begin();
sit != alpha.end(); sit++){
      alphabet.insert(*sit);
    }
    st[0].endpos = 0;
  }

  void sa_extend(char c){
    int cur = st.size();

st.push_back(state(st[last].le
n + 1));
    st[cur].first_pos =
```

```
st[cur].len - 1;
    int p = last;
    while (p != -1 &&
!st[p].next.count(c)){
      st[p].next[c] = cur;
      p = st[p].link;
    }
    if (p == -1){
      st[cur].link = 0;
    } else {
      int q = st[p].next[c];
      if (st[p].len + 1 ==
st[q].len){
        st[cur].link = q;
        //printf("Set link %d
-> %d\n",cur,q);
      } else {
        int clone = st.size();

st.push_back(state(st[p].len +
1, st[q].link));
        st[clone].next =
st[q].next;
        st[clone].is_clone =
true;
        st[clone].endpos = 0;
        st[clone].first_pos =
st[q].first_pos;
        while (p != -1 &&
st[p].next[c] == q){
          st[p].next[c] =
clone;
          p = st[p].link;
        }
        st[q].link =
st[cur].link = clone;
      }
    }
    last = cur;
    complete = false;
  }

  ~SuffixAutomaton(){
    int i;
    for (i = 0; i < st.size();
i++){
      st[i].next.clear();
      st[i].inv_link.clear();
    }
    st.clear();
    alphabet.clear();
  }

  void kth_smallest(int k,
vector<char> &str){
    check();
    kth_smallest(0, k, str);
  }

  int
FindFirstOccurrenceIndex(vecto
r<char> &str){
    check();
    int ind =
find_occurrence_index(0, 0,
str);
    if (ind == 0) return -1;
    else if (ind == -1) return
st.size();
    else return
st[ind].first_pos + 1 - (int)
str.size();
  }

  void
FindAllOccurrenceIndex(vector<
char> &str, vector<int> &pos){
    check();
    int ind =
find_occurrence_index(0, 0,
str);
    if (ind != -1)
output_all_occurrences(ind,
str.size(), pos);
  }

  int Occurrences(vector<char>
&str){
    check();
    int ind =
find_occurrence_index(0, 0,
str);
    if (ind == 0) return 1;
    else if (ind == -1) return
0;
    else return
st[ind].endpos;
  }

  void klen_smallest(int k,
vector<char> &str){
    check();
    if (st[0].height >= k)
klen_smallest(0, k, str);
  }

  void
minimum_non_existing_string(ve
ctor<char> &str){
    check();
    int ind =
find_occurrence_index(0, 0,
str);
    if (ind != -1)
minimum_non_existing_string(in
d, str);
  }
};
```

```
/*Suffix Array*/
#define MAX 100000
    vector<int>

sort_cyclic_shifts(char *s){
  int n = strlen(s);
  const int alphabet = 256;
  vector<int> p(n), c(n),
cnt(max(alphabet, n), 0);
  for (int i = 0; i < n; i++)
    cnt[s[i]]++;
  for (int i = 1; i <
alphabet; i++)
    cnt[i] += cnt[i - 1];
  for (int i = 0; i < n; i++)
    p[--cnt[s[i]]] = i;
  c[p[0]] = 0;
  int classes = 1;
  for (int i = 1; i < n; i++){
    if (s[p[i]] != s[p[i -
1]])
      classes++;
    c[p[i]] = classes - 1;
  }
  vector<int> pn(n), cn(n);
  for (int h = 0; (1 << h) <
n; ++h){
    for (int i = 0; i < n;
i++){
      pn[i] = p[i] - (1 << h);
      if (pn[i] < 0)
        pn[i] += n;
    }
    fill(cnt.begin(),
cnt.begin() + classes, 0);
    for (int i = 0; i < n;
i++)
      cnt[c[pn[i]]]++;
    for (int i = 1; i <
classes; i++)
      cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >=
0; i--)
      p[--cnt[c[pn[i]]]] =
pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i = 1; i < n;
i++){
      int ind = p[i] + (1 <<
h);
      if (ind >= n) ind = ind
- n;
      pair<int, int> cur =
{c[p[i]], c[ind]};
      ind = p[i - 1] + (1 <<
h);
      if (ind >= n) ind = ind
- n;
      pair<int, int> prev =
{c[p[i - 1]], c[ind]};
      if (cur != prev)
        ++classes;
      cn[p[i]] = classes - 1;
    }
    c.swap(cn);
  }
  return p;
}

vector<int>
suffix_array_construction(char
*s){
  int n = strlen(s);
  s[n] = '#';
  vector<int> sorted_shifts =
sort_cyclic_shifts(s);

sorted_shifts.erase(sorted_shi
fts.begin());
  s[n] = '\0';
  return sorted_shifts;
```

```
}

vector<int>
lcp_construction(char *s,
vector<int> const &p){
  int n = strlen(s);
  vector<int> rank(n, 0);
  for (int i = 0; i < n; i++)
    rank[p[i]] = i;

  int k = 0;
  vector<int> lcp(n - 1, 0);
  for (int i = 0; i < n; i++){
    if (rank[i] == n - 1){
      k = 0;
      continue;
    }
    int j = p[rank[i] + 1];
    while (i + k < n && j + k
< n && s[i + k] == s[j + k])
      k++;
    lcp[rank[i]] = k;
    if (k)
      k--;
  }
  return lcp;
}

int lcp(int i, int j){
  int ans = 0;
  for (int k = log_n; k >= 0;
k--){
    if (c[k][i] == c[k][j]){
      ans += 1 << k;
      i += 1 << k;
      j += 1 << k;
    }
  }
  return ans;
}

/*Strongly Connected
Component*/

vector <vector<int>> adj,
adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v){
  used[v] = true;

  for (auto u : adj[v])
    if (!used[u])
      dfs1(u);

  order.push_back(v);
}

void dfs2(int v){
  used[v] = true;
  component.push_back(v);

  for (auto u : adj_rev[v])
    if (!used[u])
      dfs2(u);
}
```

```
int main(){
  int n;
  // ... read n ...

  for (;;){
    int a, b;
    // ... read next directed
edge (a,b) ...
    adj[a].push_back(b);
    adj_rev[b].push_back(a);
  }

  used.assign(n, false);

  for (int i = 0; i < n; i++)
    if (!used[i])
      dfs1(i);

  used.assign(n, false);
  reverse(order.begin(),
order.end());

  for (auto v : order)
    if (!used[v]){
      dfs2(v);
      // ... processing next
component ...
      component.clear();
    }
}

/*Sparse Table*/
template<class T>
class STable {
  int n;
  pair<int, int> *cal;
  vector <T> *SparseTable;

  T (*comp)(T, T);

  void initialize(){
    int i, j;
    cal[1].second = 1;
    for (i = 1, j = 1 << i; j
<= n; i++, j = 1 << i){
      cal[j].first = 1;
      cal[j].second = j;
    }
    for (i = 2; i <= n; i++){
      cal[i].first =
cal[i].first + cal[i -
1].first;
      if (cal[i].second == 0)
cal[i].second = cal[i -
1].second;
    }
  }

public:
  STable(vector <T> &arr, T
(*f)(T, T)){
    n = arr.size();
    comp = f;
    cal = new pair<int, int>[n
+ 1];
    initialize();
```

```
    SparseTable = new
vector<T>[n];
    int i, j, m;
    for (i = 0, j = 0; i < n;
i++){

SparseTable[i].push_back(arr[i
]);
    }
    for (j = 0, m = 1 << j; m
< n; j++, m = 1 << j){
      for (i = 0; i + m < n;
i++){

SparseTable[i].push_back(comp(
SparseTable[i][j],
SparseTable[i +
m][SparseTable[i + m].size() -
1]));
      }
    }
  }

  T query(int l, int r){
    int difference = (r - l +
1);
    return
comp(SparseTable[l][cal[differ
ence].first],
          SparseTable[r -
cal[difference].second +
1][cal[difference].first]);
  }

  ~STable(){
    int i;
    for (i = 0; i < n; i++){
      SparseTable[i].clear();
    }
    delete[]SparseTable;
    delete[]cal;
    comp = 0;
  }
}

/* Sieve Multiplicatives*/
const int MAX = 10000005;
int phi[MAX], dvc[MAX],
sig[MAX], mob[MAX];
// phi = Euler Phi function
// dvc = divisor count ot
sigma 0
// sig = sigma, the sum of
the divisors of n. Also
called sigma_1
// mob = mobius function
int least[MAX], lstCnt[MAX],
lstSum[MAX];
vector<int> primes;

void RunLinearSieve(int n) {
  n = max(n, 1);
  for (int i = 0; i <= n;
i++) least[i] = lstCnt[i] =
lstSum[i] = 0;

  primes.clear();
  phi[1] = dvc[1] = sig[1] =
```

```
mob[1] = 1;

  for (int i = 2; i <= n;
i++) {
    if (least[i] == 0) {
      least[i] = i;
      lstCnt[i] = 1;
      lstSum[i] = 1 + i;
      phi[i] = i - 1;
      dvc[i] = 2;
      sig[i] = 1 + i;
      mob[i] = -1;
      primes.push_back(i);
    }
    for (int x : primes) {
      if (x > least[i] || i
* x > n) break;
      least[i * x] = x;
      if (least[i] == x) {
        lstCnt[i * x] =
lstCnt[i] + 1;
        lstSum[i * x] = 1 +
x * lstSum[i];
        phi[i * x] = phi[i]
* x;
        dvc[i * x] = dvc[i]
/ (lstCnt[i] + 1) *
(lstCnt[i * x] + 1);
        sig[i * x] = sig[i]
/ lstSum[i] * lstSum[i * x];
        mob[i * x] = 0;
      } else {
        lstCnt[i * x] = 1;
        lstSum[i * x] = 1 +
x;
        phi[i * x] = phi[i]
* (x - 1);
        dvc[i * x] = dvc[i]
* 2;
        sig[i * x] = sig[i]
* (1 + x);
        mob[i * x] = -
mob[i];
      }
    }
  }
}
```

/*Roman Arabic*/

