12k Club (IME) ACM-ICPC Team Notebook

Contents

1 Flags + Template + vimrc

1.1 Flags

```
g++ -fsanitize=address,undefined -fno-omit-frame-pointer -g -
      Wall -Wshadow -std=c++17 -Wno-unused-result -Wno-sign-
      compare -Wno-char-subscripts
```

1.2 Template

```cpp
#define db(x) cerr << #x << " == " << x << endl
#define dbs(x) cerr << x << endl
#define _ << ", " <<
mt19937_64 llrand((int) chrono::steady_clock::now().
      time_since_epoch().count());
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  //freopen("in", "r", stdin);
  //freopen("out", "w", stdout);
  return 0;
}
```

1.3 vimrc

```
syntax on
set et ts=2 sw=0 sts=-1 ai nu hls cindent
nnoremap ; :
vnoremap ; :
noremap <c-j> 15gj
noremap <c-k> 15gk
nnoremap <s-k> i<CR><ESC>
inoremap ,. <esc>
vnoremap ,. <esc>
nnoremap ,. <esc>
```

2 Data Structures

2.1 Bit Binary Search

```cpp
// --- Bit Binary Search in o(log(n)) ---
const int M = 20
const int N = 1 << M

int lower_bound(int val){
  int ans = 0, sum = 0;
  for(int i = M - 1; i >= 0; i--){
    int x = ans + (1 << i);
    if(sum + bit[x] < val)
      ans = x, sum += bit[x];
  }

  return ans + 1;
}
```

2.2 Bit

```cpp
// Fenwick Tree / Binary Indexed Tree
ll bit[N];

void add(int p, int v) {
  for (p += 2; p < N; p += p & -p) bit[p] += v;
}

ll query(int p) {
  ll r = 0;
  for (p += 2; p; p -= p & -p) r += bit[p];
  return r;
}
```

2.3 Bit 2D

```cpp
// Thank you for the code tfg!

// O(N(logN)^2)
template<class T = int>
struct Bit2D{
  vector<T> ord;
  vector<vector<T>> fw, coord;

  // pts needs all points that will be used in the upd
  // if range upds remember to build with {x1, y1}, {x1, y2 +
       1}, {x2 + 1, y1}, {x2 + 1, y2 + 1}
  Bit2D(vector<pair<T, T>> pts){
    sort(pts.begin(), pts.end());
    for(auto a : pts)
```

```
      if(ord.empty() || a.first != ord.back())
        ord.push_back(a.first);
    fw.resize(ord.size() + 1);
    coord.resize(fw.size());

    for(auto &a : pts)
      swap(a.first, a.second);
    sort(pts.begin(), pts.end());
    for(auto &a : pts){
      swap(a.first, a.second);
      for(int on = std::upper_bound(ord.begin(), ord.end(), a.
          first) - ord.begin(); on < fw.size(); on += on & -on)
        if(coord[on].empty() || coord[on].back() != a.second)
          coord[on].push_back(a.second);
    }

    for(int i = 0; i < fw.size(); i++)
      fw[i].assign(coord[i].size() + 1, 0);
  }

  // point upd
  void upd(T x, T y, T v){
    for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.
        begin(); xx < fw.size(); xx += xx & -xx)
      for(int yy = upper_bound(coord[xx].begin(), coord[xx].end
          (), y) - coord[xx].begin(); yy < fw[xx].size(); yy +=
           yy & -yy)
        fw[xx][yy] += v;
  }

  // point qry
  T qry(T x, T y){
    T ans = 0;
    for(int xx = upper_bound(ord.begin(), ord.end(), x) - ord.
        begin(); xx > 0; xx -= xx & -xx)
      for(int yy = upper_bound(coord[xx].begin(), coord[xx].end
          (), y) - coord[xx].begin(); yy > 0; yy -= yy & -yy)
        ans += fw[xx][yy];
    return ans;
  }

  // range qry
  T qry(T x1, T y1, T x2, T y2){
    return qry(x2, y2) - qry(x2, y1 - 1) - qry(x1 - 1, y2) + qry
        (x1 - 1, y1 - 1);
  }

  // range upd
  void upd(T x1, T y1, T x2, T y2, T v) {
    upd(x1, y1, v);
    upd(x1, y2 + 1, -v);
    upd(x2 + 1, y1, -v);
    upd(x2 + 1, y2 + 1, v);
  }
};
```

2.4 Centroid Decomposition

```
// Centroid decomposition

vector<int> adj[N];
int forb[N], sz[N], par[N];
int n, m;
unordered_map<int, int> dist[N];

void dfs(int u, int p) {
  sz[u] = 1;
  for(int v : adj[u]) {
    if(v != p and !forb[v]) {
      dfs(v, u);
      sz[u] += sz[v];
    }
  }
}

int find_cen(int u, int p, int qt) {
  for(int v : adj[u]) {
    if(v == p or forb[v]) continue;
    if(sz[v] > qt / 2) return find_cen(v, u, qt);
  }
  return u;
}

void getdist(int u, int p, int cen) {
  for(int v : adj[u]) {
    if(v != p and !forb[v]) {
      dist[cen][v] = dist[v][cen] = dist[cen][u] + 1;
      getdist(v, u, cen);
    }
  }
}

void decomp(int u, int p) {
  dfs(u, -1);

  int cen = find_cen(u, -1, sz[u]);
  forb[cen] = 1;
  par[cen] = p;
  dist[cen][cen] = 0;
  getdist(cen, -1, cen);

  for(int v : adj[cen]) if(!forb[v])
    decomp(v, cen);
}

// main
decomp(1, -1);
```

2.5 Heavy-Light Decomposition (Lamarca)

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

#define fr(i,n) for(int i = 0; i<n; i++)
#define all(v) (v).begin(),(v).end()
typedef long long ll;

template<int N> struct Seg{
ll s[4*N], lazy[4*N];
void build(int no = 1, int l = 0, int r = N){
    if(r-l==1){
        s[no] = 0;
        return;
    }
    int mid = (l+r)/2;
    build(2*no,l,mid);
    build(2*no+1,mid,r);
    s[no] = max(s[2*no],s[2*no+1]);
}
Seg(){ //build da HLD tem de ser assim, pq chama sem os
    parametros
  build();
}
void updlazy(int no, int l, int r, ll x){
    s[no] += x;
    lazy[no] += x;
}
void pass(int no, int l, int r){
    int mid = (l+r)/2;
    updlazy(2*no,l,mid,lazy[no]);
    updlazy(2*no+1,mid,r,lazy[no]);
    lazy[no] = 0;
}
void upd(int lup, int rup, ll x, int no = 1, int l = 0, int r =
    N){
    if(rup<=l or r<=lup) return;
    if(lup<=l and r<=rup){
        updlazy(no,l,r,x);
        return;
    }
    pass(no,l,r);
    int mid = (l+r)/2;
    upd(lup,rup,x,2*no,l,mid);
    upd(lup,rup,x,2*no+1,mid,r);
    s[no] = max(s[2*no],s[2*no+1]);
}
ll qry(int lq, int rq, int no = 1, int l = 0, int r = N){
    if(rq<=l or r<=lq) return -LLONG_MAX;
    if(lq<=l and r<=rq){
        return s[no];
    }
    pass(no,l,r);
    int mid = (l+r)/2;
    return max(qry(lq,rq,2*no,l,mid),qry(lq,rq,2*no+1,mid,r));
}
};

template<int N, bool IN_EDGES> struct HLD {
  int t;
  vector<int> g[N];
  int pai[N], sz[N], d[N];
  int root[N], pos[N]; /// vi rpos;
  void ae(int a, int b) { g[a].push_back(b), g[b].push_back(a);
      }
  void dfsSz(int no = 0) {
    if (~pai[no]) g[no].erase(find(all(g[no]),pai[no]));
    sz[no] = 1;
    for(auto &it : g[no]) {
      pai[it] = no; d[it] = d[no]+1;
      dfsSz(it); sz[no] += sz[it];
      if (sz[it] > sz[g[no][0]]) swap(it, g[no][0]);
    }
  }
  void dfsHld(int no = 0) {
    pos[no] = t++; /// rpos.pb(no);
    for(auto &it : g[no]) {
      root[it] = (it == g[no][0] ? root[no] : it);
      dfsHld(it); }
  }
  void init() {
    root[0] = d[0] = t = 0; pai[0] = -1;
    dfsSz(); dfsHld(); }
  Seg<N> tree; //lembrar de ter build da seg sem nada
  template <class Op>
  void processPath(int u, int v, Op op) {
    for (; root[u] != root[v]; v = pai[root[v]]) {
      if (d[root[u]] > d[root[v]]) swap(u, v);
      op(pos[root[v]], pos[v]); }
    if (d[u] > d[v]) swap(u, v);
    op(pos[u]+IN_EDGES, pos[v]);
  }

  /*
  void changeNode(int v, node val){
    tree.upd(pos[v],val);
  }*/
  void modifySubtree(int v, int val) {
    tree.upd(pos[v]+IN_EDGES,pos[v]+sz[v],val);
  }
  ll querySubtree(int v){
    return tree.qry(pos[v]+IN_EDGES,pos[v]+sz[v]);
  }
  void modifyPath(int u, int v, int val) {
    processPath(u,v,[this, &val](int l,int r) {
      tree.upd(l,r+1,val); });
  }
  ll queryPath(int u, int v) { //modificacoes geralmente vem
      aqui (para hld soma)
    ll res = -LLONG_MAX; processPath(u,v,[this,&res](int l,int r
        ) {
      res = max(tree.qry(l,r+1),res); });
    return res;
  }
};
```

2.6 Lichao Tree (ITA)

```
//LiChao Segment Tree
typedef long long ll;
class LiChao {
  vector<ll> m, b;
  int n, sz; ll *x;
#define gx(i) (i < sz ? x[i] : x[sz-1])
  void update(int t, int l, int r, ll nm, ll nb) {
    ll xl = nm * gx(l) + nb, xr = nm * gx(r) + nb;
    ll yl = m[t] * gx(l) + b[t], yr = m[t] * gx(r) + b[t];
      if (yl >= xl && yr >= xr) return;
    if (yl <= xl && yr <= xr) {
      m[t] = nm, b[t] = nb; return;
    }
    int mid = (l + r) / 2;
    update(t<<1, l, mid, nm, nb);
    update(1+(t<<1), mid+1, r, nm, nb);
  }
public:
  LiChao(ll *st, ll *en) : x(st) {
    sz = int(en - st);
    for(n = 1; n < sz; n <<= 1);
    m.assign(2*n, 0); b.assign(2*n, -INF);
  }
  void insert_line(ll nm, ll nb) {
    update(1, 0, n-1, nm, nb);
  }
  ll query(int i) {
    ll ans = -INF;
    for(int t = i+n; t; t >>= 1)
      ans = max(ans, m[t] * x[i] + b[t]);
    return ans;
  }
```

```
};
```

2.7 Merge Sort Tree

```
// Mergesort Tree - Time <O(nlogn), O(log^2n)> - Memory O(nlogn)
// Mergesort Tree is a segment tree that stores the sorted
    subarray
// on each node.
vi st[4*N];

void build(int p, int l, int r) {
  if (l == r) { st[p].pb(s[l]); return; }
  build(2*p, l, (l+r)/2);
  build(2*p+1, (l+r)/2+1, r);
  st[p].resize(r-l+1);
  merge(st[2*p].begin(), st[2*p].end(),
        st[2*p+1].begin(), st[2*p+1].end(),
        st[p].begin());
}

int query(int p, int l, int r, int i, int j, int a, int b) {
  if (j < l or i > r) return 0;
  if (i <= l and j >= r)
    return upper_bound(st[p].begin(), st[p].end(), b) -
           lower_bound(st[p].begin(), st[p].end(), a);
  return query(2*p, l, (l+r)/2, i, j, a, b) +
         query(2*p+1, (l+r)/2+1, r, i, j, a, b);
}
```

2.8 Minimum Queue

```
// O(1) complexity for all operations, except for clear,
// which could be done by creating another deque and using swap

struct MinQueue {
  int plus = 0;
  int sz = 0;
  deque<pair<int, int>> dq;

  bool empty() { return dq.empty(); }
  void clear() { plus = 0; sz = 0; dq.clear(); }
  void add(int x) { plus += x; } // Adds x to every element in
      the queue
  int min() { return dq.front().first + plus; } // Returns the
      minimum element in the queue
  int size() { return sz; }

  void push(int x) {
    x -= plus;
    int amt = 1;
    while (dq.size() and dq.back().first >= x)
      amt += dq.back().second, dq.pop_back();
    dq.push_back({ x, amt });
    sz++;
  }

  void pop() {
    dq.front().second--, sz--;
    if (!dq.front().second) dq.pop_front();
  }
};
```

2.9 Ordered Set

```
#include<bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;

ordered_set s;
s.insert(2), s.insert(3), s.insert(7), s.insert(9);

//find_by_order returns an iterator to the element at a given
    position
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7

//order_of_key returns the position of a given element
cout << s.order_of_key(7) << "\n"; // 2

//If the element does not appear in the set, we get the position
    that the element would have in the set
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

2.10 Dynamic Segment Tree (Lazy)

```
vector<int> e, d, mx, lazy;
//begin creating node 0, then start your segment tree creating
    node 1
int create(){
  mx.push_back(0);
  lazy.push_back(0);
  e.push_back(0);
  d.push_back(0);
  return mx.size() - 1;
}

void push(int pos, int ini, int fim){
  if(pos == 0) return;
  if (lazy[pos]) {
    mx[pos] += lazy[pos];
    // RMQ (max/min)   -> update: = lazy[p],         incr: +=
        lazy[p]
    // RSQ (sum)       -> update: = (r-l+1)*lazy[p], incr: += (r
        -l+1)*lazy[p]
    // Count lights on -> flip:   = (r-l+1)-st[p];
    if (ini != fim) {
      if(e[pos] == 0){
        int aux = create();
        e[pos] = aux;
      }
      if(d[pos] == 0){
        int aux = create();
        d[pos] = aux;
      }
      lazy[e[pos]] += lazy[pos];
      lazy[d[pos]] += lazy[pos];
      // update:    lazy[2*p] = lazy[p],  lazy[2*p+1] = lazy[p];
      // increment: lazy[2*p] += lazy[p], lazy[2*p+1] += lazy[p
          ];
      // flip:      lazy[2*p] ^= 1,       lazy[2*p+1] ^= 1;
    }
    lazy[pos] = 0;
  }
}

void update(int pos, int ini, int fim, int p, int q, int val){
  if(pos == 0) return;

  push(pos, ini, fim);

  if(q < ini || p > fim) return;

  if(p <= ini and fim <= q){
    lazy[pos] += val;
    // update:    lazy[p] = k;
    // increment: lazy[p] += k;
    // flip:      lazy[p] = 1;
    push(pos, ini, fim);
    return;
  }

  int m = (ini + fim) >> 1;
  if(e[pos] == 0){
    int aux = create();
    e[pos] = aux;
  }
  update(e[pos], ini, m, p, q, val);
  if(d[pos] == 0){
    int aux = create();
    d[pos] = aux;
  }
  update(d[pos], m + 1, fim, p, q, val);
  mx[pos] = max(mx[e[pos]], mx[d[pos]]);
}

int query(int pos, int ini, int fim, int p, int q){
  if(pos == 0) return 0;

  push(pos, ini, fim);

  if(q < ini || p > fim) return 0;

  if(p <= ini and fim <= q) return mx[pos];

  int m = (ini + fim) >> 1;
  return max(query(e[pos], ini, m, p, q) , query(d[pos], m + 1,
      fim, p, q));
}
```

2.11 Iterative Segment Tree

```
int n; // Array size
int st[2*N];

int query(int a, int b) {
  a += n; b += n;
  int s = 0;
  while (a <= b) {
    if (a%2 == 1) s += st[a++];
    if (b%2 == 0) s += st[b--];
    a /= 2; b /= 2;
  }
  return s;
}

void update(int p, int val) {
  p += n;
  st[p] += val;
  for (p /= 2; p >= 1; p /= 2)
    st[p] = st[2*p]+st[2*p+1];
}
```

2.12 Persistent Segment Tree

```
vector<int> e, d, sum;
//begin creating node 0, then start your segment tree creating
    node 1
int create(){
    sum.push_back(0);
    e.push_back(0);
    d.push_back(0);
    return sum.size() - 1;
}

int update(int pos, int ini, int fim, int id, int val){
    int novo = create();

    sum[novo] = sum[pos];
    e[novo] = e[pos];
    d[novo] = d[pos];
    pos = novo;

    if(ini == fim){
        sum[pos] = val;
        return novo;
    }

    int m = (ini + fim) >> 1;
    if(id <= m){
        int aux = update(e[pos], ini, m, id, val);
        e[pos] = aux;
    }
    else{
        int aux = update(d[pos], m + 1, fim, id, val);
        d[pos] = aux;
    }

    sum[pos] = sum[e[pos]] + sum[d[pos]];
    return pos;
}

int query(int pos, int ini, int fim, int p, int q){
    if(q < ini || p > fim) return 0;

    if(pos == 0) return 0;

    if(p <= ini and fim <= q) return sum[pos];

    int m = (ini + fim) >> 1;
    return query(e[pos], ini, m, p, q) + query(d[pos], m + 1,
        fim, p, q);
}
```