```
map<int,string> AtoR;
map<char,int>RtoA;
void preprocess(){
// Map of arabic to romans
  AtoR[1000] = "M";
AtoR[900] = "CM";  AtoR[500]
= "D"; AtoR[400] = "CD";
  AtoR[100] = "C";
AtoR[90] = "XC";   AtoR[50]
= "L";  AtoR[40] = "XL";
  AtoR[10] = "X";    AtoR[9]
= "IX";    AtoR[5] = "V";
AtoR[4] = "IV";
  AtoR[1] = "I";
  // Map of romans to Arabic
  RtoA['I'] = 1;
RtoA['V'] = 5;
  RtoA['X'] = 10;
RtoA['L'] = 50;
  RtoA['C'] = 100;
RtoA['D'] = 500;
  RtoA['M'] = 1000;
}
// Arabic numerals to Roman
string ArabicToRoman(int A){
  string R = "";
  for(auto i =
AtoR.rbegin();i!=AtoR.rend()
;i++){
    while(A>=i->first){
      R = R+((string)i-
>second).c_str();
      A-=i->first;
    }
  }
  return R;
}
// Roman numerals to Arabic
int RomanToArabic(string R){
  int value = 0;
  int n = R.size();
  for(int i=0;i<n;i++){
    if( R[i+1] && RtoA[ R[i]
] < RtoA[ R[i+1] ] ){
      value+= RtoA[ R[i+1] ]
- RtoA[ R[i] ];
      i++;
    } else{
      value+=RtoA[ R[i] ];
    }
  }
  return value;
}
```

/*Rank of A Matrix*/

```
const double EPS = 1E-9;

int compute_rank(vector < vector <
double >> A){
 int n = A.size();
 int m = A[0].size();
 int rank = 0;
 vector<bool> row_selected(n,
false);
 for (int i = 0; i < m; ++i){
  int j;
  for (j = 0; j < n; ++j){
   if (!row_selected[j] && abs(A[j][i])
> EPS)
    break;
  }
  if (j != n){
   ++rank;
   row_selected[j] = true;
   for (int p = i + 1; p < m; ++p)
    A[j][p] /= A[j][i];
   for (int k = 0; k < n; ++k){
    if (k != j && abs(A[k][i]) > EPS){
     for (int p = i + 1; p < m; ++p)
      A[k][p] -= A[j][p] * A[k][i];
    }
   }
  }
 }
 return rank;
}
```

/*Ordered Set*/

```
#include
<ext/pb_ds/assoc_container.hpp> //
Common file
#include
<ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
#define MAX 400000
#define FASTIO
ios_base::sync_with_stdio(false);cin.t
ie(NULL)
typedef tree<int, null_type, less <
int>, rb_tree_tag,
tree_order_statistics_node_update>
new_data_set;
new_data_set s[MAX];
//s[j].order_of_key(int);
```

/*Mo*/

```
void remove(int idx);  //
TODO: remove value at idx from
data structure
void add(int idx);    // TODO:
add value at idx from data
structure
int get_answer();   // TODO:
extract the current answer of
the data structure
int block_size;

struct Query {
  int l, r, k, idx;

  bool operator<(Query other)
const {
    if (l / block_size !=
other.l / block_size) return
(l < other.l);
    return (l / block_size &
1) ? (r < other.r) : (r >
other.r);
  }
};

vector<int>
mo_s_algorithm(vector < Query
> queries){
  vector<int>
answers(queries.size());
  sort(queries.begin(),
queries.end());
  // TODO: initialize data
structure
  int cur_l = 0;
  int cur_r = -1;
```

```
  for (Query q : queries){
    while (cur_l > q.l){
      cur_l--;
      add(cur_l);
    }
    while (cur_r < q.r){
      cur_r++;
      add(cur_r);
    }
    while (cur_l < q.l){
      remove(cur_l);
      cur_l++;
    }
    while (cur_r > q.r){
      remove(cur_r);
      cur_r--;
    }
    answers[q.idx] =
get_answer();
  }
  return answers;
}
```

/*Minkowski*/

```
struct pt {
  long long x, y;

  pt(){}

  pt(long long _x, long long
_y) : x(_x), y(_y){}

  pt operator+(const pt &p)
const {
    return pt(x + p.x, y +
p.y);
  }

  pt operator-(const pt &p)
const {
    return pt(x - p.x, y -
p.y);
  }

  long long cross(const pt &p)
const {
    return x * p.y - y * p.x;
  }

  long long dot(const pt &p)
const {
    return x * p.x + y * p.y;
  }

  long long cross(const pt &a,
const pt &b) const {
    return (a - *this).cross(b
- *this);
  }

  long long dot(const pt &a,
const pt &b) const {
    return (a - *this).dot(b -
*this);
  }

  long long sqrLen() const {
    return this->dot(*this);
  }
};

class pointLocationInPolygon {
  bool lexComp(const pt &l,
const pt &r){
    return l.x < r.x || (l.x
== r.x && l.y < r.y);
  }

  int sgn(long long val){
    return val > 0 ? 1 : (val
== 0 ? 0 : -1);
  }

  vector <pt> seq;
  int n;
  pt translate;

  bool pointInTriangle(pt a,
pt b, pt c, pt point){
    long long s1 =
abs(a.cross(b, c));
    long long s2 =
abs(point.cross(a, b)) +
abs(point.cross(b, c)) +
abs(point.cross(c, a));
    return s1 == s2;
  }

public:
  pointLocationInPolygon(){

  }


pointLocationInPolygon(vector
<pt> &points){
    prepare(points);
  }

  void prepare(vector <pt>
&points){
    seq.clear();
    n = points.size();
    int pos = 0;
    for (int i = 1; i < n;
i++){
      if (lexComp(points[i],
points[pos]))
        pos = i;
    }
    translate.x =
points[pos].x;
    translate.y =
points[pos].y;
    rotate(points.begin(),
points.begin() + pos,
points.end());
    n--;
    seq.resize(n);
    for (int i = 0; i < n;
i++)
      seq[i] = points[i + 1] -
points[0];
  }

  bool pointInConvexPolygon(pt
point){
    point.x -= translate.x;
    point.y -= translate.y;
    if (seq[0].cross(point) !=
0 && sgn(seq[0].cross(point))
!= sgn(seq[0].cross(seq[n -
1])))
      return false;
    if (seq[n -
1].cross(point) != 0 &&
sgn(seq[n - 1].cross(point))
!= sgn(seq[n -
1].cross(seq[0])))
      return false;
    if (seq[0].cross(point) ==
0)
      return seq[0].sqrLen()
>= point.sqrLen();
    int l = 0, r = n - 1;
    while (r - l > 1){
      int mid = (l + r) / 2;
      int pos = mid;
      if
(seq[pos].cross(point) >= 0)l
= mid;
      else r = mid;
    }
    int pos = l;
    return
pointInTriangle(seq[pos],
seq[pos + 1], pt(0, 0),
point);
  }

  ~pointLocationInPolygon(){
    seq.clear();
  }
}

class Minkowski {
  static void
reorder_polygon(vector <pt>
&P){
    size_t pos = 0;
    for (size_t i = 1; i <
P.size(); i++){
      if (P[i].y < P[pos].y ||
(P[i].y == P[pos].y && P[i].x
< P[pos].x))
        pos = i;
    }
    rotate(P.begin(),
P.begin() + pos, P.end());
  }

public:
  static vector <pt>
minkowski(vector <pt> P,
vector <pt> Q){
    // the first vertex must
be the lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic
indexing
```

```
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while (i < P.size() - 2 ||
j < Q.size() - 2){
      result.push_back(P[i] +
Q[j]);
      auto cross = (P[i + 1] -
P[i]).cross(Q[j + 1] - Q[j]);
      if (cross >= 0)
        ++i;
      if (cross <= 0)
        ++j;
    }
    return result;
  }
};
```

/*Minimum Cost Maximum Flow*/

```
struct Edge {
  int from, to, capacity,
cost;
};
vector <vector<int>> adj,
cost, capacity;
const int INF = 1e9;

void shortest_paths(int n, int
v0, vector<int> &d,
vector<int> &p){
  d.assign(n, INF);
  d[v0] = 0;
  vector<bool> inq(n, false);
  queue<int> q;
  q.push(v0);
  p.assign(n, -1);
  while (!q.empty()){
    int u = q.front();
    q.pop();
    inq[u] = false;
    for (int v : adj[u]){
      if (capacity[u][v] > 0
&& d[v] > d[u] + cost[u][v]){
        d[v] = d[u] +
cost[u][v];
        p[v] = u;
        if (!inq[v]){
          inq[v] = true;
          q.push(v);
        }
      }
    }
  }
}

int min_cost_flow(int N,
vector <Edge> edges, int K,
int s, int t){
  adj.assign(N,
vector<int>());
  cost.assign(N,
vector<int>(N, 0));
  capacity.assign(N,
vector<int>(N, 0));
  for (Edge e : edges){

adj[e.from].push_back(e.to);

adj[e.to].push_back(e.from);
    cost[e.from][e.to] =
e.cost;
    cost[e.to][e.from] = -
e.cost;
    capacity[e.from][e.to] =
e.capacity;
  }

  int flow = 0;
  int cost = 0;
  vector<int> d, p;
  while (flow < K){
    shortest_paths(N, s, d,
p);
    if (d[t] == INF)
      break;

    // find max flow on that
path
    int f = K - flow;
    int cur = t;
    while (cur != s){
      f = min(f,
capacity[p[cur]][cur]);
      cur = p[cur];
    }

    // apply flow
    flow += f;
    cost += f * d[t];
    cur = t;
    while (cur != s){
      capacity[p[cur]][cur] -=
f;
      capacity[cur][p[cur]] +=
f;
      cur = p[cur];
    }
  }

  if (flow < K)
    return -1;
  else
    return cost;
}
```

/*Maximum Bipartite Matching*/

```
// A class to represent
Bipartite graph for
// Hopcroft Karp
implementation
class BGraph {
  // m and n are number of
vertices on left
  // and right sides of
Bipartite Graph
  int m, n;
  // adj[u] stores adjacents
of left side
  // vertex 'u'. The value of
u ranges from 1 to m.
  // 0 is used for dummy
vertex
  std::list<int> *adj;
  // pointers for
hopcroftKarp()
  int *pair_u, *pair_v, *dist;
public:
  BGraph(int m, int n);   //
Constructor
  void addEdge(int u, int v);
// To add edge
  // Returns true if there is
an augmenting path
  bool bfs();

  // Adds augmenting path if
there is one beginning
  // with u
  bool dfs(int u);

  // Returns size of maximum
matching
  int hopcroftKarpAlgorithm();
};

// Returns size of maximum
matching
int
BGraph::hopcroftKarpAlgorithm(
){
  // pair_u[u] stores pair of
u in matching on left side of
Bipartite Graph.
  // If u doesn't have any
pair, then pair_u[u] is NIL
  pair_u = new int[m + 1];
  // pair_v[v] stores pair of
v in matching on right side of
Biparite Graph.
  // If v doesn't have any
pair, then pair_u[v] is NIL
  pair_v = new int[n + 1];
  // dist[u] stores distance
of left side vertices
  dist = new int[m + 1];
  // Initialize NIL as pair of
all vertices
  for (int u = 0; u <= m; u++)
    pair_u[u] = NIL;
  for (int v = 0; v <= n; v++)
    pair_v[v] = NIL;
  // Initialize result
  int result = 0;
  // Keep updating the result
while there is an
  // augmenting path possible.
  while (bfs()){
    // Find a free vertex to
check for a matching
    for (int u = 1; u <= m;
u++)
      // If current vertex is
free and there is
      // an augmenting path
from current vertex
      // then increment the
result
```

```
      if (pair_u[u] == NIL &&
dfs(u))
        result++;
  }
  return result;
}

// Returns true if there is an
augmenting path available,
else returns false
bool BGraph::bfs(){
  std::queue<int> q; //an
integer queue for bfs
  // First layer of vertices
(set distance as 0)
  for (int u = 1; u <= m;
u++){
    // If this is a free
vertex, add it to queue
    if (pair_u[u] == NIL){
      // u is not matched so
distance is 0
      dist[u] = 0;
      q.push(u);
    }
      // Else set distance as
infinite so that this vertex
is considered next time for
availibility
    else
      dist[u] = INF;
  }
  // Initialize distance to
NIL as infinite
  dist[NIL] = INF;
  // q is going to contain
vertices of left side only.
  while (!q.empty()){
    // dequeue a vertex
    int u = q.front();
    q.pop();
    // If this node is not NIL
and can provide a shorter path
to NIL then
    if (dist[u] < dist[NIL]){
      // Get all the adjacent
vertices of the dequeued
vertex u
      std::list<int>::iterator
it;
      for (it =
adj[u].begin(); it !=
adj[u].end(); ++it){
        int v = *it;
        // If pair of v is not
considered so far
        // i.e. (v, pair_v[v])
is not yet explored edge.
        if (dist[pair_v[v]] ==
INF){
          // Consider the pair
and push it to queue
          dist[pair_v[v]] =
dist[u] + 1;
          q.push(pair_v[v]);
        }
      }
    }
  }
  // If we could come back to
NIL using alternating path of
distinct
  // vertices then there is an
augmenting path available
  return (dist[NIL] != INF);
}

// Returns true if there is an
augmenting path beginning with
free vertex u
bool BGraph::dfs(int u){
  if (u != NIL){
    std::list<int>::iterator
it;
    for (it = adj[u].begin();
it != adj[u].end(); ++it){
      // Adjacent vertex of u
      int v = *it;
      // Follow the distances
set by BFS search
      if (dist[pair_v[v]] ==
dist[u] + 1){
        // If dfs for pair of
v also returnn true then
        if (dfs(pair_v[v]) ==
true){   // new matching
possible, store the matching
          pair_v[v] = u;
          pair_u[u] = v;
          return true;
        }
      }
    }
    // If there is no
augmenting path beginning with
u then.
    dist[u] = INF;
    return false;
  }
  return true;
}

// Constructor for
initialization
BGraph::BGraph(int m, int n){
  this->m = m;
  this->n = n;
  adj = new std::list<int>[m +
1];
}

// function to add edge from u
to v
void BGraph::addEdge(int u,
int v){
  adj[u].push_back(v); // Add
v to u’s list.
}
/*Miller Rabin Algorithm*/
using u64 = uint64_t;

u64 mulmod(u64 a, u64 b, u64
c){
  u64 x = 0, y = a % c;
  while (b > 0){
    if (b % 2 == 1){
      x = (x + y) % c;
    }
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}

u64 power(u64 a, u64 b, u64
n){
  if (b == 0)
    return 1;
  if (b == 1) return a % n;
  u64 c = power(a, b / 2, n);
  u64 p = mulmod(c % n, c % n,
n);
  if (b % 2 == 0)
    return p;
  else return (mulmod(p, a,
n));
}

bool check_composite(u64 n,
u64 a, u64 d, int s){
  u64 x = power(a, d, n);
  if (x == 1 || x == n - 1)
    return false;
  for (int r = 1; r < s; r++){
    x = power(x, 2, n);
    if (x == n - 1)
      return false;
  }
  return true;
};

bool MillerRabin(u64 n){ //
returns true if n is prime,
else returns false.
  if (n < 2)
    return false;
  int r = 0;
  u64 d = n - 1;
  while ((d & 1) == 0){
    d >>= 1;
    r++;
  }
  for (int a : {2, 3, 5, 7,
11, 13, 17, 19, 23, 29, 31,
37}){
    if (n == a)
      return true;
    if (check_composite(n, a,
d, r))
      return false;
  }
  return true;
}

/*Manacher’s Algorithm*/
int main(){
  int T, l;
  char s[MAX];
  gets(s);
  int n = strlen(s);
  vector<int> d1(n);
```

```
  for (int i = 0, l = 0, r = -
1; i < n; i++){
    int k = (i > r) ? 1 :
min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k
< n && s[i - k] == s[i + k]){
      k++;
    }
    d1[i] = k--;
    if (i + k > r){
      l = i - k;
      r = i + k;
    }
  }
  vector<int> d2(n);
  for (int i = 0, l = 0, r = -
1; i < n; i++){
    int k = (i > r) ? 0 :
min(d2[l + r - i + 1], r - i +
1);
    while (0 <= i - k - 1 && i
+ k < n && s[i - k - 1] == s[i
+ k]){
      k++;
    }
    d2[i] = k--;
    if (i + k > r){
      l = i - k - 1;
      r = i + k;
    }
  }
  return 0;
}
```

/*Linear Reccurence*/

```
#define rep(i, a, b) for(int
i = a; i < (b); ++i)
#define sz(x)
(int)(x).size()

vector <ll>
berlekampMassey(vector <ll>
s) {
    int n = sz(s), L = 0, m
= 0;
    vector <ll> C(n), B(n),
T;
    C[0] = B[0] = 1;
    ll b = 1;
    rep(i, 0, n) {
        ++m;
        ll d = s[i] % mod;
        rep(j, 1, L + 1) d =
(d + C[j] * s[i - j]) % mod;
        if (!d) continue;
        T = C;
        ll coef = d *
modpow(b, mod - 2) % mod;
        rep(j, m, n) C[j] =
(C[j] - coef * B[j - m]) %
mod;
        if (2 * L > i)
continue;
        L = i + 1 - L;
        B = T;b = d;m = 0;
    }
    C.resize(L + 1);
    C.erase(C.begin());
    for (ll &x : C) x = (mod
- x) % mod;
    return C;
}

ll linearRec(vector <ll> S,
vector <ll> tr, ll k) {
    int n = sz(tr);
    auto combine =
[&](vector <ll> a, vector
<ll> b) {
        vector <ll> res(n *
2 + 1);
        rep(i, 0, n + 1)
rep(j, 0, n + 1)res[i + j] =
(res[i + j] + a[i] * b[j]) %
mod;
        for (int i = 2 * n;
i > n; --i)
            rep(j, 0,
n)res[i - 1 - j] = (res[i -
1 - j] + res[i] * tr[j]) %
mod;
        res.resize(n + 1);
        return res;
    };
    vector <ll> pol(n + 1),
e(pol);
    pol[0] = e[1] = 1;
    for (++k; k; k /= 2) {
        if (k % 2) pol =
combine(pol, e);
        e = combine(e, e);
    }
    ll res = 0;
    rep(i, 0, n) res = (res
+ pol[i + 1] * S[i]) % mod;
    return res;
}
// linearRec({0,1},{1,1},3)
//<= Fib
```

/*Line Segment Intersection*/

```
const double EPS = 1E-9;

struct pt {
  double x, y;

  bool operator<(const pt &p)
const {
    return x < p.x - EPS ||
(abs(x - p.x) < EPS && y < p.y
- EPS);
  }
};

struct line {
  double a, b, c;

  line(){}

  line(pt p, pt q){
    a = p.y - q.y;
    b = q.x - p.x;
    c = -a * p.x - b * p.y;
    norm();
  }

  void norm(){
    double z = sqrt(a * a + b
* b);
    if (abs(z) > EPS)
      a /= z, b /= z, c /= z;
  }

  double dist(pt p) const {
return a * p.x + b * p.y + c;
}
};

double det(double a, double b,
double c, double d){
  return a * d - b * c;
}

inline bool betw(double l,
double r, double x){
  return min(l, r) <= x + EPS
&& x <= max(l, r) + EPS;
}

inline bool
intersect_1d(double a, double
b, double c, double d){
  if (a > b)
    swap(a, b);
  if (c > d)
    swap(c, d);
  return max(a, c) <= min(b,
d) + EPS;
}

bool intersect(pt a, pt b, pt
c, pt d, pt &left, pt &right){
  if (!intersect_1d(a.x, b.x,
c.x, d.x) ||
!intersect_1d(a.y, b.y, c.y,
d.y))
    return false;
  line m(a, b);
  line n(c, d);
  double zn = det(m.a, m.b,
n.a, n.b);
  if (abs(zn) < EPS){
    if (abs(m.dist(c)) > EPS
|| abs(n.dist(a)) > EPS)
      return false;
    if (b < a)
      swap(a, b);
    if (d < c)
      swap(c, d);
    left = max(a, c);
    right = min(b, d);
    return true;
  } else {
    left.x = right.x = -
det(m.c, m.b, n.c, n.b) / zn;
    left.y = right.y = -
det(m.a, m.c, n.a, n.c) / zn;
    return betw(a.x, b.x,
left.x) && betw(a.y, b.y,
left.y) &&
        betw(c.x, d.x,
left.x) && betw(c.y, d.y,
left.y);
```