2.13 Segment Tree 2D

```
// Segment Tree 2D - O(nlog(n)log(n)) of Memory and Runtime
const int N = 1e8+5, M = 2e5+5;
int n, k=1, st[N], lc[N], rc[N];

void addx(int x, int l, int r, int u) {
  if (x < l or r < x) return;

  st[u]++;
  if (l == r) return;

  if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
  addx(x, l, (l+r)/2, lc[u]);
  addx(x, (l+r)/2+1, r, rc[u]);
}

// Adds a point (x, y) to the grid.
void add(int x, int y, int l, int r, int u) {
  if (y < l or r < y) return;

  if (!st[u]) st[u] = ++k;
  addx(x, 1, n, st[u]);

  if (l == r) return;

  if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
  add(x, y, l, (l+r)/2, lc[u]);
  add(x, y, (l+r)/2+1, r, rc[u]);
}

int countx(int x, int l, int r, int u) {
  if (!u or x < l) return 0;
  if (r <= x) return st[u];

  return countx(x, l, (l+r)/2, lc[u]) +
         countx(x, (l+r)/2+1, r, rc[u]);
}

// Counts number of points dominated by (x, y)
// Should be called with l = 1, r = n and u = 1
int count(int x, int y, int l, int r, int u) {
  if (!u or y < l) return 0;
  if (r <= y) return countx(x, 1, n, st[u]);

  return count(x, y, l, (l+r)/2, lc[u]) +
         count(x, y, (l+r)/2+1, r, rc[u]);
}
```

2.14 Set Of Intervals

```
// Set of Intervals
// Use when you have disjoint intervals
#include <bits/stdc++.h>
using namespace std;

const int N = 2e5 + 5;

typedef pair<int, int> pii;
typedef pair<pii, int> piii;

int n, m, x, t;
set<piii> s;

void in(int l, int r, int i) {
  vector<piii> add, rem;
  auto it = s.lower_bound({{l, 0}, 0});
  if(it != s.begin()) it--;
  for(; it != s.end(); it++) {
    int ll = it->first.first;
    int rr = it->first.second;
    int idx = it->second;

    if(ll > r) break;
    if(rr < l) continue;
    if(ll < l) add.push_back({{ll, l-1}, idx});
    if(rr > r) add.push_back({{r+1, rr}, idx});
    rem.push_back(*it);
  }
  add.push_back({{l, r}, i});
  for(auto x : rem) s.erase(x);
  for(auto x : add) s.insert(x);
}
```

2.15 Sparse Table

```
const int N;
const int M; //log2(N)
int sparse[N][M];

void build() {
  for(int i = 0; i < n; i++)
    sparse[i][0] = v[i];

  for(int j = 1; j < M; j++)
    for(int i = 0; i < n; i++)
      sparse[i][j] =
        i + (1 << j - 1) < n
        ? min(sparse[i][j - 1], sparse[i + (1 << j - 1)][j - 1])
        : sparse[i][j - 1];
}

int query(int a, int b){
  int pot = 32 - __builtin_clz(b - a) - 1;
  return min(sparse[a][pot], sparse[b - (1 << pot) + 1][pot]);
}
```

2.16 Sparse Table 2D

```
// 2D Sparse Table - <O(n^2 (log n) ^ 2), O(1)>
const int N = 1e3+1, M = 10;
int t[N][N], v[N][N], dp[M][M][N][N], lg[N], n, m;

void build() {
  int k = 0;
  for(int i=1; i<N; ++i) {
    if (1<<k == i/2) k++;
    lg[i] = k;
  }

  // Set base cases
  for(int x=0; x<n; ++x) for(int y=0; y<m; ++y) dp[0][0][x][y] =
      v[x][y];
  for(int j=1; j<M; ++j) for(int x=0; x<n; ++x) for(int y=0; y
      +(1<<j)<=m; ++y)
    dp[0][j][x][y] = max(dp[0][j-1][x][y], dp[0][j-1][x][y+(1<<j
        -1)]);

  // Calculate sparse table values
  for(int i=1; i<M; ++i) for(int j=0; j<M; ++j)
    for(int x=0; x+(1<<i)<=n; ++x) for(int y=0; y+(1<<j)<=m; ++y
        )
      dp[i][j][x][y] = max(dp[i-1][j][x][y], dp[i-1][j][x+(1<<i
          -1)][y]);
}

int query(int x1, int x2, int y1, int y2) {
  int i = lg[x2-x1+1], j = lg[y2-y1+1];
  int m1 = max(dp[i][j][x1][y1], dp[i][j][x2-(1<<i)+1][y1]);
  int m2 = max(dp[i][j][x1][y2-(1<<j)+1], dp[i][j][x2-(1<<i)+1][
      y2-(1<<j)+1]);
  return max(m1, m2);
}
```

2.17 KD Tree (Stanford)

```
const int maxn=200005;

struct kdtree
{
  int xl,xr,yl,yr,zl,zr,max,flag; // flag=0:x axis  1:y 2:z
} tree[5000005];

int N,M,lastans,xq,yq;
int a[maxn],pre[maxn],nxt[maxn];
int x[maxn],y[maxn],z[maxn],wei[maxn];
int xc[maxn],yc[maxn],zc[maxn],wc[maxn],hash[maxn],biao[maxn];

bool cmp1(int a,int b)
{
  return x[a]<x[b];
}

bool cmp2(int a,int b)
{
  return y[a]<y[b];
}

bool cmp3(int a,int b)
{
  return z[a]<z[b];
}

void makekdtree(int node,int l,int r,int flag)
{
  if (l>r)
  {
    tree[node].max=-maxlongint;
    return;
  }
  int xl=maxlongint,xr=-maxlongint;
  int yl=maxlongint,yr=-maxlongint;
  int zl=maxlongint,zr=-maxlongint,maxc=-maxlongint;
  for (int i=l;i<=r;i++)
    xl=min(xl,x[i]),xr=max(xr,x[i]),
    yl=min(yl,y[i]),yr=max(yr,y[i]),
    zl=min(zl,z[i]),zr=max(zr,z[i]),
    maxc=max(maxc,wei[i]),
    xc[i]=x[i],yc[i]=y[i],zc[i]=z[i],wc[i]=wei[i],biao[i]=i;
  tree[node].flag=flag;
  tree[node].xl=xl,tree[node].xr=xr,tree[node].yl=yl;
  tree[node].yr=yr,tree[node].zl=zl,tree[node].zr=zr;
  tree[node].max=maxc;
  if (l==r) return;
  if (flag==0) sort(biao+l,biao+r+1,cmp1);
  if (flag==1) sort(biao+l,biao+r+1,cmp2);
  if (flag==2) sort(biao+l,biao+r+1,cmp3);
  for (int i=l;i<=r;i++)
    x[i]=xc[biao[i]],y[i]=yc[biao[i]],
    z[i]=zc[biao[i]],wei[i]=wc[biao[i]];
  makekdtree(node*2,l,(l+r)/2,(flag+1)%3);
  makekdtree(node*2+1,(l+r)/2+1,r,(flag+1)%3);
}

int getmax(int node,int xl,int xr,int yl,int yr,int zl,int zr)
{
  xl=max(xl,tree[node].xl);
  xr=min(xr,tree[node].xr);
  yl=max(yl,tree[node].yl);
  yr=min(yr,tree[node].yr);
  zl=max(zl,tree[node].zl);
  zr=min(zr,tree[node].zr);
  if (tree[node].max==-maxlongint) return 0;
  if ((xr<tree[node].xl)||(xl>tree[node].xr)) return 0;
  if ((yr<tree[node].yl)||(yl>tree[node].yr)) return 0;
  if ((zr<tree[node].zl)||(zl>tree[node].zr)) return 0;
  if ((tree[node].xl==xl)&&(tree[node].xr==xr)&&
    (tree[node].yl==yl)&&(tree[node].yr==yr)&&
    (tree[node].zl==zl)&&(tree[node].zr==zr))
  return tree[node].max;
  else
  return max(getmax(node*2,xl,xr,yl,yr,zl,zr),
        getmax(node*2+1,xl,xr,yl,yr,zl,zr));
}

int main()
{
  // N 3D-rect with weights
  // find the maximum weight containing the given 3D-point
  return 0;
}
```

2.18 Treap

```
// Treap (probabilistic BST)
// O(logn) operations (supports lazy propagation)

mt19937_64 llrand(random_device{}());

struct node {
  int val;
  int cnt, rev;
  int mn, mx, mindiff; // value-based treap only!
  ll pri;
  node* l;
  node* r;

  node() {}
  node(int x) : val(x), cnt(1), rev(0), mn(x), mx(x), mindiff(
      INF), pri(llrand()), l(0), r(0) {}
};

struct treap {
  node* root;
```

```
treap() : root(0) {}
~treap() { clear(); }

int cnt(node* t) { return t ? t->cnt : 0; }
int mn (node* t) { return t ? t->mn  : INF; }
int mx (node* t) { return t ? t->mx  : -INF; }
int mindiff(node* t) { return t ? t->mindiff : INF; }

void clear() { del(root); }
void del(node* t) {
  if (!t) return;
  del(t->l); del(t->r);
  delete t;
  t = 0;
}

void push(node* t) {
  if (!t or !t->rev) return;
  swap(t->l, t->r);
  if (t->l) t->l->rev ^= 1;
  if (t->r) t->r->rev ^= 1;
  t->rev = 0;
}

void update(node*& t) {
  if (!t) return;
  t->cnt = cnt(t->l) + cnt(t->r) + 1;
  t->mn = min(t->val, min(mn(t->l), mn(t->r)));
  t->mx = max(t->val, max(mx(t->l), mx(t->r)));
  t->mindiff = min(mn(t->r) - t->val, min(t->val - mx(t->l),
       min(mindiff(t->l), mindiff(t->r))));
}

node* merge(node* l, node* r) {
  push(l); push(r);
  node* t;
  if (!l or !r) t = l ? l : r;
  else if (l->pri > r->pri) l->r = merge(l->r, r), t = l;
  else r->l = merge(l, r->l), t = r;
  update(t);
  return t;
}

// pos: amount of nodes in the left subtree or
// the smallest position of the right subtree in a 0-indexed
     array
pair<node*, node*> split(node* t, int pos) {
  if (!t) return {0, 0};
  push(t);

  if (cnt(t->l) < pos) {
    auto x = split(t->r, pos-cnt(t->l)-1);
    t->r = x.st;
    update(t);
    return { t, x.nd };
  }

  auto x = split(t->l, pos);
  t->l = x.nd;
  update(t);
  return { x.st, t };
}

// Position-based treap
// used when the values are just additional data
// the positions are known when it's built, after that you
// query to get the values at specific positions
// 0-indexed array!
/*
void insert(int pos, int val) {
  push(root);
  node* x = new node(val);
  auto t = split(root, pos);
  root = merge(merge(t.st, x), t.nd);
}

void erase(int pos) {
  auto t1 = split(root, pos);
  auto t2 = split(t1.nd, 1);
  delete t2.st;
  root = merge(t1.st, t2.nd);
}

int get_val(int pos) { return get_val(root, pos); }
int get_val(node* t, int pos) {
  push(t);
  if (cnt(t->l) == pos) return t->val;
  if (cnt(t->l) < pos) return get_val(t->r, pos-cnt(t->l)-1);
  return get_val(t->l, pos);
}
*/
// -------------------

// Value-based treap
// used when the values needs to be ordered
int order(node* t, int val) {
  if (!t) return 0;
  push(t);
  if (t->val < val) return cnt(t->l) + 1 + order(t->r, val);
  return order(t->l, val);
}

bool has(node* t, int val) {
  if (!t) return 0;
  push(t);
  if (t->val == val) return 1;
  return has((t->val > val ? t->l : t->r), val);
}

void insert(int val) {
  if (has(root, val)) return; // avoid repeated values
  push(root);
  node* x = new node(val);
  auto t = split(root, order(root, val));
  root = merge(merge(t.st, x), t.nd);
}

void erase(int val) {
  if (!has(root, val)) return;

  auto t1 = split(root, order(root, val));
  auto t2 = split(t1.nd, 1);
  delete t2.st;
  root = merge(t1.st, t2.nd);
}

// Get the maximum difference between values
int querymax(int i, int j) {
  if (i == j) return -1;
  auto t1 = split(root, j+1);
  auto t2 = split(t1.st, i);

  int ans = mx(t2.nd) - mn(t2.nd);
  root = merge(merge(t2.st, t2.nd), t1.nd);
  return ans;
}

// Get the minimum difference between values
int querymin(int i, int j) {
  if (i == j) return -1;
  auto t2 = split(root, j+1);
  auto t1 = split(t2.st, i);

  int ans = mindiff(t1.nd);
  root = merge(merge(t1.st, t1.nd), t2.nd);
  return ans;
}
// ------------------

void reverse(int l, int r) {
  auto t2 = split(root, r+1);
  auto t1 = split(t2.st, l);
  t1.nd->rev = 1;
  root = merge(merge(t1.st, t1.nd), t2.nd);
}

void print() { print(root); printf("\n"); }
void print(node* t) {
  if (!t) return;
  push(t);
  print(t->l);
  printf("%d ", t->val);
  print(t->r);
}
};
```

2.19 Trie

```
// Trie <O(|S|), O(|S|)>
int trie[N][26], trien = 1;

int add(int u, char c){
  c-='a';
  if (trie[u][c]) return trie[u][c];
  return trie[u][c] = ++trien;
}

//to add a string s in the trie
int u = 1;
for(char c : s) u = add(u, c);
```

2.20 Union Find

```
/*
     ************************************************************************
* DSU (DISJOINT SET UNION / UNION-FIND)
                                                       *
* Time complexity:  Unite - O(alpha n)
                                                        *
*                   Find - O(alpha n)
                                                         *
* Usage: find(node), unite(node1, node2), sz[find(node)]
                                       *
* Notation: par: vector of parents
                                                           *
*           sz:  vector of subsets sizes, i.e. size of the
     subset a node is in    *
************************************************************************
     */

int par[N], sz[N], his[N];
stack <pii> sp, ss;

int find(int a) { return par[a] == a ? a : par[a] = find(par[a])
     ; }

void unite(int a, int b) {
  if ((a = find(a)) == (b = find(b))) return;
  if (sz[a] < sz[b]) swap(a, b);
  par[b] = a; sz[a] += sz[b];
}

//in main
for(int i = 0; i < N; i++) par[i] = i, sz[i] = 1, his[i] = 0;

//Rollback
int find (int a) { return par[a] == a ? a : find(par[a]); }

void unite (int a, int b) {
  if ((a = find(a)) == (b = find(b))) return;
  if (sz[a] < sz[b]) swap(a, b);
  ss.push({a, sz[a]});
  sp.push({b, par[b]});
  sz[a] += sz[b];
  par[b] = a;
}

void rollback() {
  par[sp.top().st] = sp.top().nd; sp.pop();
  sz[ss.top().st]  = ss.top().nd; ss.pop();
}

//Partial Persistence
int t, par[N], sz[N]

int find(int a, int t){
  if(par[a] == a) return a;
  if(his[a] > t) return a;
  return find(par[a], t);
}

void unite(int a, int b){
  if(find(a, t) == find(b, t)) return;
  a = find(a, t), b = find(b, t), t++;
  if(sz[a] < sz[b]) swap(a, b);
  sz[a] += sz[b], par[b] = a, his[b] = t;
}
```

3 Dynamic Programming

3.1 Convex Hull Trick (emaxx)

```
struct Point{
  ll x, y;
  Point(ll x = 0, ll y = 0):x(x), y(y) {}
  Point operator-(Point p){ return Point(x - p.x, y - p.y); }
  Point operator+(Point p){ return Point(x + p.x, y + p.y); }
  Point ccw(){ return Point(-y, x); }
```

```
  ll operator%(Point p){ return x*p.y - y*p.x; }
  ll operator*(Point p){ return x*p.x + y*p.y; }
  bool operator<(Point p) const { return x == p.x ? y < p.y : x
      < p.x; }
};

pair<vector<Point>, vector<Point>> ch(Point *v){
  vector<Point> hull, vecs;
  for(int i = 0; i < n; i++){
    if(hull.size() and hull.back().x == v[i].x) continue;

    while(vecs.size() and vecs.back()*(v[i] - hull.back()) <= 0)
      vecs.pop_back(), hull.pop_back();

    if(hull.size())
      vecs.pb((v[i] - hull.back()).ccw());

    hull.pb(v[i]);
  }
  return {hull, vecs};
}

ll get(ll x) {
    Point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](
        Point a, Point b) {
        return a%b > 0;
    });
    return query*hull[it - vecs.begin()];
}
```