```
  }
}
```

*/*Li Chao Tree*/*

```
class LiChaoTree {
  long long L, R;
  bool minimize;
  int lines;

  struct Node {
    complex<long long> line;
    Node *children[2];

    Node(complex<long long> ln
= {0, 1000000000000000000}){
      line = ln;
      children[0] = 0;
      children[1] = 0;
    }
  } *root;

  long long dot(complex<long
long> a, complex<long long>
b){
    return (conj(a) *
b).real();
  }

  long long f(complex<long
long> a, long long x){
    return dot(a, {x, 1});
  }

  void clear(Node *&node){
    if (node->children[0]){
      clear(node-
>children[0]);
    }
    if (node->children[1]){
      clear(node-
>children[1]);
    }
    delete node;
  }

  void add_line(complex<long
long> nw, Node *&node, long
long l, long long r){
    if (node == 0){
      node = new Node(nw);
      return;
    }
    long long m = (l + r) / 2;
    bool lef = (f(nw, l) <
f(node->line, l) && minimize)
|| ((!minimize) && f(nw, l) >
f(node->line, l));
    bool mid = (f(nw, m) <
f(node->line, m) && minimize)
|| ((!minimize) && f(nw, m) >
f(node->line, m));
    if (mid){
      swap(node->line, nw);
    }
    if (r - l == 1){
      return;
    } else if (lef != mid){
      add_line(nw, node-
>children[0], l, m);
    } else {
      add_line(nw, node-
>children[1], m, r);
    }
  }

  long long get(long long x,
Node *&node, long long l, long
long r){
    long long m = (l + r) / 2;
    if (r - l == 1){
      return f(node->line, x);
    } else if (x < m){
      if (node->children[0] ==
0) return f(node->line, x);
      if (minimize) return
min(f(node->line, x), get(x,
node->children[0], l, m));
      else return max(f(node-
>line, x), get(x, node-
>children[0], l, m));
    } else {
      if (node->children[1] ==
0) return f(node->line, x);
      if (minimize) return
min(f(node->line, x), get(x,
node->children[1], m, r));
      else return max(f(node-
>line, x), get(x, node-
>children[1], m, r));
    }
  }

public:
  LiChaoTree(long long l = -
1000000001, long long r =
1000000001, bool mn = false){
    L = l;
    R = r;
    root = 0;
    minimize = mn;
    lines = 0;
  }

  void AddLine(pair<long long,
long long> ln){
    add_line({ln.first,
ln.second}, root, L, R);
    lines++;
  }

  int number_of_lines(){
    return lines;
  }

  long long
getOptimumValue(long long x){
    return get(x, root, L, R);
  }

  ~LiChaoTree(){
    if (root != 0)
clear(root);
  }
}
```

*/*Largest zero submatrix*/*

```
int zero_matrix(vector <
vector < int >> a){
  int n = a.size();
  int m = a[0].size();

  int ans = 0;
  vector<int> d(m, -1), d1(m),
d2(m);
  stack<int> st;
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < m;
++j){
      if (a[i][j] == 1)
        d[j] = i;
    }
    for (int j = 0; j < m;
++j){
      while (!st.empty() &&
d[st.top()] <= d[j])
        st.pop();
      d1[j] = st.empty() ? -1
: st.top();
      st.push(j);
    }
    while (!st.empty())
      st.pop();
    for (int j = m - 1; j >=
0; --j){
      while (!st.empty() &&
d[st.top()] <= d[j])
        st.pop();
      d2[j] = st.empty() ? m :
st.top();
      st.push(j);
    }
    while (!st.empty())
      st.pop();
    for (int j = 0; j < m;
++j)
      ans = max(ans, (i -
d[j]) * (d2[j] - d1[j] - 1));
  }
  return ans;
}
```

*/*Integration using Simpson*/*

```
const int N = 1000 * 1000; //
number of steps (already
multiplied by 2)
double
simpson_integration(double a,
double b){
  double h = (b - a) / N;
  double s = f(a) + f(b); // a
= x_0 and b = x_2n
  for (int i = 1; i <= N - 1;
++i){
    double x = a + h * i;
    s += f(x) * ((i & 1) ? 4 :
2);
  }
  s *= h / 3;
  return s;
}
```

```
/*Implicit Treap*/
template<class T>
class implicit_treap {
  struct item {
    int prior, cnt;
    T value;
    bool rev;
    item *l, *r;

    item(T v){
      value = v;
      rev = false;
      l = NULL;
      r = NULL;
      cnt = 1;
      prior = rand();
    }
  } *root, *node;

  int cnt(item *it){
    return it ? it->cnt : 0;
  }

  void upd_cnt(item *it){
    if (it)
      it->cnt = cnt(it->l) +
cnt(it->r) + 1;
  }

  void push(item *it){
    if (it && it->rev){
      it->rev = false;
      swap(it->l, it->r);
      if (it->l) it->l->rev ^=
true;
      if (it->r) it->r->rev ^=
true;
    }
  }

  void merge(item *&t, item
*l, item *r){
    push(l);
    push(r);
    if (!l || !r)
      t = l ? l : r;
    else if (l->prior > r-
>prior)
      merge(l->r, l->r, r), t
= l;
    else
      merge(r->l, l, r->l), t
= r;
    upd_cnt(t);
  }

  void split(item *t, item
*&l, item *&r, int key, int
add = 0){
    if (!t)
      return void(l = r = 0);
    push(t);
    int cur_key = add + cnt(t-
>l);
    if (key <= cur_key)
      split(t->l, l, t->l,
key, add), r = t;
    else
      split(t->r, t->r, r,
key, add + 1 + cnt(t->l)), l =
t;
    upd_cnt(t);
  }

  void insert(item *&t, item
*element, int key){
    item *l, *r;
    split(t, l, r, key);
    merge(l, l, element);
    merge(t, l, r);
    l = NULL;
    r = NULL;
  }

  T elementAt(item *&t, int
key){
    push(t);
    T ans;
    if (cnt(t->l) == key) ans
= t->value;
    else if (cnt(t->l) > key)
ans = elementAt(t->l, key);
    else ans = elementAt(t->r,
key - 1 - cnt(t->l));
    return ans;
  }

  void erase(item *&t, int
key){
    push(t);
    if (!t) return;
    if (key == cnt(t->l))
      merge(t, t->l, t->r);
    else if (key < cnt(t->l))
      erase(t->l, key);
    else
      erase(t->r, key - cnt(t-
>l) - 1);
    upd_cnt(t);
  }

  void reverse(item *&t, int
l, int r){
    item *t1, *t2, *t3;
    split(t, t1, t2, l);
    split(t2, t2, t3, r - l +
1);
    t2->rev ^= true;
    merge(t, t1, t2);
    merge(t, t, t3);
  }

  void cyclic_shift(item *&t,
int L, int R){
    if (L == R) return;
    item *l, *r, *m;
    split(t, t, l, L);
    split(l, l, m, R - L + 1);
    split(l, l, r, R - L);
    merge(t, t, r);
    merge(t, t, l);
    merge(t, t, m);
    l = NULL;
    r = NULL;
    m = NULL;
  }

  void output(item *t, vector
<T> &arr){
    if (!t) return;
    push(t);
    output(t->l, arr);
    arr.push_back(t->value);
    output(t->r, arr);
  }

public:
  implicit_treap(){
    root = NULL;
  }

  void insert(T value, int
position){
    node = new item(value);
    insert(root, node,
position);
  }

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

  void reverse(int l, int r){
    reverse(root, l, r);
  }

  T elementAt(int position){
    return elementAt(root,
position);
  }

  void cyclic_shift(int L, int
R){
    cyclic_shift(root, L, R);
  }

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

  void output(vector <T>
&arr){
    output(root, arr);
  }
};
```

```
/*Hungarian Algorithm*/
class HungarianAlgorithm {
  int N, inf, n, max_match;
  int *lx, *ly, *xy, *yx,
*slack, *slackx, *prev;
  int **cost;
  bool *S, *T;

  void init_labels(){
    for (int x = 0; x < n;
x++) lx[x] = 0;
    for (int y = 0; y < n;
y++) ly[y] = 0;
    for (int x = 0; x < n;
```

```
x++)
      for (int y = 0; y < n;
y++)
        lx[x] = max(lx[x],
cost[x][y]);
  }

  void update_labels(){
    int x, y, delta = inf;
//init delta as infinity
    for (y = 0; y < n; y++)
//calculate delta using slack
      if (!T[y])
        delta = min(delta,
slack[y]);
    for (x = 0; x < n; x++)
//update X labels
      if (S[x]) lx[x] -=
delta;
    for (y = 0; y < n; y++)
//update Y labels
      if (T[y]) ly[y] +=
delta;
    for (y = 0; y < n; y++)
//update slack array
      if (!T[y])
        slack[y] -= delta;
  }

  void add_to_tree(int x, int
prevx)
//x - current vertex,prevx -
vertex from X before x in the
alternating path,
//so we add edges (prevx,
xy[x]), (xy[x], x)
  {
    S[x] = true; //add x to S
    prev[x] = prevx; //we need
this when augmenting
    for (int y = 0; y < n;
y++) //update slacks, because
we add new vertex to S
      if (lx[x] + ly[y] -
cost[x][y] < slack[y]){
        slack[y] = lx[x] +
ly[y] - cost[x][y];
        slackx[y] = x;
      }
  }