3.2 Divide and Conquer Optimization

```
/*
     **********************************************************************************
* DIVIDE AND CONQUER OPTIMIZATION ( dp[i][k] = min j<k {dp[j][k
    -1] + C(j,i)} )      *
* Description: searches for bounds to optimal point using the
    monotocity condition*
* Condition: L[i][k] <= L[i+1][k]
                                          *
* Time Complexity: O(K*N^2) becomes O(K*N*logN)
                                    *
* Notation: dp[i][k]: optimal solution using k positions, until
    position i          *
*          L[i][k]: optimal point, smallest j which minimizes
    dp[i][k]            *
*          C(i,j): cost for splitting range [j,i] to j and i
                   *
**********************************************************************************
    */

const int N = 1e3+5;

ll dp[N][N];

//Cost for using i and j
ll C(ll i, ll j);

void compute(ll l, ll r, ll k, ll optl, ll optr){
    // stop condition
    if(l > r) return;

    ll mid = (l+r)/2;
    //best : cost, pos
    pair<ll,ll> best = {LINF,-1};

    //searchs best: lower bound to right, upper bound to left
    for(ll i = optl; i <= min(mid, optr); i++){
        best = min(best, {dp[i][k-1] + C(i,mid), i});
    }
    dp[mid][k] = best.first;
    ll opt = best.second;

    compute(l, mid-1, k, optl, opt);
    compute(mid + 1, r, k, opt, optr);
}

//Iterate over k to calculate
ll solve(){
  //dimensions of dp[N][K]
  int n, k;

  //Initialize DP
  for(ll i = 1; i <= n; i++){
    //dp[i,1] = cost from 0 to i
    dp[i][1] = C(0, i);
  }

  for(ll l = 2; l <= k; l++){
    compute(1, n, l, 1, n);
  }

  /*+ Iterate over i to get min{dp[i][k]}, don't forget cost
      from n to i
    for(ll i=1;i<=n;i++){
        ll rest = ;
        ans = min(ans,dp[i][k] + rest);
    }
  */
}
```

3.3 Knuth Optimization

```
// Knuth DP Optimization - O(n^3) -> O(n^2)
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j-1] <= A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[
    i][j]
//
// reference (pt-br): https://algorithmmarch.wordpress.com
    /2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-maratona/
//

// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
int n;
int dp[N][N], a[N][N];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][i] = 0;

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][i] = i;

  for (int j = 2; j <= n; ++j)
    for (int i = j; i >= 1; --i){
      for (int k = a[i][j-1]; k <= a[i+1][j]; ++k) {
        ll v = dp[i][k] + dp[k][j] + cost(i, j);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
      //+ Iterate over i to get min{dp[i][j]} for each j, don't
          forget cost from n to
    }
}


// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
  // ...
}

void knuth() {
  // calculate base cases
  memset(dp, 63, sizeof(dp));
  for (int i = 1; i <= n; i++) dp[i][1] = // ...

  // set initial a[i][j]
  for (int i = 1; i <= n; i++) a[i][1] = 1, a[n+1][i] = n;

  for (int j = 2; j <= maxj; j++)
    for (int i = n; i >= 1; i--){
      for (int k = a[i][j-1]; k <= a[i+1][j]; k++) {
        ll v = dp[k][j-1] + cost(k, i);

        // store the minimum answer for d[i][k]
        // in case of maximum, use v > dp[i][k]
        if (v < dp[i][j])
          a[i][j] = k, dp[i][j] = v;
      }
      //+ Iterate over i to get min{dp[i][j]} for each j, don't
          forget cost from n to
    }
}
```

3.4 Longest Increasing Subsequence

```
// Longest Increasing Subsequence - O(nlogn)
//
// dp(i) = max j<i { dp(j) | a[j] < a[i] } + 1
//

// int dp[N], v[N], n, lis;

memset(dp, 63, sizeof dp);
for (int i = 0; i < n; ++i) {
  // increasing: lower_bound
  // non-decreasing: upper_bound
  int j = lower_bound(dp, dp + lis, v[i]) - dp;
  dp[j] = min(dp[j], v[i]);
  lis = max(lis, j + 1);
}
```

3.5 SOS DP

```
// O(N * 2^N)
// A[i] = initial values
// Calculate F[i] = Sum of A[j] for j subset of i
for(int i = 0; i < (1 << N); i++)
  F[i] = A[i];
for(int i = 0; i < N; i++)
  for(int j = 0; j < (1 << N); j++)
    if(j & (1 << i))
      F[j] += F[j ^ (1 << i)];
```

3.6 Steiner tree

```
// Steiner-Tree O(2^t*n^2 + n*3^t + APSP)

// N - number of nodes
// T - number of terminals
// dist[N][N] - Adjacency matrix
// steiner_tree() = min cost to connect first t nodes, 1-indexed
// dp[i][bit_mask] = min cost to connect nodes active in bitmask
    rooting in i
// min{dp[i][bit_mask]}, i <= n if root doesn't matter

int n, t, dp[N][(1 << T)], dist[N][N];

int steiner_tree() {
  for (int k = 1; k <= n; ++k)
    for (int i = 1; i <= n; ++i)
      for (int j = 1; j <= n; ++j)
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

  for(int i = 1; i <= n; i++)
    for(int j = 0; j < (1 << t); j++)
      dp[i][j] = INF;
  for(int i = 1; i <= t; i++) dp[i][1 << (i-1)] = 0;

  for(int msk = 0; msk < (1 << t); msk++) {
    for(int i = 1; i <= n; i++) {
      for(int ss = msk; ss > 0; ss = (ss - 1) & msk)
        dp[i][msk] = min(dp[i][msk], dp[i][ss] + dp[i][msk - ss
            ]);

      if(dp[i][msk] != INF)
        for(int j = 1; j <= n; j++)
          dp[j][msk] = min(dp[j][msk], dp[i][msk] + dist[i][j]);
    }
  }

  int mn = INF;
  for(int i = 1; i <= n; i++) mn = min(mn, dp[i][(1 << t) - 1]);
  return mn;
}
```

4 Graphs

4.1 2-SAT Kosaraju

```
// Time complexity: O(V+E)

     *

int n, vis[2*N], ord[2*N], ordn, cnt, cmp[2*N], val[N];
vector<int> adj[2*N], adjt[2*N];

// for a variable u with idx i
// u is 2*i and !u is 2*i+1
// (a v b) == !a -> b ^ !b -> a

int v(int x) { return 2*x; }
int nv(int x) { return 2*x+1; }

// add clause (a v b)
void add(int a, int b){
  adj[a^1].push_back(b);
  adj[b^1].push_back(a);
  adjt[b].push_back(a^1);
  adjt[a].push_back(b^1);
}

void dfs(int x){
  vis[x] = 1;
  for(auto v : adj[x]) if(!vis[v]) dfs(v);
  ord[ordn++] = x;
}

void dfst(int x){
  cmp[x] = cnt, vis[x] = 0;
  for(auto v : adjt[x]) if(vis[v]) dfst(v);
}

bool run2sat(){
  for(int i = 1; i <= n; i++) {
    if(!vis[v(i)]) dfs(v(i));
    if(!vis[nv(i)]) dfs(nv(i));
  }
  for(int i = ordn-1; i >= 0; i--)
    if(vis[ord[i]]) cnt++, dfst(ord[i]);
  for(int i = 1; i <= n; i ++){
    if(cmp[v(i)] == cmp[nv(i)]) return false;
    val[i] = cmp[v(i)] > cmp[nv(i)];
  }
  return true;
}

int main () {
    for (int i = 1; i <= n; i++) {
        if (val[i]); // i-th variable is true
        else         // i-th variable is false
}
```

4.2 Shortest Path (Bellman-Ford)

```
//Time complexity: O(VE)
const int N = 1e4+10; // Maximum number of nodes
vector<int> adj[N], adjw[N];
int dist[N], v, w;

memset(dist, 63, sizeof(dist));
dist[0] = 0;
for (int i = 0; i < n-1; ++i)
  for (int u = 0; u < n; ++u)
    for (int j = 0; j < adj[u].size(); ++j)
      v = adj[u][j], w = adjw[u][j],
      dist[v] = min(dist[v], dist[u]+w);
```

4.3 Block Cut

```
// Tarjan for Block Cut Tree (Node Biconnected Componentes) - O(
     n + m)
#define pb push_back
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5+5;

// Regular Tarjan stuff
int n, num[N], low[N], cnt, ch[N], art[N];
vector<int> adj[N], st;

int lb[N]; // Last block that node is contained
int bn; // Number of blocks
vector<int> blc[N]; // List of nodes from block

void dfs(int u, int p) {
  num[u] = low[u] = ++cnt;
  ch[u] = adj[u].size();
  st.pb(u);

  if (adj[u].size() == 1) blc[++bn].pb(u);

  for(int v : adj[u]) {
    if (!num[v]) {
      dfs(v, u), low[u] = min(low[u], low[v]);
      if (low[v] == num[u]) {
        if (p != -1 or ch[u] > 1) art[u] = 1;
        blc[++bn].pb(u);
        while(blc[bn].back() != v)
          blc[bn].pb(st.back()), st.pop_back();
      }
    }
    else if (v != p) low[u] = min(low[u], num[v]), ch[v]--;
  }

  if (low[u] == num[u]) st.pop_back();
}

// Nodes from 1 .. n are blocks
// Nodes from n+1 .. 2*n are articulations
vector<int> bct[2*N]; // Adj list for Block Cut Tree

void build_tree() {
  for(int u=1; u<=n; ++u) for(int v : adj[u]) if (num[u] > num[v
       ]) {
    if (lb[u] == lb[v] or blc[lb[u]][0] == v) /* edge u-v
         belongs to block lb[u] */;
    else { /* edge u-v belongs to block cut tree */;
      int x = (art[u] ? u + n : lb[u]), y = (art[v] ? v + n : lb
           [v]);
      bct[x].pb(y), bct[y].pb(x);
    }
  }
}

void tarjan() {
  for(int u=1; u<=n; ++u) if (!num[u]) dfs(u, -1);
  for(int b=1; b<=bn; ++b) for(int u : blc[b]) lb[u] = b;
  build_tree();
}
```

4.4 Articulation points and bridges

```
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
  low[u] = num[u] = ++cnt;
  for (int v : adj[u]) {
    if (!num[v]) {
      par[v] = u; ch[u]++;
      articulation(v);
      if (low[v] >= num[u]) art[u] = 1;
      if (low[v] >  num[u]) { /* u-v bridge */ }
      low[u] = min(low[u], low[v]);
    }
    else if (v != par[u]) low[u] = min(low[u], num[v]);
  }
}

for (int i = 0; i < n; ++i) if (!num[i])
  articulation(i), art[i] = ch[i]>1;
```

4.5 Dominator Tree

```
// a node u is said to be dominating node v if, from every path
     from the entry point to v you have to pass through u
// so this code is able to find every dominator from a specific
     entry point (usually 1)
// for directed graphs obviously

const int N = 1e5 + 7;

vector<int> adj[N], radj[N], tree[N], bucket[N];
int sdom[N], par[N], dom[N], dsu[N], label[N], arr[N], rev[N],
     cnt;

void dfs(int u) {
  cnt++;
  arr[u] = cnt;
  rev[cnt] = u;
  label[cnt] = cnt;
  sdom[cnt] = cnt;
  dsu[cnt] = cnt;
  for(auto e : adj[u]) {
    if(!arr[e]) {
      dfs(e);
      par[arr[e]] = arr[u];
    }
    radj[arr[e]].push_back(arr[u]);
  }
}

int find(int u, int x = 0) {
  if(u == dsu[u]) {
    return (x ? -1 : u);
  }
  int v = find(dsu[u], x + 1);
  if(v == -1) {
    return u;
  }
  if(sdom[label[dsu[u]]] < sdom[label[u]]) {
    label[u] = label[dsu[u]];
  }
  dsu[u] = v;
  return (x ? v : label[u]);
}

void unite(int u, int v) {
  dsu[v] = u;
}

// in main

dfs(1);
for(int i = cnt; i >= 1; i--) {
  for(auto e : radj[i]) {
    sdom[i] = min(sdom[i], sdom[find(e)]);
  }
  if(i > 1) {
    bucket[sdom[i]].push_back(i);
  }
  for(auto e : bucket[i]) {
    int v = find(e);
    if(sdom[e] == sdom[v]) {
      dom[e] = sdom[e];
    } else {
      dom[e] = v;
    }
  }
  if(i > 1) {
    unite(par[i], i);
  }
}
for(int i = 2; i <= cnt; i++) {
  if(dom[i] != sdom[i]) {
    dom[i] = dom[dom[i]];
  }
  tree[rev[i]].push_back(rev[dom[i]]);
  tree[rev[dom[i]]].push_back(rev[i]);
}
```

4.6 Erdos Gallai

```
// Erdos-Gallai - O(nlogn)
// check if it's possible to create a simple graph (undirected
     edges) from
// a sequence of vertice's degrees
bool gallai(vector<int> v) {
  vector<ll> sum;
  sum.resize(v.size());

  sort(v.begin(), v.end(), greater<int>());
  sum[0] = v[0];
  for (int i = 1; i < v.size(); i++) sum[i] = sum[i-1] + v[i];
  if (sum.back() % 2) return 0;

  for (int k = 1; k < v.size(); k++) {
```

```
    int p = lower_bound(v.begin(), v.end(), k, greater<int>()) -
        v.begin();
    if (p < k) p = k;
    if (sum[k-1] > 1ll*k*(p-1) + sum.back() - sum[p-1]) return
        0;
  }
  return 1;
}
```

4.7 Eulerian Path

```
vector<int> ans, adj[N];
int in[N];

void dfs(int v){
  while(adj[v].size()){
    int x = adj[v].back();
    adj[v].pop_back();
    dfs(x);
  }
  ans.pb(v);
}

// Verify if there is an eulerian path or circuit
vector<int> v;
for(int i = 0; i < n; i++) if(adj[i].size() != in[i]){
  if(abs((int)adj[i].size() - in[i]) != 1) //-> There is no
      valid eulerian circuit/path
  v.pb(i);
}

if(v.size()){
  if(v.size() != 2) //-> There is no valid eulerian path
  if(in[v[0]] > adj[v[0]].size()) swap(v[0], v[1]);
  if(in[v[0]] > adj[v[0]].size()) //-> There is no valid
      eulerian path
  adj[v[1]].pb(v[0]); // Turn the eulerian path into a eulerian
      circuit
}

dfs(0);
for(int i = 0; i < cnt; i++)
  if(adj[i].size()) //-> There is no valid eulerian circuit/path
      in this case because the graph is not conected

ans.pop_back(); // Since it's a curcuit, the first and the last
    are repeated
reverse(ans.begin(), ans.end());

int bg = 0; // Is used to mark where the eulerian path begins
if(v.size()){
  for(int i = 0; i < ans.size(); i++)
    if(ans[i] == v[1] and ans[(i + 1)%ans.size()] == v[0]){
      bg = i + 1;
      break;
    }
}
```

4.8 Fast Kuhn

```
const int N = 1e5+5;

int x, marcB[N], matchB[N], matchA[N], ans, n, m, p;
vector<int> adj[N];

bool dfs(int v){
  for(int i = 0; i < adj[v].size(); i++){
    int viz = adj[v][i];
    if(marcB[viz] == 1 ) continue;
    marcB[viz] = 1;

    if((matchB[viz] == -1) || dfs(matchB[viz])){
      matchB[viz] = v;
      matchA[v] = viz;
      return true;
    }

  }
  return false;
}

int main(){
  //...
  for(int i = 0; i<=n; i++) matchA[i] = -1;
  for(int j = 0; j<=m; j++) matchB[j] = -1;

  bool aux = true;
  while(aux){
    for(int j=1; j<=m; j++) marcB[j] = 0;
    aux = false;
    for(int i=1; i<=n; i++){
      if(matchA[i] != -1) continue;
      if(dfs(i)){
        ans++;
        aux = true;
      }
    }
  }
  //...
}
```

4.9 Find Cycle of size 3 and 4

```
#define N 330000

int n, m;
vector<int> go[N], lk[N];
int w[N], deg[N], pos[N], id[N];

bool circle3() {
  int ans = 0;
  for(int i = 1; i <= n; i++) w[i] = 0;
  for(int x = 1; x <= n; x++)  {
    for(int y : lk[x]) w[y] = 1;
    for(int y : lk[x]) for(int z:lk[y]) if(w[z]) {
      ans=(ans+go[x].size()+go[y].size()+go[z].size() - 6);
      if(ans) return true;
    }
    for(int y:lk[x]) w[y] = 0;
  }
  return false;
}

bool circle4() {
  for(int i = 1; i <= n; i++) w[i] = 0;
  int ans = 0;
  for(int x = 1; x <= n; x++) {
    for(int y:go[x]) for(int z:lk[y]) if(pos[z] > pos[x]) {
      ans = (ans+w[z]);
      w[z]++;
      if(ans) return true;
    }
    for(int y:go[x]) for(int z : lk[y]) w[z] = 0;
  }
  return false;
}

inline bool cmp(const int &x, const int &y) {
  return deg[x] < deg[y];
}

int main() {
  cin >> n >> m;

  int x, y;
  for(int i = 0; i < n; i++) {
    cin >> x >> y;
  }

  for(int i = 1; i <= n; i++) {
    deg[i] = 0, go[i].clear(), lk[i].clear();
  }
  while (m--){
    int a, b;
    cin >> a >> b;
    deg[a]++, deg[b]++;
    go[a].push_back(b);
    go[b].push_back(a);
  }

  for(int i = 1; i <= n; i++) id[i]= i;
  sort(id+1, id+1+n, cmp);
  for(int i = 1; i<= n; i++) pos[id[i]]=i;
  for(int x = 1; x<= n; x++) {
    for(int y:go[x]) {
      if(pos[y]>pos[x]) lk[x].push_back(y);
    }
  };

  //Check circle3() then circle4()
  return 0;
}
```