  void augment() //main
function of the algorithm
  {
    if (max_match == n)
return; //check wether
matching is already perfect
    int x, y, root; //just
counters and root vertex
    int q[N], wr = 0, rd = 0;
//q - queue for bfs, wr,rd -
write and read
//pos in queue
    //memset(S, false,
sizeof(S)); //init set S
    for (int i = 0; i < n;
i++) S[i] = false;
    //memset(T, false,
sizeof(T)); //init set T
    for (int i = 0; i < n;
i++) T[i] = false;
    //memset(prev, -1,
sizeof(prev)); //init set prev
- for the alternating tree
    for (int i = 0; i < n;
i++) prev[i] = -1;
    for (x = 0; x < n; x++)
//finding root of the tree
    {
      if (xy[x] == -1){
        q[wr++] = root = x;
        prev[x] = -2;
        S[x] = true;
        break;
      }
    }
    for (y = 0; y < n; y++)
//initializing slack array
    {
      slack[y] = lx[root] +
ly[y] - cost[root][y];
      slackx[y] = root;
    }
    while (true) //main cycle
    {
      while (rd < wr)
//building tree with bfs cycle
      {
        x = q[rd++]; //current
vertex from X part
        for (y = 0; y < n;
y++) //iterate through all
edges in equality graph
        {
          if (cost[x][y] ==
lx[x] + ly[y] && !T[y]){
            if (yx[y] == -1)
break; //an exposed vertex in
Y found, so
//augmenting path exists!
            T[y] = true;
//else just add y to T,
            q[wr++] = yx[y];
//add vertex yx[y], which is
matched
//with y, to the queue
            add_to_tree(yx[y],
x); //add edges (x,y) and
(y,yx[y]) to the tree
          }
        }
        if (y < n) break;
//augmenting path found!
      }
      if (y < n) break;
//augmenting path found!
      update_labels();
//augmenting path not found,
so improve labeling
      wr = rd = 0;
      for (y = 0; y < n; y++){
        //in this cycle we add
edges that were added to the
equality graph as a
//result of improving the
labeling, we add edge
(slackx[y], y) to the tree if
//and only if !T[y] &&
slack[y] == 0, also with this
edge we add another one
//(y, yx[y]) or augment the
matching, if y was exposed
        if (!T[y] && slack[y]
== 0){
          if (yx[y] == -1)
//exposed vertex in Y found -
augmenting path exists!
          {
            x = slackx[y];
            break;
          } else {
            T[y] = true;
//else just add y to T,
            if (!S[yx[y]]){
              q[wr++] = yx[y];
//add vertex yx[y], which is
matched with
//y, to the queue

add_to_tree(yx[y], slackx[y]);
//and add edges (x,y) and (y,
//yx[y]) to the tree
            }
          }
        }
      }
      if (y < n) break;
//augmenting path found!
    }
    if (y < n) //we found
augmenting path!
    {
      max_match++; //increment
matching
//in this cycle we inverse
edges along augmenting path
      for (int cx = x, cy = y,
ty; cx != -2; cx = prev[cx],
cy = ty){
        ty = xy[cx];
        yx[cy] = cx;
        xy[cx] = cy;
      }
      augment(); //recall
function, go to step 1 of the
algorithm
    }
  }//end of augment() function
public:
  HungarianAlgorithm(int vv,
int inf = 1000000000){
    N = vv;
    n = N;
    max_match = 0;
    this->inf = inf;
    lx = new int[N];
    ly = new int[N];//labels
of X and Y parts
    xy = new int[N];//xy[x] -
vertex that is matched with x,
    yx = new int[N];//yx[y] -
```

```
vertex that is matched with y
    slack = new int[N];//as in
the algorithm description
    slackx = new
int[N];//slackx[y] such a
vertex, that l(slackx[y]) +
l(y) - w(slackx[y],y) =
slack[y]
    prev = new int[N];//array
for memorizing alternating
paths
    S = new bool[N];
    T = new bool[N];//sets S
and T in algorithm
    cost = new int *[N];//cost
matrix
    for (int i = 0; i < N;
i++){
      cost[i] = new int[N];
    }
  }

  ~HungarianAlgorithm(){
    delete[]lx;
    delete[]ly;
    delete[]xy;
    delete[]yx;
    delete[]slack;
    delete[]slackx;
    delete[]prev;
    delete[]S;
    delete[]T;
    int i;
    for (i = 0; i < N; i++){
      delete[](cost[i]);
    }
    delete[]cost;
  }

  void setCost(int i, int j,
int c){
    cost[i][j] = c;
  }

  int *matching(bool first =
true){
    int *ans;
    ans = new int[N];
    for (int i = 0; i < N;
i++){
      if (first) ans[i] =
xy[i];
      else ans[i] = yx[i];
    }
    return ans;
  }

  int hungarian(){
    int ret = 0; //weight of
the optimal matching
    max_match = 0; //number of
vertices in current matching
    for (int x = 0; x < n;
x++) xy[x] = -1;
    for (int y = 0; y < n;
y++) yx[y] = -1;
    init_labels(); //step 0
    augment(); //steps 1-3
    for (int x = 0; x < n;
x++) //forming answer there
      ret += cost[x][xy[x]];
    return ret;
  }
};
```

*/*Heavy Light Decomposition*/*

```
vector<int> parent, depth,
heavy, head, pos;
int cur_pos;

int dfs(int v, vector
<vector<int>> const &adj){
  int size = 1;
  int max_c_size = 0;
  for (int c : adj[v]){
    if (c != parent[v]){
      parent[c] = v, depth[c]
= depth[v] + 1;
      int c_size = dfs(c,
adj);
      size += c_size;
      if (c_size > max_c_size)
        max_c_size = c_size,
heavy[v] = c;
    }
  }
  return size;
}

void decompose(int v, int h,
vector <vector<int>> const
&adj){
  head[v] = h, pos[v] =
cur_pos++;
  if (heavy[v] != -1)
    decompose(heavy[v], h,
adj);
  for (int c : adj[v]){
    if (c != parent[v] && c !=
heavy[v])
      decompose(c, c, adj);
  }
}

void init(vector < vector <
int >>
const& adj){
int n = adj.size();
parent = vector < int > (n);
depth = vector < int > (n);
heavy = vector < int > (n, -
1);
head = vector < int > (n);
pos = vector < int > (n);
cur_pos = 0;

dfs(0, adj);
decompose(0, 0, adj);
}

int query(int a, int b){
  int res = 0;
  for (; head[a] != head[b]; b
= parent[head[b]]){
    if (depth[head[a]] >
depth[head[b]])
      swap(a, b);
    int cur_heavy_path_max =
segment_tree_query(pos[head[b]
], pos[b]);
    res = max(res,
cur_heavy_path_max);
  }
  if (depth[a] > depth[b])
    swap(a, b);
  int last_heavy_path_max =
segment_tree_query(pos[a],
pos[b]);
  res = max(res,
last_heavy_path_max);
  return res;
}
```

*/*Half-Plane Intersection*/*

```
class HalfPlaneIntersection {
  static double eps, inf;
public:
  struct Point {
    double x, y;

    explicit Point(double x =
0, double y = 0) : x(x),
y(y){}

    // Addition, substraction,
multiply by constant, cross
product.
    friend Point
operator+(const Point &p,
const Point &q){
      return Point(p.x + q.x,
p.y + q.y);
    }

    friend Point operator-
(const Point &p, const Point
&q){
      return Point(p.x - q.x,
p.y - q.y);
    }

    friend Point
operator*(const Point &p,
const double &k){
      return Point(p.x * k,
p.y * k);
    }

    friend double cross(const
Point &p, const Point &q){
      return p.x * q.y - p.y *
q.x;
    }
  };

// Basic half-plane struct.
  struct Halfplane {
    // 'p' is a passing point
of the line and 'pq' is the
direction vector of the line.
    Point p, pq;
```

```
    double angle;

    Halfplane(){}

    Halfplane(const Point &a,
const Point &b) : p(a), pq(b -
a){
        angle = atan2l(pq.y,
pq.x);
    }

    // Check if point 'r' is
outside this half-plane.
    // Every half-plane allows
the region to the LEFT of its
line.
    bool out(const Point &r){
        return cross(pq, r - p)
< -eps;
    }

    // If the angle of both
half-planes is equal, the
leftmost one should go first.
    bool operator<(const
Halfplane &e) const {
        if (fabsl(angle -
e.angle) < eps) return
cross(pq, e.p - p) < 0;
        return angle < e.angle;
    }

    // We use equal comparator
for std::unique to easily
remove parallel half-planes.
    bool operator==(const
Halfplane &e) const {
        return fabsl(angle -
e.angle) < eps;
    }

    // Intersection point of
the lines of two half-planes.
It is assumed they're never
parallel.
    friend Point inter(const
Halfplane &s, const Halfplane
&t){
        double alpha =
cross((t.p - s.p), t.pq) /
cross(s.pq, t.pq);
        return s.p + (s.pq *
alpha);
    }
  };

  static vector <Point>
hp_intersect(vector
<Halfplane> &H){
    Point box[4] =  //
Bounding box in CCW order
        {
            Point(inf, inf),
            Point(-inf, inf),
            Point(-inf, -inf),
            Point(inf, -inf)
        };
    for (int i = 0; i < 4;
i++)   // Add bounding box
half-planes.
    {
        Halfplane aux(box[i],
box[(i + 1) % 4]);
        H.push_back(aux);
    }
    // Sort and remove
duplicates
    sort(H.begin(), H.end());
    H.erase(unique(H.begin(),
H.end()), H.end());
    deque <Halfplane> dq;
    int len = 0;
    for (int i = 0; i <
int(H.size()); i++){
        // Remove from the back
of the deque while last half-
plane is redundant
        while (len > 1 &&
H[i].out(inter(dq[len - 1],
dq[len - 2]))){
            dq.pop_back();
            --len;
        }
        // Remove from the front
of the deque while first half-
plane is redundant
        while (len > 1 &&
H[i].out(inter(dq[0],
dq[1]))){
            dq.pop_front();
            --len;
        }
        // Add new half-plane
        dq.push_back(H[i]);
        ++len;
    }
    // Final cleanup: Check
half-planes at the front
against the back and vice-
versa
    while (len > 2 &&
dq[0].out(inter(dq[len - 1],
dq[len - 2]))){
        dq.pop_back();
        --len;
    }
    while (len > 2 && dq[len -
1].out(inter(dq[0], dq[1]))){
        dq.pop_front();
        --len;
    }
    // Report empty
intersection if necessary
    if (len < 3) return
vector<Point>();
    // Reconstruct the convex
polygon from the remaining
half-planes.
    vector<Point> ret(len);
    for (int i = 0; i + 1 <
len; i++){
        ret[i] = inter(dq[i],
dq[i + 1]);
    }
    ret.back() = inter(dq[len
- 1], dq[0]);
    return ret;
  }
};

double
HalfPlaneIntersection::eps =
1e-9;
double
HalfPlaneIntersection::inf =
1e9;
```

/*Gray Code*/

```
int g(int n){
  return n ^ (n >> 1);
}

int rev_g(int g){
  int n = 0;
  for (; g; g >>= 1)
    n ^= g;
  return n;
}
```

/*Fenwick Tree*/