4.10 Hungarian Navarro

```
// Hungarian - O(n^2 * m)
template<bool is_max = false, class T = int, bool
    is_zero_indexed = false>
struct Hungarian {
  bool swap_coord = false;
  int lines, cols;
  T ans;

  vector<int> pairV, way;
  vector<bool> used;
  vector<T> pu, pv, minv;
  vector<vector<T>> cost;

  Hungarian(int _n, int _m) {
    if (_n > _m) {
      swap(_n, _m);
      swap_coord = true;
    }

    lines = _n + 1, cols = _m + 1;

    clear();
    cost.resize(lines);
    for (auto& line : cost) line.assign(cols, 0);
  }

  void clear() {
    pairV.assign(cols, 0);
    way.assign(cols, 0);
    pv.assign(cols, 0);
    pu.assign(lines, 0);
  }

  void update(int i, int j, T val) {
    if (is_zero_indexed) i++, j++;
    if (is_max) val = -val;
    if (swap_coord) swap(i, j);

    assert(i < lines);
    assert(j < cols);

    cost[i][j] = val;
  }

  T run() {
    T _INF = numeric_limits<T>::max();
    for (int i = 1, j0 = 0; i < lines; i++) {
      pairV[0] = i;
      minv.assign(cols, _INF);
      used.assign(cols, 0);
      do {
        used[j0] = 1;
        int i0 = pairV[j0], j1;
        T delta = _INF;
        for (int j = 1; j < cols; j++) {
          if (used[j]) continue;
          T cur = cost[i0][j] - pu[i0] - pv[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 < cols; j++) {
          if (used[j]) pu[pairV[j]] += delta, pv[j] -= delta;
          else minv[j] -= delta;
        }
        j0 = j1;
      } while (pairV[j0]);

      do {
        int j1 = way[j0];
        pairV[j0] = pairV[j1];
        j0 = j1;
      } while (j0);
    }

    ans = 0;
    for (int j = 1; j < cols; j++) if (pairV[j]) ans += cost[
        pairV[j]][j];

    if (is_max) ans = -ans;
    if (is_zero_indexed) {
      for (int j = 0; j + 1 < cols; j++) pairV[j] = pairV[j +
          1], pairV[j]--;
```

```
      pairV[cols - 1] = -1;
    }
    if (swap_coord) {
      vector<int> pairV_sub(lines, 0);
      for (int j = 0; j < cols; j++) if (pairV[j] >= 0)
        pairV_sub[pairV[j]] = j;
      swap(pairV, pairV_sub);
    }

    return ans;
  }
};

template <bool is_max = false, bool is_zero_indexed = false>
struct HungarianMult : public Hungarian<is_max, long double,
    is_zero_indexed> {
  using super = Hungarian<is_max, long double, is_zero_indexed>;

  HungarianMult(int _n, int _m) : super(_n, _m) {}

  void update(int i, int j, long double x) {
    super::update(i, j, log2(x));
  }
};
```

4.11 Strongly Connected Components

```
//Time complexity: O(V+E)
const int N = 2e5 + 5;

vector<int> adj[N], adjt[N];
int n, ordn, scc_cnt, vis[N], ord[N], scc[N];

//Directed Version
void dfs(int u) {
  vis[u] = 1;
  for (auto v : adj[u]) if (!vis[v]) dfs(v);
  ord[ordn++] = u;
}

void dfst(int u) {
  scc[u] = scc_cnt, vis[u] = 0;
  for (auto v : adjt[u]) if (vis[v]) dfst(v);
}

// add edge: u -> v
void add_edge(int u, int v){
  adj[u].push_back(v);
  adjt[v].push_back(u);
}

//Undirected version:
/*
  int par[N];

  void dfs(int u) {
    vis[u] = 1;
    for (auto v : adj[u]) if(!vis[v]) par[v] = u, dfs(v);
    ord[ordn++] = u;
  }

  void dfst(int u) {
    scc[u] = scc_cnt, vis[u] = 0;
    for (auto v : adj[u]) if(vis[v] and u != par[v]) dfst(v);
  }

  // add edge: u -> v
  void add_edge(int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

*/

// run kosaraju
void kosaraju(){
  for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
  for (int i = ordn - 1; i >= 0; --i) if (vis[ord[i]]) scc_cnt
      ++, dfst(ord[i]);
}
```

4.12 LCA (Max Weight On Path)

```
// Using LCA to find max edge weight between (u, v)
const int N = 1e5+5;  // Max number of vertices
const int K = 20;     // Each 1e3 requires ~ 10 K
const int M = K+5;
int n;                // Number of vertices
vector <pair<int, int>> adj[N];
int vis[N], h[N], anc[N][M], mx[N][M];

void dfs (int u) {
  vis[u] = 1;
  for (auto p : adj[u]) {
    int v = p.st;
    int w = p.nd;
    if (!vis[v]) {
      h[v] = h[u]+1;
      anc[v][0] = u;
      mx[v][0] = w;
      dfs(v);
    }
  }
}

void build () {
  // cl(mn, 63) -- Don't forget to initialize with INF if min
      edge!
  anc[1][0] = 1;
  dfs(1);
  for (int j = 1; j <= K; j++) for (int i = 1; i <= n; i++) {
    anc[i][j] = anc[anc[i][j-1]][j-1];
    mx[i][j] = max(mx[i][j-1], mx[anc[i][j-1]][j-1]);
  }
}

int mxedge (int u, int v) {
  int ans = 0;

  if (h[u] < h[v]) swap(u, v);
  for (int j = K; j >= 0; j--) if (h[anc[u][j]] >= h[v]) {
    ans = max(ans, mx[u][j]);
    u = anc[u][j];
  }
  if (u == v) return ans;
  for (int j = K; j >= 0; j--) if (anc[u][j] != anc[v][j]) {
    ans = max(ans, mx[u][j]);
    ans = max(ans, mx[v][j]);
    u = anc[u][j];
    v = anc[v][j];
  } //LCA: anc[0][u]
  return max({ans, mx[u][0], mx[v][0]});
}
```

4.13 Max Flow

```
// Dinic - O(V^2 * E)
// Bipartite graph or unit flow - O(sqrt(V) * E)
// Small flow - O(F * (V + E))
// USE INF = 1e9!
template <class T = int>
class Dinic {
public:
  struct Edge {
    Edge(int a, T b){to = a;cap = b;}
    int to;
    T cap;
  };

  Dinic(int _n) : n(_n) {
    edges.resize(n);
  }

  T maxFlow(int src, int sink) {
    T ans = 0;
    while(bfs(src, sink)) {
      // maybe random shuffle edges against bad cases?
      T flow;
      pt = std::vector<int>(n, 0);
      while((flow = dfs(src, sink))) {
        ans += flow;
      }
    }
    return ans;
  }

  void addEdge(int from, int to, T cap, T other = 0) {
    edges[from].push_back(list.size());
    list.push_back(Edge(to, cap));
    edges[to].push_back(list.size());
    list.push_back(Edge(from, other));
  }

  bool inCut(int u) const { return h[u] < n; }
  int size() const { return n; }
private:
  int n;
  std::vector<std::vector<int> > edges;
  std::vector<Edge> list;
  std::vector<int> h, pt;

  T dfs(int on, int sink, T flow = 1e9) {
    if(flow == 0) {
      return 0;
    } if(on == sink) {
      return flow;
    }
    for(; pt[on] < (int) edges[on].size(); pt[on]++) {
      int cur = edges[on][pt[on]];
      if(h[on] + 1 != h[list[cur].to]) {
        continue;
      }
      T got = dfs(list[cur].to, sink, std::min(flow, list[cur].
          cap));
      if(got) {
        list[cur].cap -= got;
        list[cur ^ 1].cap += got;
        return got;
      }
    }
    return 0;
  }

  bool bfs(int src, int sink) {
    h = std::vector<int>(n, n);
    h[src] = 0;
    std::queue<int> q;
    q.push(src);
    while(!q.empty()) {
      int on = q.front();
      q.pop();
      for(auto a : edges[on]) {
        if(list[a].cap == 0) {
          continue;
        }
        int to = list[a].to;
        if(h[to] > h[on] + 1) {
          h[to] = h[on] + 1;
          q.push(to);
        }
      }
    }
    return h[sink] < n;
  }
};

/*
      ************************************************************************

* FLOW WITH DEMANDS

      *
*

      *
* 1 - Finding an arbitrary flow

      *
* Assume a network with [L, R] on edges (some may have L = 0),
    let's call it old network.        *
* Create a New Source and New Sink (this will be the src and snk
     for Dinic).                      *
* Modelling Network:

      *
* 1) Every edge from the old network will have cost R - L
                                              *
* 2) Add an edge from New Source to every vertex v with cost:
                                              *
*    Sum(L) for every (u, v). (sum all L that LEAVES v)
                                              *
* 3) Add an edge from every vertex v to New Sink with cost:
                                              *
*    Sum(L) for every (v, w). (sum all L that ARRIVES v)
                                              *
* 4) Add an edge from Old Source to Old Sink with cost INF (
    circulation problem)                      *
* The Network will be valid if and only if the flow saturates
     the network (max flow == sum(L)) *
*
```

```
*
* 2 - Finding Min Flow
*
* To find min flow that satisfies just do a binary search in the
    (Old Sink -> Old Source) edge *
* The cost of this edge represents all the flow from old network
                                                        *
* Min flow = Sum(L) that arrives in Old Sink + flow that leaves
    (Old Sink -> Old Source)      *
****************************************************************************************************
    */
```

4.14 Min Cost Max Flow

```
template <class T = int>
class MCMF {
public:
  struct Edge {
    Edge(int a, T b, T c) : to(a), cap(b), cost(c) {}
    int to;
    T cap, cost;
  };

  MCMF(int size) {
    n = size;
    edges.resize(n);
    pot.assign(n, 0);
    dist.resize(n);
    visit.assign(n, false);
  }

  std::pair<T, T> mcmf(int src, int sink) {
    std::pair<T, T> ans(0, 0);
    if(!SPFA(src, sink)) return ans;
    fixPot();
    // can use dijkstra to speed up depending on the graph
    while(SPFA(src, sink)) {
      auto flow = augment(src, sink);
      ans.first += flow.first;
      ans.second += flow.first * flow.second;
      fixPot();
    }
    return ans;
  }

  void addEdge(int from, int to, T cap, T cost) {
    edges[from].push_back(list.size());
    list.push_back(Edge(to, cap, cost));
    edges[to].push_back(list.size());
    list.push_back(Edge(from, 0, -cost));
  }
private:
  int n;
  std::vector<std::vector<int>> edges;
  std::vector<Edge> list;
  std::vector<int> from;
  std::vector<T> dist, pot;
  std::vector<bool> visit;

  /*bool dij(int src, int sink) {
    T INF = std::numeric_limits<T>::max();
    dist.assign(n, INF);
    from.assign(n, -1);
    visit.assign(n, false);
    dist[src] = 0;
    for(int i = 0; i < n; i++) {
      int best = -1;
      for(int j = 0; j < n; j++) {
        if(visit[j]) continue;
        if(best == -1 || dist[best] > dist[j]) best = j;
      }
      if(dist[best] >= INF) break;
      visit[best] = true;
      for(auto e : edges[best]) {
        auto ed = list[e];
        if(ed.cap == 0) continue;
        T toDist = dist[best] + ed.cost + pot[best] - pot[ed.to
            ];
        assert(toDist >= dist[best]);
        if(toDist < dist[ed.to]) {
          dist[ed.to] = toDist;
          from[ed.to] = e;
        }
      }
    }
    return dist[sink] < INF;
  }*/

  std::pair<T, T> augment(int src, int sink) {
    std::pair<T, T> flow = {list[from[sink]].cap, 0};
    for(int v = sink; v != src; v = list[from[v]^1].to) {
      flow.first = std::min(flow.first, list[from[v]].cap);
      flow.second += list[from[v]].cost;
    }
    for(int v = sink; v != src; v = list[from[v]^1].to) {
      list[from[v]].cap -= flow.first;
      list[from[v]^1].cap += flow.first;
    }
    return flow;
  }

  std::queue<int> q;
  bool SPFA(int src, int sink) {
    T INF = std::numeric_limits<T>::max();
    dist.assign(n, INF);
    from.assign(n, -1);
    q.push(src);
    dist[src] = 0;
    while(!q.empty()) {
      int on = q.front();
      q.pop();
      visit[on] = false;
      for(auto e : edges[on]) {
        auto ed = list[e];
        if(ed.cap == 0) continue;
        T toDist = dist[on] + ed.cost + pot[on] - pot[ed.to];
        if(toDist < dist[ed.to]) {
          dist[ed.to] = toDist;
          from[ed.to] = e;
          if(!visit[ed.to]) {
            visit[ed.to] = true;
            q.push(ed.to);
          }
        }
      }
    }
    return dist[sink] < INF;
  }

  void fixPot() {
    T INF = std::numeric_limits<T>::max();
    for(int i = 0; i < n; i++) {
      if(dist[i] < INF) pot[i] += dist[i];
    }
  }
};
```

4.15 Small to Large

```
// Imagine you have a tree with colored vertices, and you want
    to do some type of query on every subtree about the colors
    inside
// complexity: O(nlogn)

vector<int> adj[N], vec[N];
int sz[N], color[N], cnt[N];

void dfs_size(int v = 1, int p = 0) {
  sz[v] = 1;
  for (auto u : adj[v]) {
    if (u != p) {
      dfs_size(u, v);
      sz[v] += sz[u];
    }
  }
}

void dfs(int v = 1, int p = 0, bool keep = false) {
  int Max = -1, bigchild = -1;
  for (auto u : adj[v]) {
    if (u != p && Max < sz[u]) {
      Max = sz[u];
      bigchild = u;
    }
  }
  for (auto u : adj[v]) {
    if (u != p && u != bigchild) {
      dfs(u, v, 0);
    }
  }
  if (bigchild != -1) {
    dfs(bigchild, v, 1);
    swap(vec[v], vec[bigchild]);
  }
  vec[v].push_back(v);
  cnt[color[v]]++;
  for (auto u : adj[v]) {
    if (u != p && u != bigchild) {
      for (auto x : vec[u]) {
        cnt[color[x]]++;
        vec[v].push_back(x);
      }
    }
  }
  // now here you can do what the query wants
  // there are cnt[c] vertex in subtree v color with c
  if (keep == 0) {
    for (auto u : vec[v]) {
      cnt[color[u]]--;
    }
  }
}
```

4.16 Stoer Wagner (Stanford)

```
// a is a N*N matrix storing the graph we use; a[i][j]=a[j][i]
memset(use,0,sizeof(use));
ans=maxlongint;
for (int i=1;i<N;i++)
{
  memcpy(visit,use,505*sizeof(int));
  memset(reach,0,sizeof(reach));
  memset(last,0,sizeof(last));
  t=0;
  for (int j=1;j<=N;j++)
    if (use[j]==0) {t=j;break;}
  for (int j=1;j<=N;j++)
    if (use[j]==0) reach[j]=a[t][j],last[j]=t;
  visit[t]=1;
  for (int j=1;j<=N-i;j++)
  {
    maxc=maxk=0;
    for (int k=1;k<=N;k++)
      if ((visit[k]==0)&&(reach[k]>maxc)) maxc=reach[k],maxk=k
          ;
    c2=maxk,visit[maxk]=1;
    for (int k=1;k<=N;k++)
      if (visit[k]==0) reach[k]+=a[maxk][k],last[k]=maxk;
  }
  c1=last[c2];
  sum=0;
  for (int j=1;j<=N;j++)
    if (use[j]==0) sum+=a[j][c2];
  ans=min(ans,sum);
  use[c2]=1;
  for (int j=1;j<=N;j++)
    if ((c1!=j)&&(use[j]==0)) {a[j][c1]+=a[j][c2];a[c1][j]=a[j
        ][c1];}
}
```

5 Strings

5.1 Aho-Corasick

```
// Aho-Corasick

// Build: O(sum size of patterns)
// Find total number of matches: O(size of input string)
// Find number of matches for each pattern: O(num of patterns +
    size of input string)

// ids start from 0 by default!

template <int ALPHA_SIZE = 62>
struct Aho {
  struct Node {
    int p, char_p, link = -1, str_idx = -1, nxt[ALPHA_SIZE];
    bool has_end = false;
    Node(int _p = -1, int _char_p = -1) : p(_p), char_p(_char_p)
        {
      fill(nxt, nxt + ALPHA_SIZE, -1);
    }
  };

  vector<Node> nodes = { Node() };
  int ans, cnt = 0;
```

```cpp
  bool build_done = false;
  vector<pair<int, int>> rep;
  vector<int> ord, occur, occur_aux;

  // change this if different alphabet
  int remap(char c) {
    if (islower(c)) return c - 'a';
    if (isalpha(c)) return c - 'A' + 26;
    return c - '0' + 52;
  }

  void add(string &p, int id = -1) {
    int u = 0;
    if (id == -1) id = cnt++;

    for (char ch : p) {
      int c = remap(ch);
      if (nodes[u].nxt[c] == -1) {
        nodes[u].nxt[c] = (int)nodes.size();
        nodes.push_back(Node(u, c));
      }

      u = nodes[u].nxt[c];
    }

    if (nodes[u].str_idx != -1) rep.push_back({ id, nodes[u].
        str_idx });
    else nodes[u].str_idx = id;
    nodes[u].has_end = true;
  }

  void build() {
    build_done = true;
    queue<int> q;

    for (int i = 0; i < ALPHA_SIZE; i++) {
      if (nodes[0].nxt[i] != -1) q.push(nodes[0].nxt[i]);
      else nodes[0].nxt[i] = 0;
    }

    while(q.size()) {
      int u = q.front();
      ord.push_back(u);
      q.pop();

      int j = nodes[nodes[u].p].link;
      if (j == -1) nodes[u].link = 0;
      else nodes[u].link = nodes[j].nxt[nodes[u].char_p];

      nodes[u].has_end |= nodes[nodes[u].link].has_end;

      for (int i = 0; i < ALPHA_SIZE; i++) {
        if (nodes[u].nxt[i] != -1) q.push(nodes[u].nxt[i]);
        else nodes[u].nxt[i] = nodes[nodes[u].link].nxt[i];
      }
    }
  }

  int match(string &s) {
    if (!cnt) return 0;
    if (!build_done) build();

    ans = 0;
    occur = vector<int>(cnt);
    occur_aux = vector<int>(nodes.size());

    int u = 0;
    for (char ch : s) {
      int c = remap(ch);
      u = nodes[u].nxt[c];
      occur_aux[u]++;
    }

    for (int i = (int)ord.size() - 1; i >= 0; i--) {
      int v = ord[i];
      int fv = nodes[v].link;
      occur_aux[fv] += occur_aux[v];
      if (nodes[v].str_idx != -1) {
        occur[nodes[v].str_idx] = occur_aux[v];
        ans += occur_aux[v];
      }
    }

    for (pair<int, int> x : rep) occur[x.first] = occur[x.second
        ];
    return ans;
  }
};
```

5.2 Booths Algorithm

```cpp
// Booth's Algorithm - Find the lexicographically least rotation
     of a string in O(n)

string least_rotation(string s) {
  s += s;
  vector<int> f((int)s.size(), -1);
  int k = 0;
  for (int j = 1; j < (int)s.size(); j++) {
    int i = f[j - k - 1];
    while (i != -1 and s[j] != s[k + i + 1]) {
      if (s[j] < s[k + i + 1]) k = j - i - 1;
      i = f[i];
    }

    if (s[j] != s[k + i + 1]) {
      if (s[j] < s[k]) k = j;
      f[j - k] = -1;
    } else f[j - k] = i + 1;
  }

  return s.substr(k, (int)s.size() / 2);
}
```

5.3 Knuth-Morris-Pratt (Automaton)

```cpp
// KMP Automaton - <O(26*pattern), O(text)>

// max size pattern
const int N = 1e5 + 5;

int cnt, nxt[N+1][26];

void prekmp(string &p) {
  nxt[0][p[0] - 'a'] = 1;
  for(int i = 1, j = 0; i <= p.size(); i++) {
    for(int c = 0; c < 26; c++) nxt[i][c] = nxt[j][c];
    if(i == p.size()) continue;
    nxt[i][p[i] - 'a'] = i+1;
    j = nxt[j][p[i] - 'a'];
  }
}

void kmp(string &s, string &p) {
  for(int i = 0, j = 0; i < s.size(); i++) {
    j = nxt[j][s[i] - 'a'];
    if(j == p.size()) cnt++; //match i - j + 1
  }
}
```

5.4 Knuth-Morris-Pratt

```cpp
// Knuth-Morris-Pratt - String Matching O(n+m)
char s[N], p[N];
int b[N], n, m; // n = strlen(s), m = strlen(p);

void kmppre() {
  b[0] = -1;
  for (int i = 0, j = -1; i < m; b[++i] = ++j)
    while (j >= 0 and p[i] != p[j])
      j = b[j];
}

void kmp() {
  for (int i = 0, j = 0; i < n;) {
    while (j >= 0 and s[i] != p[j]) j=b[j];
    i++, j++;
    if (j == m) {
      // match position i-j
      j = b[j];
    }
  }
}
```

5.5 Manacher

```cpp
// Manacher O(n)

vector<int> d1, d2;

// d1 -> odd : size = 2 * d1[i] - 1, palindrome from i - d1[i] +
     1 to i + d1[i] - 1
// d2 -> even : size = 2 * d2[i], palindrome from i - d2[i] to i
     + d2[i] - 1

void manacher(string &s) {
  int n = s.size();
  d1.resize(n), d2.resize(n);
  for(int i = 0, l1 = 0, l2 = 0, r1 = -1, r2 = -1; i < n; i++) {
    if(i <= r1) {
      d1[i] = min(d1[r1 + l1 - i], r1 - i + 1);
    }
    if(i <= r2) {
      d2[i] = min(d2[r2 + l2 - i + 1], r2 - i + 1);
    }
    while(i - d1[i] >= 0 and i + d1[i] < n and s[i - d1[i]] == s
        [i + d1[i]]) {
      d1[i]++;
    }
    while(i - d2[i] - 1 >= 0 and i + d2[i] < n and s[i - d2[i] -
          1] == s[i + d2[i]]) {
      d2[i]++;
    }
    if(i + d1[i] - 1 > r1) {
      l1 = i - d1[i] + 1;
      r1 = i + d1[i] - 1;
    }
    if(i + d2[i] - 1 > r2) {
      l2 = i - d2[i];
      r2 = i + d2[i] - 1;
    }
  }
}
```

5.6 Recursive-String Matching

```cpp
void p_f(char *s, int *pi) {
  int n = strlen(s);
  pi[0]=pi[1]=0;
  for(int i = 2; i <= n; i++) {
    pi[i] = pi[i-1];
    while(pi[i]>0 and s[pi[i]]!=s[i])
      pi[i]=pi[pi[i]];
    if(s[pi[i]]==s[i-1])
      pi[i]++;
  }
}

int main() {
  //...
  //Initialize prefix function
  char p[N]; //Pattern
  int len = strlen(p); //Pattern size
  int pi[N]; //Prefix function
  p_f(p, pi);

  // Create KMP automaton
  int A[N][128]; //A[i][j]: from state i (size of largest suffix
       of text which is prefix of pattern), append character j
       -> new state A[i][j]
  for( char c : ALPHABET )
    A[0][c] = (p[0] == c);
  for( int i = 1; p[i]; i++ ) {
    for( char c : ALPHABET ) {
      if(c==p[i])
        A[i][c]=i+1; //match
      else
        A[i][c]=A[pi[i]][c]; //try second largest suffix
    }
  }

  //Create KMP "string appending" automaton
  // g_n = g_(n-1) + char(n) + g_(n-1)
  // g_0 = "", g_1 = "a", g_2 = "aba", g_3 = "abacaba", ...
  int F[M][N]; //F[i][j]: from state j (size of largest suffix
      of text which is prefix of pattern), append string g_i ->
       new state F[i][j]
  for(int i = 0; i < m; i++) {
    for(int j = 0; j <= len; j++) {
      if(i==0)
        F[i][j] = j; //append empty string
      else {
        int x = F[i-1][j]; //append g_(i-1)
        x = A[x][j]; //append character j
        x = F[i-1][x]; //append g_(i-1)
        F[i][j] = x;
      }
```

```
    }
  }

  //Create number of matches matrix
  int K[M][N]; //K[i][j]: from state j (size of largest suffix
      of text which is prefix of pattern), append string g_i ->
       K[i][j] matches
  for(int i = 0; i < m; i++) {
    for(int j = 0; j <= len; j++) {
      if(i==0)
        K[i][j] = (j==len); //append empty string
      else {
        int x = F[i-1][j]; //append g_(i-1)
        x = A[x][j]; //append character j

        K[i][j] = K[i-1][j] /*append g_(i-1)*/ + (x==len) /*
              append character j*/ + K[i-1][x]; /*append g_(i-1)
              */
      }
    }
  }
  //number of matches in g_k
  int answer = K[0][k];
  //...
}
```

5.7 String Hashing

```
// String Hashing
// Rabin Karp - O(n + m)

// max size txt + 1
const int N = 1e6 + 5;

// lowercase letters p = 31 (remember to do s[i] - 'a' + 1)
// uppercase and lowercase letters p = 53 (remember to do s[i] -
      'a' + 1)
// any character p = 313

const int MOD = 1e9+9;
ull h[N], p[N];
ull pr = 313; //177771

int cnt;


void build(string &s) {
  p[0] = 1, p[1] = pr;
  for(int i = 1; i <= s.size(); i++) {
    h[i] = ((p[1]*h[i-1]) % MOD + s[i-1]) % MOD;
    p[i] = (p[1]*p[i-1]) % MOD;
  }
}

// 1-indexed
ull fhash(int l, int r) {
  return (h[r] - ((h[l-1]*p[r-l+1]) % MOD) + MOD) % MOD;
}

ull shash(string &pt) {
  ull h = 0;
  for(int i = 0; i < pt.size(); i++)
    h = ((h*pr) % MOD + pt[i]) % MOD;
  return h;
}

void rabin_karp(string &s, string &pt) {
  build(s);
  ull hp = shash(pt);
  for(int i = 0, m = pt.size(); i + m <= s.size(); i++) {
    if(fhash(i+1, i+m) == hp) {
      // match at i
      cnt++;
    }
  }
}
```

5.8 String Multihashing

```
// String Hashing
// Rabin Karp - O(n + m)
template <int N = 3>
struct Hash {
  int hs[N];
  static vector<int> mods;

  static int add(int a, int b, int mod) { return a >= mod - b ?
      a + b - mod : a + b; }
  static int sub(int a, int b, int mod) { return a - b < 0 ? a -
       b + mod : a - b; }
  static int mul(int a, int b, int mod) { return 1ll * a * b %
      mod; }

  Hash(int x = 0) { fill(hs, hs + N, x); }

  bool operator<(const Hash& b) const {
    for (int i = 0; i < N; i++) {
      if (hs[i] < b.hs[i]) return true;
      if (hs[i] > b.hs[i]) return false;
    }
    return false;
  }

  Hash operator+(const Hash& b) const {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = add(hs[i], b.hs[i],
        mods[i]);
    return ans;
  }

  Hash operator-(const Hash& b) const {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = sub(hs[i], b.hs[i],
        mods[i]);
    return ans;
  }

  Hash operator*(const Hash& b) const {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = mul(hs[i], b.hs[i],
        mods[i]);
    return ans;
  }

  Hash operator+(int b) const {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = add(hs[i], b, mods[i
        ]);
    return ans;
  }

  Hash operator*(int b) const {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = mul(hs[i], b, mods[i
        ]);
    return ans;
  }

  friend Hash operator*(int a, const Hash& b) {
    Hash ans;
    for (int i = 0; i < N; i++) ans.hs[i] = mul(b.hs[i], a, b.
        mods[i]);
    return ans;
  }

  friend ostream& operator<<(ostream& os, const Hash& b) {
    for (int i = 0; i < N; i++) os << b.hs[i] << " \n"[i == N -
        1];
    return os;
  }
};

template <int N> vector<int> Hash<N>::mods = { (int) 1e9 + 9, (
     int) 1e9 + 33, (int) 1e9 + 87 };

// In case you need to generate the MODs, uncomment this:
// Obs: you may need this on your template
// mt19937_64 llrand((int) chrono::steady_clock::now().
      time_since_epoch().count());
// In main: gen<>();
/*
template <int N> vector<int> Hash<N>::mods;
template<int N = 3>
void gen() {
  while (Hash<N>::mods.size() < N) {
    int mod;
    bool is_prime;
    do {
      mod = (int) 1e8 + (int) (llrand() % (int) 9e8);
      is_prime = true;
      for (int i = 2; i * i <= mod; i++) {
        if (mod % i == 0) {
          is_prime = false;
          break;
        }
      }
    } while (!is_prime);
    Hash<N>::mods.push_back(mod);
  }
}
*/

template <int N = 3>
struct PolyHash {
  vector<Hash<N>> h, p;

  PolyHash(string& s, int pr = 313) {
    int sz = (int)s.size();
    p.resize(sz + 1);
    h.resize(sz + 1);

    p[0] = 1, h[0] = s[0];
    for (int i = 1; i < sz; i++) {
      h[i] = pr * h[i - 1] + s[i];
      p[i] = pr * p[i - 1];
    }
  }

  Hash<N> fhash(int l, int r) {
    if (!l) return h[r];
    return h[r] - h[l - 1] * p[r - l + 1];
  }

  static Hash<N> shash(string& s, int pr = 313) {
    Hash<N> ans;
    for (int i = 0; i < (int)s.size(); i++) ans = pr * ans + s[i
        ];
    return ans;
  }

  friend int rabin_karp(string& s, string& pt) {
    PolyHash hs = PolyHash(s);
    Hash<N> hp = hs.shash(pt);
    int cnt = 0;
    for (int i = 0, m = (int)pt.size(); i + m <= (int)s.size();
        i++) {
      if (hs.fhash(i, i + m - 1) == hp) {
        // match at i
        cnt++;
      }
    }

    return cnt;
  }
};
```

5.9 Suffix Array

```
// Suffix Array O(nlogn)
// s.push('$');
vector<int> suffix_array(string &s){
  int n = s.size(), alph = 256;
  vector<int> cnt(max(n, alph)), p(n), c(n);

  for(auto c : s) cnt[c]++;
  for(int i = 1; i < alph; i++) cnt[i] += cnt[i - 1];
  for(int i = 0; i < n; i++) p[--cnt[s[i]]] = i;
  for(int i = 1; i < n; i++)
    c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);

  vector<int> c2(n), p2(n);

  for(int k = 0; (1 << k) < n; k++){
    int classes = c[p[n - 1]] + 1;
    fill(cnt.begin(), cnt.begin() + classes, 0);

    for(int i = 0; i < n; i++) p2[i] = (p[i] - (1 << k) + n)%n;
    for(int i = 0; i < n; i++) cnt[c[i]]++;
    for(int i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
    for(int i = n - 1; i >= 0; i--) p[--cnt[c[p2[i]]]] = p2[i];

    c2[p[0]] = 0;
    for(int i = 1; i < n; i++){
      pair<int, int> b1 = {c[p[i]], c[(p[i] + (1 << k))%n]};
      pair<int, int> b2 = {c[p[i - 1]], c[(p[i - 1] + (1 << k))%
          n]};
      c2[p[i]] = c2[p[i - 1]] + (b1 != b2);
    }

    c.swap(c2);
  }
```

```
  return p;
}

// Longest Common Prefix with SA O(n)
vector<int> lcp(string &s, vector<int> &p){
  int n = s.size();
  vector<int> ans(n - 1), pi(n);
  for(int i = 0; i < n; i++) pi[p[i]] = i;

  int lst = 0;
  for(int i = 0; i < n - 1; i++){
    if(pi[i] == n - 1) continue;
    while(s[i + lst] == s[p[pi[i] + 1] + lst]) lst++;

    ans[pi[i]] = lst;
    lst = max(0, lst - 1);
  }

  return ans;
}

// Longest Repeated Substring O(n)
int lrs = 0;
for (int i = 0; i < n; ++i) lrs = max(lrs, lcp[i]);

// Longest Common Substring O(n)
// m = strlen(s);
// strcat(s, "$"); strcat(s, p); strcat(s, "#");
// n = strlen(s);
int lcs = 0;
for (int i = 1; i < n; ++i) if ((sa[i] < m) != (sa[i-1] < m))
  lcs = max(lcs, lcp[i]);

// To calc LCS for multiple texts use a slide window with
     minqueue
// The numver of different substrings of a string is n*(n + 1)/2
      - sum(lcs[i])
```