```
struct FenwickTree {
  vector<int> bit;  // binary
indexed tree
  int n;

  FenwickTree(int n){
    this->n = n;
    bit.assign(n, 0);
  }

  FenwickTree(vector<int> a) :
FenwickTree(a.size()){
    for (size_t i = 0; i <
a.size(); i++)
      add(i, a[i]);
  }

  int sum(int r){
    int ret = 0;
    for (; r >= 0; r = (r & (r
+ 1)) - 1)
      ret += bit[r];
    return ret;
  }

  int sum(int l, int r){
    return sum(r) - sum(l -
1);
  }

  void add(int idx, int
delta){
    for (; idx < n; idx = idx
| (idx + 1))
      bit[idx] += delta;
  }
};
```

/*Fast Fourier Transform*/

```
using cd = complex<double>;
const double PI = acos(-1);
```

```
void fft(vector<cd>& a, bool
invert)
{
  int n = a.size();
  for(int i = 1, j = 0; i < n;
i++){
    int bit = n>>1;
    for(; j&bit; bit>>=1){
      j^=bit;
    }
    j ^= bit;
    if(i < j)
      swap(a[i], a[j]);
  }
  for(int len = 2; len <= n;
len <<= 1){
    double ang =
2*PI/len*(invert ? -1 : 1);
    cd wlen(cos(ang),
sin(ang));
    for(int i = 0; i < n; i +=
len){
      cd w(1);
      for(int j = 0; j <
len/2; j++){
        cd u = a[i+j], v =
a[i+j+len/2]*w;
        a[i+j] = u+v;
        a[i+j+len/2] = u-v;
        w *= wlen;
      }
    }
  }
  if(invert){
    for(cd &x: a)
    x /= n;
  }
}
vector<int>
multiply(vector<int> const& a,
vector<int> const&b)
{
  vector<cd> fa(a.begin(),
a.end());
  vector<cd> fb(b.begin(),
b.end());
  int n = 1;
  while(n < a.size()+b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for(int i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);
  vector<int> result(n);
  for(int i = 0; i < n; i++)
    result[i] =
round(fa[i].real());
  return result;
}

//Number Theoretic
Transformation
long long int gcd(long long
int a,long long int b)
{
  if(b==0) return a;
  else return gcd(b,a%b);
}
long long int egcd(long long
int a, long long int b, long
long int & x, long long int &
y){
  if (a == 0){
    x = 0;
    y = 1;
    return b;
  }
  long long int x1, y1;
  long long int d = egcd(b %
a, a, x1, y1);
  x = y1 - (b / a) * x1;
  y = x1;
  return d;
}
long long int
ModuloInverse(long long int
a,long long int n)
{
  long long int x,y;
  x=gcd(a,n);
  a=a/x;
  n=n/x;
  long long int res =
egcd(a,n,x,y);
  x=(x%n+n)%n;
  return x;
}
const int mod = 998244353;
const int root = 15311432;
const int root_1 = 469870224;
const int root_pw = 1 << 23;
void fft(vector<int> & a, bool
invert){
  int n = a.size();
  for (int i = 1, j = 0; i <
n; i++){
    int bit = n >> 1;
    for (; j & bit; bit >>= 1)
      j ^= bit;
    j ^= bit;
    if (i < j)
      swap(a[i], a[j]);
  }
  for (int len = 2; len <= n;
len <<= 1){
    int wlen = invert ? root_1
: root;
    for (int i = len; i <
root_pw; i <<= 1)
      wlen = (int)(1LL * wlen
* wlen % mod);
    for (int i = 0; i < n; i
+= len){
      int w = 1;
      for (int j = 0; j < len
/ 2; j++){
        int u = a[i+j], v =
(int)(1LL * a[i+j+len/2] * w %
mod);
        a[i+j] = u + v < mod ?
u + v : u + v - mod;
        a[i+j+len/2] = u - v
>= 0 ? u - v : u - v + mod;
        w = (int)(1LL * w *
wlen % mod);
      }
    }
  }
  if (invert){
    int n_1 = (int)
ModuloInverse(n, mod);
    for (int & x : a)
      x = (int)(1LL * x * n_1
% mod);
  }
}
vector<int>
multiply(vector<int> const& a,
vector<int> const&b)
{
  vector<int> fa(a.begin(),
a.end());
  vector<int> fb(b.begin(),
b.end());
  int n = 1;
  while(n < a.size()+b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for(int i = 0; i < n; i++)
    fa[i] = (int)
(1LL*fa[i]*fb[i]%mod);
  fft(fa, true);
  vector<int> result(n);
  for(int i = 0; i < n; i++)
    result[i] = fa[i];
  return result;
}
/*Eulerian Path*/
int main(){
  int n;
  vector<vector<int>> g(n,
vector<int>(n));
  // reading the graph in the
adjacency matrix
  vector<int> deg(n);
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < n;
++j)
      deg[i] += g[i][j];
  }
  int first = 0;
  while (first < n &&
!deg[first])
    ++first;
  if (first == n){
    cout << -1;
    return 0;
  }
  int v1 = -1, v2 = -1;
  bool bad = false;
  for (int i = 0; i < n; ++i){
    if (deg[i] & 1){
      if (v1 == -1) v1 = i;
```

```
      else if (v2 == -1)
        v2 = i;
      else bad = true;
    }
  }
  if (v1 != -1)
    ++g[v1][v2], ++g[v2][v1];
  stack<int> st;
  st.push(first);
  vector<int> res;
  while (!st.empty()){
    int v = st.top();
    int i;
    for (i = 0; i < n; ++i)
      if (g[v][i])
        break;
    if (i == n){
      res.push_back(v);
      st.pop();
    } else {
      --g[v][i];
      --g[i][v];
      st.push(i);
    }
  }
  if (v1 != -1){
    for (size_t i = 0; i + 1 <
res.size(); ++i){
      if ((res[i] == v1 &&
res[i + 1] == v2) ||
        (res[i] == v2 && res[i
+ 1] == v1)){
        vector<int> res2;
        for (size_t j = i + 1;
j < res.size(); ++j)

res2.push_back(res[j]);
        for (size_t j = 1; j
<= i; ++j)

res2.push_back(res[j]);
        res = res2;
        break;
      }
    }
  }
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < n;
++j){
      if (g[i][j])
        bad = true;
    }
  }
  if (bad){
    cout << -1;
  } else {
    for (int x : res)
      cout << x << " ";
  }
}
```

```
/*Edmond's Algorithm*/
```

```
const int M = 500;
struct struct_edge {
    int v;
    struct_edge *n;
};
typedef struct_edge *edge;
struct_edge pool[M * M * 2];
edge top = pool, adj[M];
int V, E, match[M], qh, qt,
q[M], father[M], base[M];
bool inq[M], inb[M], ed[M][M];

void add_edge(int u, int v){
    top->v = v, top->n =
adj[u], adj[u] = top++;
    top->v = u, top->n =
adj[v], adj[v] = top++;
}

int LCA(int root, int u, int
v){
    static bool inp[M];
    memset(inp, 0,
sizeof(inp));
    while (1){
        inp[u = base[u]] =
true;
        if (u == root) break;
        u = father[match[u]];
    }
    while (1){
        if (inp[v = base[v]])
return v;
        else v =
father[match[v]];
    }
}

void mark_blossom(int lca, int
u){
    while (base[u] != lca){
        int v = match[u];
        inb[base[u]] =
inb[base[v]] = true;
        u = father[v];
        if (base[u] != lca)
father[u] = v;
    }
}

void blossom_contraction(int
s, int u, int v){
    int lca = LCA(s, u, v);
    memset(inb, 0,
sizeof(inb));
    mark_blossom(lca, u);
    mark_blossom(lca, v);
    if (base[u] != lca)
        father[u] = v;
    if (base[v] != lca)
        father[v] = u;
    for (int u = 0; u < V;
u++)
        if (inb[base[u]]){
            base[u] = lca;
            if (!inq[u])
                inq[q[++qt] =
u] = true;
            }
}

int find_augmenting_path(int
s){
    memset(inq, 0,
sizeof(inq));
    memset(father, -1,
sizeof(father));
    for (int i = 0; i < V;
i++) base[i] = i;
    inq[q[qh = qt = 0] = s] =
true;
    while (qh <= qt){
        int u = q[qh++];
        for (edge e = adj[u];
e; e = e->n){
            int v = e->v;
            if (base[u] !=
base[v] && match[u] != v)
                if ((v == s)
|| (match[v] != -1 &&
father[match[v]] != -1))

blossom_contraction(s, u, v);
                else if
(father[v] == -1){
                    father[v]
= u;
                    if
(match[v] == -1)
                        return
v;
                    else if
(!inq[match[v]])

inq[q[++qt] = match[v]] =
true;
                }
            }
        }
    return -1;
}

int augment_path(int s, int
t){
    int u = t, v, w;
    while (u != -1){
        v = father[u];
        w = match[v];
        match[v] = u;
        match[u] = v;
        u = w;
    }
    return t != -1;
}

int edmonds(){
    int matchc = 0;
    memset(match, -1,
sizeof(match));
    for (int u = 0; u < V;
u++)
        if (match[u] == -1)
            matchc +=
augment_path(u,
```