5.10 Suffix Automaton

```
// Suffix Automaton Construction - O(n)

const int N = 1e6+1, K = 26;
int sl[2*N], len[2*N], sz, last;
ll cnt[2*N];
map<int, int> adj[2*N];

void add(int c) {
  int u = sz++;
  len[u] = len[last] + 1;
  cnt[u] = 1;

  int p = last;
  while(p != -1 and !adj[p][c])
    adj[p][c] = u, p = sl[p];

  if (p == -1) sl[u] = 0;
  else {
    int q = adj[p][c];
    if (len[p] + 1 == len[q]) sl[u] = q;
    else {
      int r = sz++;
      len[r] = len[p] + 1;
      sl[r] = sl[q];
      adj[r] = adj[q];
      while(p != -1 and adj[p][c] == q)
        adj[p][c] = r, p = sl[p];
      sl[q] = sl[u] = r;
    }
  }

  last = u;
}

void clear() {
  for(int i=0; i<=sz; ++i) adj[i].clear();
  last = 0;
  sz = 1;
  sl[0] = -1;
}

void build(char *s) {
  clear();
  for(int i=0; s[i]; ++i) add(s[i]);
}

// Pattern matching - O(|p|)
bool check(char *p) {
  int u = 0, ok = 1;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) ok = 0;
  }
  return ok;
}

// Substring count - O(|p|)
ll d[2*N];

void substr_cnt(int u) {
  d[u] = 1;
  for(auto p : adj[u]) {
    int v = p.second;
    if (!d[v]) substr_cnt(v);
    d[u] += d[v];
  }
}

ll substr_cnt() {
  memset(d, 0, sizeof d);
  substr_cnt(0);
  return d[0] - 1;
}

// k-th Substring - O(|s|)
// Just find the k-th path in the automaton.
// Can be done with the value d calculated in previous problem.

// Smallest cyclic shift - O(|s|)
// Build the automaton for string s + s. And adapt previous dp
// to only count paths with size |s|.


// Number of occurences - O(|p|)
vector<int> t[2*N];

void occur_count(int u) {
  for(int v : t[u]) occur_count(v), cnt[u] += cnt[v];
}

void build_tree() {
  for(int i=1; i<=sz; ++i)
    t[sl[i]].push_back(i);
  occur_count(0);
}

ll occur_count(char *p) {
  // Call build tree once per automaton
  int u = 0;
  for(int i=0; p[i]; ++i) {
    u = adj[u][p[i]];
    if (!u) break;
  }
  return !u ? 0 : cnt[u];
}

// First occurence - (|p|)
// Store the first position of occurence fp.
// Add the the code to add function:
// fp[u] = len[u] - 1;
// fp[r] = fp[q];

// To answer a query, just output fp[u] - strlen(p) + 1
// where u is the state corresponding to string p

// All occurences - O(|p| + |ans|)
// All the occurences can reach the first occurence via suffix
     links.
// So every state that contains a occreunce is reacheable by the
// first occurence state in the suffix link tree. Just do a DFS
     in this
// tree, starting from the first occurence.
// OBS: cloned nodes will output same answer twice.


// Smallest substring not contained in the string - O(|s| * K)
// Just do a dynamic programming:
// d[u] = 1 // if d does not have 1 transition
// d[u] = 1 + min d[v] // otherwise


// LCS of 2 Strings - O(|s| + |t|)
// Build automaton of s and traverse the automaton wih string t
// mantaining the current state and the current lenght.
// When we have a transition: update state, increase lenght by
     one.
// If we don't update state by suffix link and the new lenght
     will
// should be reduced (if bigger) to the new state length.
// Answer will be the maximum length of the whole traversal.

// LCS of n Strings - O(n*|s|*K)
// Create a new string S = s_1 + d1 + ... + s_n + d_n,
// where d_i are delimiters that are unique (d_i != d_j).
// For each state use DP + bitmask to calculate if it can
// reach a d_i transition without going through other d_j.
// The answer will be the biggest len[u] that can reach all
// d_i's.
```

5.11 Suffix Tree

```
// Suffix Tree
// Build: O(|s|)
// Match: O(|p|)

template<int ALPHA_SIZE = 62>
struct SuffixTree {
  struct Node {
    int p, link = -1, l, r, nch = 0;
    vector<int> nxt;
    Node(int _l = 0, int _r = -1, int _p = -1) : p(_p), l(_l), r
        (_r), nxt(ALPHA_SIZE, -1) {}

    int len() { return r - l + 1; }
    int next(char ch) { return nxt[remap(ch)]; }

    // change this if different alphabet
    int remap(char c) {
      if (islower(c)) return c - 'a';
      if (isalpha(c)) return c - 'A' + 26;
      return c - '0' + 52;
    }

    void setEdge(char ch, int nx) {
      int c = remap(ch);
      if (nxt[c] != -1 and nx == -1) nch--;
      else if (nxt[c] == -1 and nx != -1) nch++;
      nxt[c] = nx;
    }
  };

  string s;
  long long num_diff_substr = 0;
  vector<Node> nodes;
  queue<int> leaves;
  pair<int, int> st = { 0, 0 };
  int ls = 0, rs = -1, n;

  int size() { return rs - ls + 1; }

  SuffixTree(string &_s) {
    s = _s;
    // Add this if you want every suffix to be a node
    // s += '$';
    n = (int)s.size();
    nodes.reserve(2 * n + 1);
    nodes.push_back(Node());
    //for (int i = 0; i < n; i++) extend();
  }

  pair<int, int> walk(pair<int, int> _st, int l, int r) {
    int u = _st.first;
    int d = _st.second;

    while (l <= r) {
      if (d == nodes[u].len()) {
        u = nodes[u].next(s[l]), d = 0;
        if (u == -1) return { u, d };
      } else {
        if (s[nodes[u].l + d] != s[l]) return { -1, -1 };
        if (r - l + 1 + d < nodes[u].len()) return { u, r - l +
            1 + d };
        l += nodes[u].len() - d;
        d = nodes[u].len();
      }
    }

    return { u, d };
  }
```

```cpp
int split(pair<int, int> _st) {
  int u = _st.first;
  int d = _st.second;

  if (d == nodes[u].len()) return u;
  if (!d) return nodes[u].p;

  Node& nu = nodes[u];
  int mid = (int)nodes.size();
  nodes.push_back(Node(nu.l, nu.l + d - 1, nu.p));
  nodes[nu.p].setEdge(s[nu.l], mid);
  nodes[mid].setEdge(s[nu.l + d], u);
  nu.p = mid;
  nu.l += d;
  return mid;
}

int getLink(int u) {
  if (nodes[u].link != -1) return nodes[u].link;
  if (nodes[u].p == -1) return 0;
  int to = getLink(nodes[u].p);
  pair<int, int> nst = { to, nodes[to].len() };
  return nodes[u].link = split(walk(nst, nodes[u].l + (nodes[u
       ].p == 0), nodes[u].r));
}

bool match(string &p) {
  int u = 0, d = 0;
  for (char ch : p) {
    if (d == min(nodes[u].r, rs) - nodes[u].l + 1) {
      u = nodes[u].next(ch), d = 1;
      if (u == -1) return false;
    } else {
      if (ch != s[nodes[u].l + d]) return false;
      d++;
    }
  }
  return true;
}

void extend() {
  int mid;
  assert(rs != n - 1);
  rs++;
  num_diff_substr += (int)leaves.size();
  do {
    pair<int, int> nst = walk(st, rs, rs);
    if (nst.first != -1) { st = nst; return; }
    mid = split(st);
    int leaf = (int)nodes.size();
    num_diff_substr++;
    leaves.push(leaf);
    nodes.push_back(Node(rs, n - 1, mid));
    nodes[mid].setEdge(s[rs], leaf);
    int to = getLink(mid);
    st = { to, nodes[to].len() };
  } while (mid);
}

void pop() {
  assert(ls <= rs);
  ls++;
  int leaf = leaves.front();
  leaves.pop();
  Node* nlf = &nodes[leaf];
  while (!nlf->nch) {
    if (st.first != leaf) {
      nodes[nlf->p].setEdge(s[nlf->l], -1);
      num_diff_substr -= min(nlf->r, rs) - nlf->l + 1;
      leaf = nlf->p;
      nlf = &nodes[leaf];
    } else {
      if (st.second != min(nlf->r, rs) - nlf->l + 1) {
        int mid = split(st);
        st.first = mid;
        num_diff_substr -= min(nlf->r, rs) - nlf->l + 1;
        nodes[mid].setEdge(s[nlf->l], -1);
        *nlf = nodes[mid];
        nodes[nlf->p].setEdge(s[nlf->l], leaf);
        nodes.pop_back();
      }
      break;
    }
  }

  if (leaf and !nlf->nch) {
    leaves.push(leaf);
    int to = getLink(nlf->p);
    pair<int, int> nst = { to, nodes[to].len() };
      st = walk(nst, nlf->l + (nlf->p == 0), nlf->r);
      nlf->l = rs - nlf->len() + 1;
      nlf->r = n - 1;
    }
  }
};
```

5.12 Z Function

```cpp
// Z-Function - O(n)

vector<int> zfunction(const string& s){
  vector<int> z (s.size());
  for (int i = 1, l = 0, r = 0, n = s.size(); i < n; i++){
    if (i <= r) z[i] = min(z[i-l], r - i + 1);
    while (i + z[i] < n and s[z[i]] == s[z[i] + i]) z[i]++;
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

6 Mathematics

6.1 Basics

```cpp
// Greatest Common Divisor & Lowest Common Multiple
ll gcd(ll a, ll b) { return b ? gcd(b, a%b) : a; }
ll lcm(ll a, ll b) { return a/gcd(a, b)*b; }

// Multiply caring overflow
ll mulmod(ll a, ll b, ll m = MOD) {
  ll r=0;
  for (a %= m; b; b>>=1, a=(a*2)%m) if (b&1) r=(r+a)%m;
  return r;
}

// Another option for mulmod is using long double
ull mulmod(ull a, ull b, ull m = MOD) {
  ull q = (ld) a * (ld) b / (ld) m;
  ull r = a * b - q * m;
  return (r + m) % m;
}

// Fast exponential
ll fexp(ll a, ll b, ll m = MOD) {
  ll r=1;
  for (a %= m; b; b>>=1, a=(a*a)%m) if (b&1) r=(r*a)%m;
  return r;
}

//cfloor
ll cfloor(ll a, ll b) {
  ll c = abs(a);
  ll d = abs(b);
  if (a * b > 0) return c/d;
  return -(c + d - 1)/d;
}
```

6.2 Advanced

```cpp
/* Line integral = integral(sqrt(1 + (dy/dx)^2)) dx */

/* Multiplicative Inverse over MOD for all 1..N - 1 < MOD in O(N
   )
 Only works for prime MOD. If all 1..MOD - 1 needed, use N = MOD
     */
ll inv[N];
inv[1] = 1;
for(int i = 2; i < N; ++i)
  inv[i] = MOD - (MOD / i) * inv[MOD % i] % MOD;

/* Catalan
 f(n) = sum(f(i) * f(n - i - 1)), i in [0, n - 1] = (2n)! / ((n
      +1)! * n!) = ...
 If you have any function f(n) (there are many) that follows
      this sequence (0-indexed):
 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
      742900, 2674440
 than it's the Catalan function */
ll cat[N];
cat[0] = 1;
for(int i = 1; i + 1 < N; i++) // needs inv[i + 1] till inv[N -
     1]
  cat[i] = 2ll * (2ll * i - 1) * inv[i + 1] % MOD * cat[i - 1] %
       MOD;

/* Floor(n / i), i = [1, n], has <= 2 * sqrt(n) diff values.
 Proof: i = [1, sqrt(n)] has sqrt(n) diff values.
 For i = [sqrt(n), n] we have that 1 <= n / i <= sqrt(n)
 and thus has <= sqrt(n) diff values.
*/
/* l = first number that has floor(N / l) = x
 r = last number that has floor(N / r) = x
 N / r >= floor(N / l)
 r <= N / floor(N / l)*/
for(int l = 1, r; l <= n; l = r + 1){
  r = n / (n / l);
  // floor(n / i) has the same value for l <= i <= r
}

/* Recurrence using matriz
 h[i + 2] = a1 * h[i + 1] + a0 * h[i]
 [h[i] h[i-1]] = [h[1] h[0]] * [a1 1] ^ (i - 1)
                               [a0 0]     */

/* Fibonacci in O(log(N)) with memoization
 f(0) = f(1) = 1
 f(2*k) = f(k)^2 + f(k - 1)^2
 f(2*k + 1) = f(k)*[f(k) + 2*f(k - 1)] */

/* Wilson's Theorem Extension
 B = b1 * b2 * ... * bm (mod n) = +-1, all bi <= n such that gcd
      (bi, n) = 1
 if(n <= 4 or n = (odd prime)^k or n = 2 * (odd prime)^k) B =
      -1; for any k
 else B = 1; */

/* Stirling numbers of the second kind
 S(n, k) = Number of ways to split n numbers into k non-empty
      sets
 S(n, 1) = S(n, n) = 1
 S(n, k) = k * S(n - 1, k) + S(n - 1, k - 1)
 Sr(n, k) = S(n, k) with at least r numbers in each set
 Sr(n, k) = k * Sr(n - 1, k) + (n - 1) * Sr(n - r, k - 1)
        (r - 1)
 S(n - d + 1, k - d + 1) = S(n, k) where if indexes i, j belong
      to the same set, then |i - j| >= d */

/* Burnside's Lemma
 |Classes| = 1 / |G| * sum(K ^ C(g)) for each g in G
 G = Different permutations possible
 C(g) = Number of cycles on the permutation g
 K = Number of states for each element

 Different ways to paint a necklace with N beads and K colors:
 G = {(1, 2, ... N), (2, 3, ... N, 1), ... (N, 1, ... N - 1)}
 gi = (i, i + 1, ... i + N), (taking mod N to get it right) i =
      1 ... N
 i -> 2i -> 3i ..., Cycles in gi all have size n / gcd(i, n), so
       C(gi) = gcd(i, n)
 Ans = 1 / N * sum(K ^ gcd(i, n)), i = 1 ... N
 (For the brave, you can get to Ans = 1 / N * sum(euler_phi(N /
      d) * K ^ d), d | N) */

/* Mobius Inversion
 Sum of gcd(i, j), 1 <= i, j <= N?
 sum(k->N) k * sum(i->N) sum(j->N) [gcd(i, j) == k], i = a * k,
      j = b * k
 = sum(k->N) k * sum(a->N/k) sum(b->N/k) [gcd(a, b) == 1]
 = sum(k->N) k * sum(a->N/k) sum(b->N/k) sum(d->N/k) [d | a] * [
      d | b] * mi(d)
 = sum(k->N) k * sum(d->N/k) mi(d) * floor(N / kd)^2, l = kd, l
      <= N, k | l, d = l / k
 = sum(l->N) floor(N / l)^2 * sum(k|l) k * mi(l / k)
 If f(n) = sum(x|n)(g(x) * h(x)) with g(x) and h(x)
      multiplicative, than f(n) is multiplicative
 Hence, g(l) = sum(k|l) k * mi(l / k) is multiplicative
 = sum(l->N) floor(N / l)^2 * g(l) */

/* Frobenius / Chicken McNugget
n, m given, gcd(n, m) = 1, we want to know if it's possible to
     create N = a * n + b * m
N, a, b >= 0
The greatest number NOT possible is n * m - n - m
We can NOT create (n - 1) * (m - 1) / 2 numbers */
```

6.3 Discrete Log (Baby-step Giant-step)

```
// O(sqrt(m))
// Solve c * a^x = b mod(m) for integer x >= 0.
// Return the smallest x possible, or -1 if there is no solution
// If all solutions needed, solve c * a^x = b mod(m) and (a*b) *
    a^y = b mod(m)
// x + k * (y + 1) for k >= 0 are all solutions
// Works for any integer values of c, a, b and positive m

// Corner Cases:
// 0^x = 1 mod(m) returns x = 0, so you may want to change it to
    -1
// You also may want to change for 0^x = 0 mod(1) to return x =
    1 instead
// We leave it like it is because you might be actually checking
    for m^x = 0^x mod(m)
// which would have x = 0 as the actual solution.
ll discrete_log(ll c, ll a, ll b, ll m){
  c = ((c % m) + m) % m, a = ((a % m) + m) % m, b = ((b % m) + m
    ) % m;
  if(c == b)
    return 0;

  ll g = __gcd(a, m);
  if(b % g) return -1;

  if(g > 1){
    ll r = discrete_log(c * a / g, a, b / g, m / g);
    return r + (r >= 0);
  }

  unordered_map<ll, ll> babystep;
  ll n = 1, an = a % m;

  // set n to the ceil of sqrt(m):
  while(n * n < m) n++, an = (an * a) % m;

  // babysteps:
  ll bstep = b;
  for(ll i = 0; i <= n; i++){
    babystep[bstep] = i;
    bstep = (bstep * a) % m;
  }

  // giantsteps:
  ll gstep = c * an % m;
  for(ll i = 1; i <= n; i++){
    if(babystep.find(gstep) != babystep.end())
      return n * i - babystep[gstep];
    gstep = (gstep * an) % m;
  }
  return -1;
}
```

6.4 Euler Phi

```
// Euler phi (totient)
int ind = 0, pf = primes[0], ans = n;
while (1ll*pf*pf <= n) {
  if (n%pf==0) ans -= ans/pf;
  while (n%pf==0) n /= pf;
  pf = primes[++ind];
}
if (n != 1) ans -= ans/n;

// IME2014
int phi[N];
void totient() {
  for (int i = 1; i < N; ++i)  phi[i]=i;
  for (int i = 2; i < N; i+=2) phi[i]>>=1;
  for (int j = 3; j < N; j+=2) if (phi[j]==j) {
    phi[j]--;
    for (int i = 2*j; i < N; i+=j) phi[i]=phi[i]/j*(j-1);
  }
}
```

6.5 Extended Euclidean and Chinese Remainder

```
// Extended Euclid:
void euclid(ll a, ll b, ll &x, ll &y) {
  if (b) euclid(b, a%b, y, x), y -= x*(a/b);
  else x = 1, y = 0;
}

// find (x, y) such that a*x + b*y = c or return false if it's
    not possible
// [x + k*b/gcd(a, b), y - k*a/gcd(a, b)] are also solutions
bool diof(ll a, ll b, ll c, ll &x, ll &y){
  euclid(abs(a), abs(b), x, y);
  ll g = abs(__gcd(a, b));
  if(c % g) return false;
  x *= c / g;
  y *= c / g;
  if(a < 0) x = -x;
  if(b < 0) y = -y;
  return true;
}

// auxiliar to find_all_solutions
void shift_solution (ll &x, ll &y, ll a, ll b, ll cnt) {
  x += cnt * b;
  y -= cnt * a;
}

// Find the amount of solutions of
// ax + by = c
// in given intervals for x and y
ll find_all_solutions (ll a, ll b, ll c, ll minx, ll maxx, ll
    miny, ll maxy) {
  ll x, y, g = __gcd(a, b);
  if(!diof(a, b, c, x, y)) 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);

  if (lx > rx) return 0;
  return (rx - lx) / abs(b) + 1;
}

bool crt_auxiliar(ll a, ll b, ll m1, ll m2, ll &ans){
  ll x, y;
  if(!diof(m1, m2, b - a, x, y)) return false;
  ll lcm = m1 / __gcd(m1, m2) * m2;
  ans = ((a + x % (lcm / m1) * m1) % lcm + lcm) % lcm;
  return true;
}

// find ans such that ans = a[i] mod b[i] for all 0 <= i < n or
    return false if not possible
// ans + k * lcm(b[i]) are also solutions
bool crt(int n, ll a[], ll b[], ll &ans){
  if(!b[0]) return false;
  ans = a[0] % b[0];
  ll l = b[0];
  for(int i = 1; i < n; i++){
   if(!b[i]) return false;
    if(!crt_auxiliar(ans, a[i] % b[i], l, b[i], ans)) return
        false;
    l *= (b[i] / __gcd(b[i], l));
  }
  return true;
}
```

6.6 Fast Fourier Transform(Tourist)

```
//
//  FFT made by tourist. It if faster and more supportive,
    although it requires more lines of code.
//  Also, it allows operations with MOD, which is usually an
    issue in FFT problems.
//
namespace fft {
  typedef double dbl;

  struct num {
    dbl x, y;
    num() { x = y = 0; }
    num(dbl x, dbl y) : x(x), y(y) {}
  };

  inline num operator+ (num a, num b) { return num(a.x + b.x, a.
      y + b.y); }
  inline num operator- (num a, num b) { return num(a.x - b.x, a.
      y - b.y); }
  inline num operator* (num a, num b) { return num(a.x * b.x - a
      .y * b.y, a.x * b.y + a.y * b.x); }
  inline num conj(num a) { return num(a.x, -a.y); }

  int base = 1;
  vector<num> roots = {{0, 0}, {1, 0}};
  vector<int> rev = {0, 1};

  const dbl PI = acosl(-1.0);

  void ensure_base(int nbase) {
    if(nbase <= base) return;

    rev.resize(1 << nbase);
    for(int i=0; i < (1 << nbase); i++) {
      rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
    }
    roots.resize(1 << nbase);

    while(base < nbase) {
      dbl angle = 2*PI / (1 << (base + 1));
      for(int i = 1 << (base - 1); i < (1 << base); i++) {
        roots[i << 1] = roots[i];
        dbl angle_i = angle * (2 * i + 1 - (1 << base));
        roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
      }
      base++;
    }
  }

  void fft(vector<num> &a, int n = -1) {
    if(n == -1) {
      n = a.size();
    }
    assert((n & (n-1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for(int i = 0; i < n; i++) {
      if(i < (rev[i] >> shift)) {
        swap(a[i], a[rev[i] >> shift]);
      }
    }
    for(int k = 1; k < n; k <<= 1) {
      for(int i = 0; i < n; i += 2 * k) {
        for(int j = 0; j < k; j++) {
          num z = a[i+j+k] * roots[j+k];
          a[i+j+k] = a[i+j] - z;
          a[i+j] = a[i+j] + z;
        }
      }
    }
  }

  vector<num> fa, fb;
  vector<int> multiply(vector<int> &a, vector<int> &b) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for(int i = 0; i < sz; i++) {
      int x = (i < (int) a.size() ? a[i] : 0);
      int y = (i < (int) b.size() ? b[i] : 0);
```

```
      fa[i] = num(x, y);
    }
    fft(fa, sz);
    num r(0, -0.25 / sz);
    for(int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
      if(i != j) {
        fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
      }
      fa[i] = z;
    }
    fft(fa, sz);
    vector<int> res(need);
    for(int i = 0; i < need; i++) {
      res[i] = fa[i].x + 0.5;
    }
    return res;
  }

  vector<int> multiply_mod(vector<int> &a, vector<int> &b, int m
      , int eq = 0) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while ((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if (sz > (int) fa.size()) {
      fa.resize(sz);
    }
    for (int i = 0; i < (int) a.size(); i++) {
      int x = (a[i] % m + m) % m;
      fa[i] = num(x & ((1 << 15) - 1), x >> 15);
    }
    fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
    fft(fa, sz);
    if (sz > (int) fb.size()) {
      fb.resize(sz);
    }
    if (eq) {
      copy(fa.begin(), fa.begin() + sz, fb.begin());
    } else {
      for (int i = 0; i < (int) b.size(); i++) {
        int x = (b[i] % m + m) % m;
        fb[i] = num(x & ((1 << 15) - 1), x >> 15);
      }
      fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
      fft(fb, sz);
    }
    dbl ratio = 0.25 / sz;
    num r2(0, -1);
    num r3(ratio, 0);
    num r4(0, -ratio);
    num r5(0, 1);
    for (int i = 0; i <= (sz >> 1); i++) {
      int j = (sz - i) & (sz - 1);
      num a1 = (fa[i] + conj(fa[j]));
      num a2 = (fa[i] - conj(fa[j])) * r2;
      num b1 = (fb[i] + conj(fb[j])) * r3;
      num b2 = (fb[i] - conj(fb[j])) * r4;
      if (i != j) {
        num c1 = (fa[j] + conj(fa[i]));
        num c2 = (fa[j] - conj(fa[i])) * r2;
        num d1 = (fb[j] + conj(fb[i])) * r3;
        num d2 = (fb[j] - conj(fb[i])) * r4;
        fa[i] = c1 * d1 + c2 * d2 * r5;
        fb[i] = c1 * d2 + c2 * d1;
      }
      fa[j] = a1 * b1 + a2 * b2 * r5;
      fb[j] = a1 * b2 + a2 * b1;
    }
    fft(fa, sz);
    fft(fb, sz);
    vector<int> res(need);
    for (int i = 0; i < need; i++) {
      long long aa = fa[i].x + 0.5;
      long long bb = fb[i].x + 0.5;
      long long cc = fa[i].y + 0.5;
      res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
    }
    return res;
  }

  vector<int> square_mod(vector<int> &a, int m) {
    return multiply_mod(a, a, m, 1);
  }
}
```

6.7 Fast Walsh-Hadamard Transform

```
// Fast Walsh-Hadamard Transform - O(nlogn)
//
// Multiply two polynomials, but instead of x^a * x^b = x^(a+b)
// we have x^a * x^b = x^(a XOR b).
//
// WARNING: assert n is a power of two!
void fwht(ll* a, int n, bool inv) {
  for(int l=1; 2*l <= n; l<<=1) {
    for(int i=0; i < n; i+=2*l) {
      for(int j=0; j<l; j++) {
        ll u = a[i+j], v = a[i+l+j];

        a[i+j] = (u+v) % MOD;
        a[i+l+j] = (u-v+MOD) % MOD;
        // % is kinda slow, you can use add() macro instead
        // #define add(x,y) (x+y >= MOD ? x+y-MOD : x+y)
      }
    }
  }

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


/* FWHT AND
  Matrix : Inverse
  0 1      -1 1
  1 1       1 0
*/
void fwht_and(vi &a, bool inv) {
  vi ret = a;
  ll u, v;
  int tam = a.size() / 2;
  for(int len = 1; 2 * len <= tam; len <<= 1) {
    for(int i = 0; i < tam; i += 2 * len) {
      for(int j = 0; j < len; j++) {
        u = ret[i + j];
        v = ret[i + len + j];
        if(!inv) {
          ret[i + j] = v;
          ret[i + len + j] = u + v;
        }
        else {
          ret[i + j] = -u + v;
          ret[i + len + j] = u;
        }
      }
    }
  }
  a = ret;
}


/* FWHT OR
  Matrix : Inverse
  1 1      0  1
  1 0      1 -1
*/
void fft_or(vi &a, bool inv) {
  vi ret = a;
  ll u, v;
  int tam = a.size() / 2;
  for(int len = 1; 2 * len <= tam; len <<= 1) {
    for(int i = 0; i < tam; i += 2 * len) {
      for(int j = 0; j < len; j++) {
        u = ret[i + j];
        v = ret[i + len + j];
        if(!inv) {
          ret[i + j] = u + v;
          ret[i + len + j] = u;
        }
        else {
          ret[i + j] = v;
          ret[i + len + j] = u - v;
        }
      }
    }
  }
  a = ret;
}
```

6.8 Gaussian Elimination (xor)

```
// Gauss Elimination for xor boolean operations
// Return false if not possible to solve
// Use boolean matrixes 0-indexed
// n equations, m variables, O(n * m * m)
// eq[i][j] = coefficient of j-th element in i-th equation
// r[i] = result of i-th equation
// Return ans[j] = xj that gives the lexicographically greatest
      solution (if possible)
// (Can be changed to lexicographically least, follow the
      comments in the code)
// WARNING!! The arrays get changed during de algorithm

bool eq[N][M], r[N], ans[M];

bool gauss_xor(int n, int m){
  for(int i = 0; i < m; i++)
    ans[i] = true;
  int lid[N] = {0}; // id + 1 of last element present in i-th
      line of final matrix
  int l = 0;
  for(int i = m - 1; i >= 0; i--){
    for(int j = l; j < n; j++)
      if(eq[j][i]){ // pivot
        swap(eq[l], eq[j]);
        swap(r[l], r[j]);
      }
    if(l == n || !eq[l][i])
      continue;
    lid[l] = i + 1;
    for(int j = l + 1; j < n; j++){ // eliminate column
      if(!eq[j][i])
        continue;
      for(int k = 0; k <= i; k++)
        eq[j][k] ^= eq[l][k];
      r[j] ^= r[l];
    }
    l++;
  }
  for(int i = n - 1; i >= 0; i--){ // solve triangular matrix
    for(int j = 0; j < lid[i + 1]; j++)
      r[i] ^= (eq[i][j] && ans[j]);
    // for lexicographically least just delete the for bellow
    for(int j = lid[i + 1]; j + 1 < lid[i]; j++){
      ans[j] = true;
      r[i] ^= eq[i][j];
    }
    if(lid[i])
      ans[lid[i] - 1] = r[i];
    else if(r[i])
      return false;
  }
  return true;
}
```

6.9 Gaussian Elimination (double)

```
//Gaussian Elimination
//double A[N][M+1], X[M]

// if n < m, there's no solution
// column m holds the right side of the equation
// X holds the solutions

for(int j=0; j<m; j++) { //collumn to eliminate
  int l = j;
  for(int i=j+1; i<n; i++) //find largest pivot
    if(abs(A[i][j])>abs(A[l][j]))
      l=i;
  if(abs(A[l][j]) < EPS) continue;
  for(int k = 0; k < m+1; k++) { //Swap lines
    swap(A[l][k],A[j][k]);
  }
  for(int i = j+1; i < n; i++) { //eliminate column
    double t=A[i][j]/A[j][j];
    for(int k = j; k < m+1; k++)
      A[i][k]-=t*A[j][k];
  }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
  for(int j = m-1; j > i; j--)
    A[i][m] -= A[i][j]*X[j];
  X[i]=A[i][m]/A[i][i];
}
```

6.10 Ternary Search

```
//Ternary Search - O(log(n))
//Max version, for minimum version just change signals
//Faster version - 300 iteratons up to 1e-6 precision
//For integers do (r - l > 3) and beware of boundaries
double ternary_search(double l, double r, int No = 300){
  // for(int i = 0; i < No; i++){
  while(r - l > EPS){
    double m1 = l + (r - l) / 3;
    double m2 = r - (r - l) / 3;
    // if (f(m1) > f(m2))
    if (f(m1) < f(m2))
      l = m1;
    else
      r = m2;
  }
  return f(l);
}
```

6.11 Golden Section Search (Ternary Search)

```
double gss(double l, double r) {
  double m1 = r-(r-l)/gr, m2 = l+(r-l)/gr;
  double f1 = f(m1), f2 = f(m2);
  while(fabs(l-r)>EPS) {
    if(f1>f2) l=m1, f1=f2, m1=m2, m2=l+(r-l)/gr, f2=f(m2);
    else r=m2, f2=f1, m2=m1, m1=r-(r-l)/gr, f1=f(m1);
  }
  return l;
}
```

6.12 Josephus

```
// UFMG
/* Josephus Problem - It returns the position to be, in order to
     not die. O(n)*/
/* With k=2, for instance, the game begins with 2 being killed
     and then n+2, n+4, ... */
ll josephus(ll n, ll k) {
  if(n==1) return 1;
  else return (josephus(n-1, k)+k-1)%n+1;
}

/* Another Way to compute the last position to be killed - O(d *
     log n) */
ll josephus(ll n, ll d) {
  ll K = 1;
  while (K <= (d - 1)*n) K = (d * K + d - 2) / (d - 1);
  return d * n + 1 - K;
}
```

6.13 Mobius Inversion

```
// multiplicative function calculator
// euler_phi and mobius are multiplicative
// if another f[N] needed just remove comments
// O(N)

bool p[N];
vector<ll> primes;
ll g[N];
// ll f[N];

void mfc(){
  // if g(1) != 1 than it's not multiplicative
  g[1] = 1;
  // f[1] = 1;
  primes.clear();
  primes.reserve(N / 10);
  for(ll i = 2; i < N; i++){
    if(!p[i]){
      primes.push_back(i);
      for(ll j = i; j < N; j *= i){
        g[j] = // g(p^k) you found
        // f[j] = f(p^k) you found
        p[j] = (j != i);
      }
    }
    for(ll j : primes){
      if(i * j >= N || i % j == 0)
        break;
      for(ll k = j; i * k < N; k *= j){
        g[i * k] = g[i] * g[k];
        // f[i * k] = f[i] * f[k];
        p[i * k] = true;
      }
    }
  }
}
```

6.14 Mobius Function

```
// 1 if n == 1
// 0 if exists x | n%(x^2) == 0
// else (-1)^k, k = #(p) | p is prime and n%p == 0

//Calculate Mobius for all integers using sieve
//O(n*log(log(n)))
void mobius() {
  for(int i = 1; i < N; i++) mob[i] = 1;

  for(ll i = 2; i < N; i++) if(!sieve[i]){
    for(ll j = i; j < N; j += i) sieve[j] = i, mob[j] *= -1;
    for(ll j = i*i; j < N; j += i*i) mob[j] = 0;
  }
}

/*
//Calculate Mobius for 1 integer
//O(sqrt(n))
int mobius(int n){
  if(n == 1) return 1;
  int p = 0;
  for(int i = 2; i*i <= n; i++)
    if(n%i == 0){
      n /= i;
      p++;
      if(n%i == 0) return 0;
    }
  if(n > 1) p++;
  return p&1 ? -1 : 1;
}
*/
```

6.15 Number Theoretic Transform

```
// copy fexp from basics.cpp

const int MOD = 998244353;
const int me = 15;
const int ms = 1 << me;