```
find_augmenting_path(u));
    return matchc;
}
```

*/*DSU on Tree*/*

```
#define MAX 100005
#define MOD 1000000007
vector<int> *pvec[MAX];
vector<int> Graph[MAX];
int subtree[MAX];
int color[MAX];
int color_counter[MAX];
pair<long long int, int> Info[MAX];

int Subtree(int node, int parent = -1){
  subtree[node] = 1;
  int i;
  for (i = 0; i < Graph[node].size();
i++){
    if (Graph[node][i] == parent)
continue;
    subtree[node] = subtree[node] +
Subtree(Graph[node][i], node);
  }
  return subtree[node];
}

pair<long long int, int> dfs(int node,
int parent = -1, bool keep = false){
  int i, j, k, child, hchild = -1;
  for (i = 0; i < Graph[node].size();
i++){
    if (Graph[node][i] == parent)
continue;
    if (hchild == -1 || subtree[hchild] <
subtree[Graph[node][i]]){
      hchild = Graph[node][i];
    }
  }
  for (i = 0; i < Graph[node].size();
i++){
    if (Graph[node][i] == parent ||
Graph[node][i] == hchild) continue;
    dfs(Graph[node][i], node, false);
  }
  if (hchild != -1){
    Info[node] = dfs(hchild, node,
true);
    pvec[node] = pvec[hchild];
  } else {
    pvec[node] = new vector<int>();
  }
  pvec[node]->push_back(node);
  color_counter[color[node]]++;
  if (color_counter[color[node]] >
Info[node].second){
    Info[node].second =
color_counter[color[node]];
    Info[node].first = color[node];
  } else if (color_counter[color[node]]
== Info[node].second){
    Info[node].first = Info[node].first +
color[node];
  }
  for (i = 0; i < Graph[node].size();
i++){
    if (Graph[node][i] == parent ||
Graph[node][i] == hchild) continue;
    child = Graph[node][i];
    for (j = 0; j < (*pvec[child]).size();
j++){
      k = (*pvec[child])[j];
      pvec[node]->push_back(k);
      color_counter[color[k]]++;
      if (color_counter[color[k]] >
Info[node].second){
        Info[node].second =
color_counter[color[k]];
        Info[node].first = color[k];
      } else if (color_counter[color[k]]
== Info[node].second){
        Info[node].first = Info[node].first
+ color[k];
      }
    }
  }
  if (!keep){
    for (j = 0; j < (*pvec[node]).size();
j++){
      k = (*pvec[node])[j];
      color_counter[color[k]]--;
    }
  }
  return Info[node];
}
```

/*Dynamic Connectivity*/

```
struct dsu_save {
  int v, rnkv, u, rnku;

  dsu_save(){}

  dsu_save(int _v, int _rnkv,
int _u, int _rnku)
      : v(_v), rnkv(_rnkv),
u(_u), rnku(_rnku){}
};

struct dsu_with_rollbacks {
  vector<int> p, rnk;
  int comps;
  stack <dsu_save> op;

  dsu_with_rollbacks(){}

  dsu_with_rollbacks(int n){
    p.resize(n);
    rnk.resize(n);
    for (int i = 0; i < n;
i++){
      p[i] = i;
      rnk[i] = 0;
    }
    comps = n;
  }

  int find_set(int v){
    return (v == p[v]) ? v :
find_set(p[v]);
  }

  bool unite(int v, int u){
    v = find_set(v);
    u = find_set(u);
    if (v == u)
      return false;
    comps--;
    if (rnk[v] > rnk[u])
      swap(v, u);
    op.push(dsu_save(v,
rnk[v], u, rnk[u]));
    p[v] = u;
    if (rnk[u] == rnk[v])
      rnk[u]++;
    return true;
  }

  void rollback(){
    if (op.empty())
      return;
    dsu_save x = op.top();
    op.pop();
    comps++;
    p[x.v] = x.v;
    rnk[x.v] = x.rnkv;
    p[x.u] = x.u;
    rnk[x.u] = x.rnku;
  }
};

struct query {
  int v, u;
  bool united;

  query(int _v, int _u) :
v(_v), u(_u){
  }
};

struct QueryTree {
  vector <vector<query>> t;
```

```
  dsu_with_rollbacks dsu;
  int T;

  QueryTree(){}

  QueryTree(int _T, int n) :
T(_T){
    dsu =
dsu_with_rollbacks(n);
    t.resize(4 * T + 4);
  }

  void add_to_tree(int v, int
l, int r, int ul, int ur,
query &q){
    if (ul > ur)
      return;
    if (l == ul && r == ur){
      t[v].push_back(q);
      return;
    }
    int mid = (l + r) / 2;
    add_to_tree(2 * v, l, mid,
ul, min(ur, mid), q);
    add_to_tree(2 * v + 1, mid
+ 1, r, max(ul, mid + 1), ur,
q);
  }

  void add_query(query q, int
l, int r){
    add_to_tree(1, 0, T - 1,
l, r, q);
  }

  void dfs(int v, int l, int
r, vector<int> &ans){
    for (query &q : t[v]){
      q.united =
dsu.unite(q.v, q.u);
    }
    if (l == r)
      ans[l] = dsu.comps;
    else {
      int mid = (l + r) / 2;
      dfs(2 * v, l, mid, ans);
      dfs(2 * v + 1, mid + 1,
r, ans);
    }
    for (query q : t[v]){
      if (q.united)
        dsu.rollback();
    }
  }

  vector<int> solve(){
    vector<int> ans(T);
    dfs(1, 0, T - 1, ans);
    return ans;
  }
};
```

```
/*Duval*/
```

```
vector <string> duval(string
const &s){
  int n = s.size();
  int i = 0;
  vector<string>
factorization;
  while (i < n){
    int j = i + 1, k = i;
    while (j < n && s[k] <=
s[j]){
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k){

factorization.push_back(s.subs
tr(i, j - k));
      i += j - k;
    }
  }
  return factorization;
}

string
min_cyclic_string(string s){
  s += s;
  int n = s.size();
  int i = 0, ans = 0;
  while (i < n / 2){
    ans = i;
    int j = i + 1, k = i;
    while (j < n && s[k] <=
s[j]){
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k)
      i += j - k;
  }
  return s.substr(ans, n / 2);
}
```

```
/*Divide and Conquer DP*/

int m, n;
vector<long long>
dp_before(n), dp_cur(n);

long long C(int i, int j);

// compute dp_cur[l], ...
dp_cur[r] (inclusive)
void compute(int l, int r, int
optl, int optr){
  if (l > r)
    return;
  int mid = (l + r) >> 1;
  pair<long long, int> best =
{LLONG_MAX, -1};
  for (int k = optl; k <=
min(mid, optr); k++){
    best = min(best, {(k ?
dp_before[k - 1] : 0) + C(k,
mid), k});
  }
  dp_cur[mid] = best.first;
  int opt = best.second;
  compute(l, mid - 1, optl,
opt);
  compute(mid + 1, r, opt,
optr);
}

int solve(){
  for (int i = 0; i < n; i++)
    dp_before[i] = C(0, i);

  for (int i = 1; i < m; i++){
    compute(0, n - 1, 0, n -
1);
    dp_before = dp_cur;
  }
  return dp_before[n - 1];
}
```

```
/*Discrete Log & Root*/
```

```
#define MAX 100000
int prime[MAX + 1], Phi[MAX +
1];

void sieve();

void PhiWithSieve();

int gcd(int a, int b);

int powmod(int a, int b, int
p){
  int res = 1;
  while (b)
    if (b & 1)
      res = int(res * 1ll * a
% p), --b;
    else
      a = int(a * 1ll * a %
p), b >>= 1;
  return res;
}

int PrimitiveRoot(int p){
  vector<int> fact;
  int phi = Phi[p];
  int n = phi;
  while (n > 1){
    if (prime[n] == 0){
      fact.push_back(n);
      n = 1;
    } else {
      int f = prime[n];
      while (n % f == 0){
        n = n / f;
```

```
      }
      fact.push_back(f);
    }
  }
  int res;
  for (res = p - 1; res > 1; --res){
    for (n = 0; n < fact.size(); n++){
      if (powmod(res, phi / fact[n], p) == 1){
        break;
      }
    }
    if (n >= fact.size()) return res;
  }
  return -1;
}

int DiscreteLog(int a, int b, int m){
  a %= m, b %= m;
  int n = sqrt(m) + 1;
  map<int, int> vals;
  for (int p = 1; p <= n; ++p)
    vals[powmod(a, (int) (1ll * p * n) % m, m)] = p;
  for (int q = 0; q <= n; ++q){
    int cur = (powmod(a, q, m) * 1ll * b) % m;
    if (vals.count(cur)){
      int ans = vals[cur] * n - q;
      return ans;
    }
  }
  return -1;
}

vector<int> DiscreteRoot(int n, int a, int k){
  int g = PrimitiveRoot(n);
  vector<int> ans;
  int any_ans = DiscreteLog(powmod(g, k, n), a, n);
  if (any_ans == -1){
    return ans;
  }
  int delta = (n - 1) / gcd(k, n - 1);
  for (int cur = any_ans % delta; cur < n - 1; cur += delta)
    ans.push_back(powmod(g, cur, n));
  sort(ans.begin(), ans.end());
  return ans;
}
```

*/*Dinic's Algorithm*/*

```
struct FlowEdge {
  int v, u;
  long long cap, flow = 0;

  FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap){}
};

struct Dinic {
  const long long flow_inf = 1e18;
  vector <FlowEdge> edges;
  vector <vector<int>> adj;
  int n, m = 0;
  int s, t;
  vector<int> level, ptr;
  queue<int> q;

  Dinic(int n, int s, int t) : n(n), s(s), t(t){
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
  }

  void add_edge(int v, int u, long long cap){
    edges.emplace_back(v, u, cap);
    edges.emplace_back(u, v, 0);
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
  }

  bool bfs(){
    while (!q.empty()){
      int v = q.front();
      q.pop();
      for (int id : adj[v]){
        if (edges[id].cap - edges[id].flow < 1)
          continue;
        if (level[edges[id].u] != -1)
          continue;
        level[edges[id].u] = level[v] + 1;
        q.push(edges[id].u);
      }
    }
    return level[t] != -1;
  }

  long long dfs(int v, long long pushed){
    if (pushed == 0)
      return 0;
    if (v == t)
      return pushed;
    for (int &cid = ptr[v]; cid < (int) adj[v].size(); cid++){
      int id = adj[v][cid];
      int u = edges[id].u;
      if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1) continue;
      long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
      if (tr == 0)
        continue;
      edges[id].flow += tr;
      edges[id ^ 1].flow -= tr;
      return tr;
    }
    return 0;
  }

  long long flow(){
    long long f = 0;
    while (true){
      fill(level.begin(), level.end(), -1);
      level[s] = 0;
      q.push(s);
      if (!bfs())
        break;
      fill(ptr.begin(), ptr.end(), 0);
      while (long long pushed = dfs(s, flow_inf)){
        f += pushed;
      }
    }
    return f;
  }
};
```

*/*Convex Hull*/*

```
struct pt {
  double x, y;
};

bool cmp(pt a, pt b){
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}

bool cw(pt a, pt b, pt c){
  return a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y) < 0;
}

bool ccw(pt a, pt b, pt c){
  return a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y) > 0;
}

vector <pt> a;
vector <pair<double, pair < double, double>> > pp;

void convex_hull(vector < pt > &a){
  if (a.size() == 1)
    return;
```

```
  sort(a.begin(), a.end(),
&cmp);
  pt p1 = a[0], p2 = a.back();
  vector<pt> up, down;
  up.push_back(p1);
  down.push_back(p1);
  for (int i = 1; i < (int)
a.size(); i++){
    if (i == a.size() - 1 ||
cw(p1, a[i], p2)){
      while (up.size() >= 2 &&
!cw(up[up.size() - 2],
up[up.size() - 1], a[i]))
        up.pop_back();
      up.push_back(a[i]);
    }
    if (i == a.size() - 1 ||
ccw(p1, a[i], p2)){
      while (down.size() >= 2
&& !ccw(down[down.size() - 2],
down[down.size() - 1], a[i]))
        down.pop_back();
      down.push_back(a[i]);
    }
  }
  a.clear();
  for (int i = 0; i < (int)
up.size(); i++)
    a.push_back(up[i]);
  for (int i = down.size() -
2; i > 0; i--)
    a.push_back(down[i]);
}

double darea(pt a, pt b, pt
c){
  double dd = a.x * (b.y -
c.y) - a.y * (b.x - c.x) + b.x
* c.y - c.x * b.y;
  if (dd < 0) dd = -dd;
  return dd;
}
```

```
/*Combinatoris*/
#define MAX 100000
#define MOD 1000000007
long long int fact[MAX + 1],
fact_inv[MAX + 1];

//Copy ModInd Related codes
from FFT
void precal() {
    int i;
    fact[0] = fact_inv[0] =
1;
    for (i = 1; i <= MAX;
i++) {
        fact[i] = (fact[i -
1] * i) % MOD;

    }
    i = MAX;
    fact_inv[i] =
ModuloInverse(fact[i], MOD);
    for (i = MAX - 1; i > 0;
i--) {
        fact_inv[i] =
(fact_inv[i + 1] * (i + 1))
% MOD;
    }
}

long long int C(int n, int
r) {
    long long int res =
fact[n];
    res = (res * fact_inv[n
- r]) % MOD;
    res = (res *
fact_inv[r]) % MOD;
    return res;
}
```

```
/*Articulation Point*/
int n; // number of nodes
vector <vector<int>> adj;

void IS_CUTPOINT(int v);

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1){
  visited[v] = true;
  tin[v] = low[v] = timer++;
  int children = 0;
  for (int to : adj[v]){
    if (to == p) continue;
    if (visited[to]){
      low[v] = min(low[v],
tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v],
low[to]);
      if (low[to] >= tin[v] &&
p != -1)
        IS_CUTPOINT(v);
      ++children;
    }
  }
  if (p == -1 && children > 1)
    IS_CUTPOINT(v);
}
void find_cutpoints(){
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i){
    if (!visited[i])
      dfs(i);
  }
}
```

```
/*Articulation Edge*/
int n; // number of nodes
vector <vector<int>> adj;
vector<bool> visited;
vector<int> tin, low;
int timer;

void IS_BRIDGE(int v, int u);

void dfs(int v, int p = -1){
  visited[v] = true;
  tin[v] = low[v] = timer++;
  for (int to : adj[v]){
    if (to == p) continue;
    if (visited[to]){
      low[v] = min(low[v],
tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v],
low[to]);
      if (low[to] > tin[v])
        IS_BRIDGE(v, to);
    }
  }
}

void find_bridges(){
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i){
    if (!visited[i])
      dfs(i);
  }
}
```

```
/*Aho Corasick*/
const int K = 26;
bool found[MAX];

struct Vertex {
  int next[2][K];
  bool leaf = false;
  int p = -1, e = -1;
  char pch;
  int link = -1;
  int go[2][K];

  Vertex(int p = -1, char ch =
'$') : p(p), pch(ch){
    fill(begin(next[0]),
end(next[0]), -1);
    fill(begin(go[0]),
end(go[0]), -1);
    fill(begin(next[1]),
end(next[1]), -1);
    fill(begin(go[1]),
end(go[1]), -1);
  }
};

bool visit[MAX * MAX];
vector <Vertex> t;
vector<int> Indices[MAX *
MAX];

void add_string(int ind, char
*s){
  int v = 0;
  while (*s){
    char ch = *s;
    int c, f;
    if (ch >= 'a'){
      f = 1;
```

```
      c = ch - 'a';
    } else {
      f = 0;
      c = ch - 'A';
    }
    if (t[v].next[f][c] == -
1){
      t[v].next[f][c] =
t.size();
      t.push_back(Vertex(v,
ch));
    }
    v = t[v].next[f][c];
    s++;
  }
  t[v].leaf = true;
  Indices[v].push_back(ind);
}

int go(int v, char ch);

int get_link(int v){
  if (t[v].link == -1){
    if (v == 0 || t[v].p == 0)
      t[v].link = 0;
    else
      t[v].link =
go(get_link(t[v].p),
t[v].pch);
  }
  return t[v].link;
}

int go(int v, char ch){
  int c, f;
  if (ch >= 'a'){
    f = 1;
    c = ch - 'a';
  } else {
    f = 0;
    c = ch - 'A';
  }
  if (t[v].go[f][c] == -1){
    if (t[v].next[f][c] != -1)
      t[v].go[f][c] =
t[v].next[f][c];
    else
      t[v].go[f][c] = v == 0 ?
0 : go(get_link(v), ch);
  }
  return t[v].go[f][c];
}

int find_exit(int v){
  if (t[v].e != -1) return
t[v].e;
  int link = get_link(v);
  if (t[link].leaf || link ==
0) t[v].e = link;
  else t[v].e =
find_exit(link);
  return t[v].e;
}

/*A pair of intersecting
segments*/
```

```
const double EPS = 1E-9;
struct pt {
  double x, y;
};

struct seg {
  pt p, q;
  int id;

  double get_y(double x) const
{
    if (abs(p.x - q.x) < EPS)
      return p.y;
    return p.y + (q.y - p.y) *
(x - p.x) / (q.x - p.x);
  }
};

bool intersect1d(double l1,
double r1, double l2, double
r2){
  if (l1 > r1)
    swap(l1, r1);
  if (l2 > r2)
    swap(l2, r2);
  return max(l1, l2) <=
min(r1, r2) + EPS;
}

int vec(const pt &a, const pt
&b, const pt &c){
  double s = (b.x - a.x) *
(c.y - a.y) - (b.y - a.y) *
(c.x - a.x);
  return abs(s) < EPS ? 0 : s
> 0 ? +1 : -1;
}

bool intersect(const seg &a,
const seg &b){
  return intersect1d(a.p.x,
a.q.x, b.p.x, b.q.x) &&
       intersect1d(a.p.y,
a.q.y, b.p.y, b.q.y) &&
       vec(a.p, a.q, b.p) *
vec(a.p, a.q, b.q) <= 0 &&
       vec(b.p, b.q, a.p) *
vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const seg &a,
const seg &b){
  double x = max(min(a.p.x,
a.q.x), min(b.p.x, b.q.x));
  return a.get_y(x) <
b.get_y(x) - EPS;
}

struct event {
  double x;
  int tp, id;

  event(){}

  event(double x, int tp, int
id) : x(x), tp(tp), id(id){}
```

```
  bool operator<(const event
&e) const {
    if (abs(x - e.x) > EPS)
      return x < e.x;
    return tp > e.tp;
  }
};

set <seg> s;
vector <set<seg>::iterator>
where;

set<seg>::iterator
prev(set<seg>::iterator it){
  return it == s.begin() ?
s.end() : --it;
}

set<seg>::iterator
next(set<seg>::iterator it){
  return ++it;
}

pair<int, int> solve(const
vector <seg> &a){
  int n = (int) a.size();
  vector<event> e;
  for (int i = 0; i < n; ++i){

e.push_back(event(min(a[i].p.x
, a[i].q.x), +1, i));

e.push_back(event(max(a[i].p.x
, a[i].q.x), -1, i));
  }
  sort(e.begin(), e.end());
  s.clear();
  where.resize(a.size());
  for (size_t i = 0; i <
e.size(); ++i){
    int id = e[i].id;
    if (e[i].tp == +1){
      set<seg>::iterator nxt =
s.lower_bound(a[id]), prv =
prev(nxt);
      if (nxt != s.end() &&
intersect(*nxt, a[id])){
        return make_pair(nxt-
>id, id);
      }
      if (prv != s.end() &&
intersect(*prv, a[id])){
        return make_pair(prv-
>id, id);
      }
      where[id] =
s.insert(nxt, a[id]);
    } else {
      set<seg>::iterator nxt =
next(where[id]), prv =
prev(where[id]);
      if (nxt != s.end() &&
prv != s.end() &&
intersect(*nxt, *prv))
        return make_pair(prv-
>id, nxt->id);
      s.erase(where[id]);
```

```
    }
  }
  return make_pair(-1, -1);
}
```

* Pick's Theorem: S=I+B/2-1
* Chinese Remainder Theorem:
x=(Summation of ai*Mi*yi) mod m
where, m= Multiplication of mi, x= ai mod mi,
Mi=m/mi, Miyi = 1 mod mi
* Binomial Coefficients:
- Sum of mCk = (n+1)C(k+1) (For 0<=m<=n)
- Sum of (n+k)Ck=(n+m+1)Cm (For 0<=k<=m)
- Sum of squares of all nCk = (2n)Cn
- Sum of all k(nCk) = n*2^(n-1)
- Sum of all (n-k)Ck = Fib(n+1)
* Catalan Number(General form): Cn=(n+k)Cn – (n+k)C(k-1) where count of X(total n)>=count of Y(total k) in any prefix
* Kirchhoff's theorem: L=D-A; D=degree matrix, A=adjacency matrix, any co-factor in L is the number of spanning trees.
* gcd(F(n),F(m)) = F(gcd(n,m)) where F(n) is nth fib with first two term as 0,1
* a+b=a⨁b+2(a&b)
* a+b=a|b+a&b
* a⨁b=a|b—a&b
* gcd(a,b) = gcd(a,a-b) = gcd(a,a%b)
* gcd(a^m−1, a^n−1) = a^gcd(m,n)−1
* for (int x = m; x;) { --x &= m; … } loops over all subset masks of m (except m itself).
* x & -x is the least bit in x.