#define add(x, y) x+y>=MOD?x+y-MOD:x+y

const int gen = 3; // use search() from PrimitiveRoot.cpp if MOD
     isn't 998244353
int bits[ms], root[ms];

void initFFT() {
  root[1] = 1;
  for(int len = 2; len < ms; len += len) {
    int z = (int) fexp(gen, (MOD - 1) / len / 2);
    for(int i = len / 2; i < len; i++) {
      root[2 * i] = root[i];
      root[2 * i + 1] = (int)((long long) root[i] * z % MOD);
    }
  }
}

void pre(int n) {
  int LOG = 0;
  while(1 << (LOG + 1) < n) {
    LOG++;
  }
  for(int i = 1; i < n; i++) {
    bits[i] = (bits[i >> 1] >> 1) | ((i & 1) << LOG);
  }
}

std::vector<int> fft(std::vector<int> a, bool inv = false) {
  int n = (int) a.size();
  pre(n);
  if(inv) {
    std::reverse(a.begin() + 1, a.end());
  }
  for(int i = 0; i < n; i++) {
    int to = bits[i];
    if(i < to) { std::swap(a[i], a[to]); }
  }
  for(int len = 1; len < n; len *= 2) {
    for(int i = 0; i < n; i += len * 2) {
      for(int j = 0; j < len; j++) {
        int u = a[i + j], v = (int)((long long) a[i + j + len] *
             root[len + j] % MOD);
        a[i + j] = add(u, v);
        a[i + j + len] = add(u, MOD - v);
      }
    }
  }
  if(inv) {
    long long rev = fexp(n, MOD-2, MOD);
    for(int i = 0; i < n; i++)
      a[i] = (int)(a[i] * rev % MOD);
  }
  return a;
}

std::vector<int> shift(const std::vector<int> &a, int s) {
  int n = std::max(0, s + (int) a.size());
  std::vector<int> b(n, 0);
  for(int i = std::max(-s, 0); i < (int) a.size(); i++) {
    b[i + s] = a[i];
  }
  return b;
}

std::vector<int> cut(const std::vector<int> &a, int n) {
  std::vector<int> b(n, 0);
  for(int i = 0; i < (int) a.size() && i < n; i++) {
    b[i] = a[i];
  }
  return b;
}

std::vector<int> operator +(std::vector<int> a, const std::
     vector<int> &b) {
  int sz = (int) std::max(a.size(), b.size());
  a.resize(sz, 0);
  for(int i = 0; i < (int) b.size(); i++) {
    a[i] = add(a[i], b[i]);
  }
  return a;
}

std::vector<int> operator -(std::vector<int> a, const std::
     vector<int> &b) {
  int sz = (int) std::max(a.size(), b.size());
  a.resize(sz, 0);
  for(int i = 0; i < (int) b.size(); i++) {
    a[i] = add(a[i], MOD - b[i]);
  }
  return a;
}

std::vector<int> operator *(std::vector<int> a, std::vector<int>
     b) {
  while(!a.empty() && a.back() == 0) a.pop_back();
  while(!b.empty() && b.back() == 0) b.pop_back();
  if(a.empty() || b.empty()) return std::vector<int>(0, 0);
  int n = 1;
  while(n-1 < (int) a.size() + (int) b.size() - 2) n += n;
  a.resize(n, 0);
  b.resize(n, 0);
  a = fft(a, false);
  b = fft(b, false);
  for(int i = 0; i < n; i++) {
    a[i] = (int) ((long long) a[i] * b[i] % MOD);
  }
  return fft(a, true);
}

std::vector<int> inverse(const std::vector<int> &a, int k) {
  assert(!a.empty() && a[0] != 0);
  if(k == 0) {
    return std::vector<int>(1, (int) fexp(a[0], MOD - 2));
  } else {
    int n = 1 << k;
    auto c = inverse(a, k-1);
    return cut(c * cut(std::vector<int>(1, 2) - cut(a, n) * c, n
         ), n);
  }
}
```

```
std::vector<int> operator /(std::vector<int> a, std::vector<int>
      b) {
  // NEED TO TEST!
  while(!a.empty() && a.back() == 0) a.pop_back();
  while(!b.empty() && b.back() == 0) b.pop_back();
  assert(!b.empty());
  if(a.size() < b.size()) return std::vector<int>(1, 0);
  std::reverse(a.begin(), a.end());
  std::reverse(b.begin(), b.end());
  int n = (int) a.size() - (int) b.size() + 1;
  int k = 0;
  while((1 << k) - 1 < n) k++;
  a = cut(a * inverse(b, k), (int) a.size() - (int) b.size() +
       1);
  std::reverse(a.begin(), a.end());
  return a;
}

std::vector<int> log(const std::vector<int> &a, int k) {
  assert(!a.empty() && a[0] != 0);
  int n = 1 << k;
  std::vector<int> b(n, 0);
  for(int i = 0; i+1 < (int) a.size() && i < n; i++) {
    b[i] = (int)((i + 1LL) * a[i+1] % MOD);
  }
  b = cut(b * inverse(a, k), n);
  assert((int) b.size() == n);
  for(int i = n - 1; i > 0; i--) {
    b[i] = (int) (b[i-1] * fexp(i, MOD - 2) % MOD);
  }
  b[0] = 0;
  return b;
}

std::vector<int> exp(const std::vector<int> &a, int k) {
  assert(!a.empty() && a[0] == 0);
  if(k == 0) {
    return std::vector<int>(1, 1);
  } else {
    auto b = exp(a, k-1);
    int n = 1 << k;
    return cut(b * cut(std::vector<int>(1, 1) + cut(a, n) - log(
          b, k), n), n);
  }
}
```

6.16 Pollard-Rho

```
// factor(N, v) to get N factorized in vector v
// O(N ^ (1 / 4)) on average
// Miller-Rabin - Primarily Test O(|base|*(logn)^2)
ll addmod(ll a, ll b, ll m){
  if(a >= m - b) return a + b - m;
  return a + b;
}

ll mulmod(ll a, ll b, ll m){
  ll ans = 0;
  while(b){
    if(b & 1) ans = addmod(ans, a, m);
    a = addmod(a, a, m);
    b >>= 1;
  }
  return ans;
}

ll fexp(ll a, ll b, ll n){
  ll r = 1;
  while(b){
    if(b & 1) r = mulmod(r, a, n);
    a = mulmod(a, a, n);
    b >>= 1;
  }
  return r;
}

bool miller(ll a, ll n){
  if (a >= n) return true;
  ll s = 0, d = n - 1;
  while(d % 2 == 0) d >>= 1, s++;
  ll x = fexp(a, d, n);
  if (x == 1 || x == n - 1) return true;
  for (int r = 0; r < s; r++, x = mulmod(x,x,n)){
    if (x == 1) return false;
    if (x == n - 1) return true;
  }
  return false;
}

bool isprime(ll n){
  if(n == 1) return false;
  int base[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
  for (int i = 0; i < 12; ++i) if (!miller(base[i], n)) return
      false;
  return true;
}

ll pollard(ll n){
  ll x, y, d, c = 1;
  if (n % 2 == 0) return 2;
  while(true){
    y = x = 2;
    while(true){
      x = addmod(mulmod(x,x,n), c, n);
      y = addmod(mulmod(y,y,n), c, n);
      y = addmod(mulmod(y,y,n), c, n);
      if (x == y) break;
      d = __gcd(abs(x-y), n);
      if (d > 1) return d;
    }
    c++;
  }
}

vector<ll> factor(ll n){
  if (n == 1 || isprime(n)) return {n};
  ll f = pollard(n);
  vector<ll> l = factor(f), r = factor(n / f);
  l.insert(l.end(), r.begin(), r.end());
  sort(l.begin(), l.end());
  return l;
}

//n < 2,047 base = {2};
//n < 9,080,191 base = {31, 73};
//n < 2,152,302,898,747 base = {2, 3, 5, 7, 11};
//n < 318,665,857,834,031,151,167,461 base = {2, 3, 5, 7, 11,
    13, 17, 19, 23, 29, 31, 37};
//n < 3,317,044,064,679,887,385,961,981 base = {2, 3, 5, 7, 11,
    13, 17, 19, 23, 29, 31, 37, 41};
```

6.17 Primitive Root

```
// Finds a primitive root modulo p
// To make it works for any value of p, we must add calculation
    of phi(p)
// n is 1, 2, 4 or p^k or 2*p^k (p odd in both cases)

//is n primitive root of p ?
bool test(long long x, long long p) {
  long long m = p - 1;
  for(int i = 2; i * i <= m; ++i) if(!(m % i)) {
    if(fexp(x, i, p) == 1) return false;
    if(fexp(x, m / i, p) == 1) return false;
  }
  return true;
}
//find the smallest primitive root for p
int search(int p) {
  for(int i = 2; i < p; i++) if(test(i, p)) return i;
  return -1;
}
```

6.18 Sieve of Eratosthenes

```
// Sieve of Erasthotenes
int p[N]; vi primes;

for (ll i = 2; i < N; ++i) if (!p[i]) {
  for (ll j = i*i; j < N; j+=i) p[j]=1;
  primes.pb(i);
}
```

6.19 Simpson Rule

```
// Simpson Integration Rule
// define the function f
double f(double x) {
  // ...
}

double simpson(double a, double b, int n = 1e6) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i += 2) s += 4 * f(a + h*i);
    for (int i = 2; i < n; i += 2) s += 2 * f(a + h*i);
    return s*h/3;
}
```

6.20 Simplex (Stanford)

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

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)
        ) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i
        ][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[
        i][n + 1] = b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    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;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s]
            && N[j] < N[s]) s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
```

```
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
            (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) &&
            B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);

  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

7 Geometry

7.1 Miscellaneous

```
/*
1)  Square (n = 4) is the only regular polygon with integer
    coordinates

2)  Pick's theorem:  A = i + b/2 - 1
  A: area of the polygon
  i: number of interior points
  b: number of points on the border

3) Conic Rotations
  Given elipse: Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0
  Convert it to: Ax^2 + Bxy + Cy^2 + Dx + Ey = 1 (this formula
      suits better for elipse, before doing this verify F = 0)
  Final conversion: A(x + D/2A)^2 + C(y + E/2C)^2 = 1 + D^2/4A +
      E^2/4C
    B != 0 (Rotate):
      theta = atan2(b, c-a)/2.0;
      A' = (a + c + b/sin(2.0*theta))/2.0; // A
      C' = (a + c - b/sin(2.0*theta))/2.0; // C
      D' = d*sin(theta) + e*cos(theta); // D
      E' = d*cos(theta) - e*sin(theta); // E
    If you do any point calculation, for example finding elipses
        focus, remember to rotate the points by theta after!
*/
```

7.2 Basics (Point)

```
const long double EPS = 1e-9;

typedef long double type;
//for big coordinates change to long long

bool ge(type x, type y) { return x + EPS > y; }
bool le(type x, type y) { return x - EPS < y; }
bool eq(type x, type y) { return ge(x, y) and le(x, y); }
int sign(type x) { return ge(x, 0) - le(x, 0); }

struct point {
  type x, y;
  // 0 => same direction
  // 1 => p is on the left
  //-1 => p is on the right
  int dir(point o, point p) {
    type x = (*this - o) % (p - o);
    return ge(x,0) - le(x,0);
  }

  bool on_seg(point p, point q) {
    if (this->dir(p, q)) return 0;
    return ge(x, min(p.x, q.x)) and le(x, max(p.x, q.x)) and ge(
        y, min(p.y, q.y)) and le(y, max(p.y, q.y));
  }
  //rotation: cos * x - sin * y, sin * x + cos * y
};
int direction(point o, point p, point q) { return p.dir(o, q); }

//double area
type area_2(point a, point b, point c) { return cross(a,b) +
    cross(b,c) + cross(c,a); }

//angle between (a1 and b1) vs angle between (a2 and b2)
//1  : bigger
//-1 : smaller
//0  : equal
int angle_less(const point& a1, const point& b1, const point& a2
    , const point& b2) {
  point p1(dot(   a1, b1), abs(cross(   a1, b1)));
  point p2(dot(   a2, b2), abs(cross(   a2, b2)));
  if(cross(p1, p2) < 0) return 1;
  if(cross(p1, p2) > 0) return -1;
  return 0;
}
```

7.3 Radial Sort

```
#include "basics.cpp"
point origin;

/*
  below < above
  order: [pi, 2 * pi)
*/

int above(point p){
  if(p.y == origin.y) return p.x > origin.x;
  return p.y > origin.y;
}

bool cmp(point p, point q){
  int tmp = above(q) - above(p);
  if(tmp) return tmp > 0;
  return p.dir(origin,q) > 0;
  //Be Careful: p.dir(origin,q) == 0
}
```

7.4 Circle

```
#include "basics.cpp"
#include "lines.cpp"

struct circle {
  point c;
  ld r;
  circle() { c = point(); r = 0; }
  circle(point _c, ld _r) : c(_c), r(_r) {}
  ld area() { return acos(-1.0)*r*r; }
  ld chord(ld rad) { return  2*r*sin(rad/2.0); }
  ld sector(ld rad) { return 0.5*rad*area()/acos(-1.0); }
  bool intersects(circle other) {
    return le(c.dist(other.c), r + other.r);
  }
  bool contains(point p) { return le(c.dist(p), r); }
  pair<point, point> getTangentPoint(point p) {
    ld d1 = c.dist(p), theta = asin(r/d1);
    point p1 = (c - p).rotate(-theta);
    point p2 = (c - p).rotate(theta);
    p1 = p1*(sqrt(d1*d1 - r*r)/d1) + p;
    p2 = p2*(sqrt(d1*d1 - r*r)/d1) + p;
    return make_pair(p1,p2);
  }
};

circle circumcircle(point a, point b, point c) {
  circle ans;
  point u = point((b - a).y, -(b - a).x);
  point v = point((c - a).y, -(c - a).x);
  point n = (c - b)*0.5;
  ld t = cross(u,n)/cross(v,u);
  ans.c = ((a + c)*0.5) + (v*t);
  ans.r = ans.c.dist(a);
  return ans;
}

point compute_circle_center(point a, point b, point c) {
  //circumcenter
  b = (a + b)/2;
  c = (a + c)/2;
  return compute_line_intersection(b, b + rotate_cw90(a - b), c,
      c + rotate_cw90(a - c));
}

int inside_circle(point p, circle c) {
  if (fabs(p.dist(c.c) - c.r)<EPS) return 1;
  else if (p.dist(c.c) < c.r) return 0;
  else return 2;
} //0 = inside/1 = border/2 = outside

circle incircle( point p1, point p2, point p3 ) {
  ld m1 = p2.dist(p3);
  ld m2 = p1.dist(p3);
  ld m3 = p1.dist(p2);
  point c = (p1*m1 + p2*m2 + p3*m3)*(1/(m1 + m2 + m3));
  ld s = 0.5*(m1 + m2 + m3);
  ld r = sqrt(s*(s - m1)*(s - m2)*(s - m3))/s;
  return circle(c, r);
}

circle minimum_circle(vector<point> p) {
  random_shuffle(p.begin(), p.end());
  circle C = circle(p[0], 0.0);
  for(int i = 0; i < (int)p.size(); i++) {
    if (C.contains(p[i])) continue;
    C = circle(p[i], 0.0);
    for(int j = 0; j < i; j++) {
      if (C.contains(p[j])) continue;
      C = circle((p[j] + p[i])*0.5, 0.5*p[j].dist(p[i]));
      for(int k = 0; k < j; k++) {
        if (C.contains(p[k])) continue;
        C = circumcircle(p[j], p[i], p[k]);
      }
    }
  }
  return C;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<point> circle_line_intersection(point a, point b, point c
    , ld r) {
  vector<point> ret;
  b = b - a;
  a = a - c;
  ld A = dot(b, b);
  ld B = dot(a, b);
  ld C = dot(a, a) - r*r;
  ld D = B*B - A*C;
```

```
  if (D < -EPS) return ret;
  ret.push_back(c + a + b*(sqrt(D + EPS) - B)/A);
  if (D > EPS)
    ret.push_back(c + a + b*(-B - sqrt(D))/A);
  return ret;
}

vector<point> circle_circle_intersection(point a, point b, ld r,
     ld R) {
  vector<point> ret;
  ld d = sqrt(a.dist2(b));
  if (d > r + R || d + min(r, R) < max(r, R)) return ret;
  ld x = (d*d - R*R + r*r)/(2*d);
  ld y = sqrt(r*r - x*x);
  point v = (b - a)/d;
  ret.push_back(a + v*x + rotate_ccw90(v)*y);
  if (y > 0)
    ret.push_back(a + v*x - rotate_ccw90(v)*y);
  return ret;
}

//GREAT CIRCLE

double gcTheta(double pLat, double pLong, double qLat, double
     qLong) {
  pLat *= acos(-1.0) / 180.0; pLong *= acos(-1.0) / 180.0; //
       convert degree to radian
  qLat *= acos(-1.0) / 180.0; qLong *= acos(-1.0) / 180.0;
  return acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
    cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
    sin(pLat)*sin(qLat));
}

double gcDistance(double pLat, double pLong, double qLat, double
     qLong, double radius) {
  return radius*gcTheta(pLat, pLong, qLat, qLong);
}
```

7.5 Closest Pair of Points

```
#include "basics.cpp"
//DIVIDE AND CONQUER METHOD
//Warning: include variable id into the struct point

struct cmp_y {
  bool operator()(const point & a, const point & b) const {
    return a.y < b.y;
  }
};

ld min_dist = LINF;
pair<int, int> best_pair;
vector<point> pts, stripe;
int n;

void upd_ans(const point & a, const point & b) {
  ld dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.
       y));
  if (dist < min_dist) {
    min_dist = dist;
    // best_pair = {a.id, b.id};
  }
}

void closest_pair(int l, int r) {
  if (r - l <= 3) {
    for (int i = l; i < r; ++i) {
      for (int j = i + 1; j < r; ++j) {
        upd_ans(pts[i], pts[j]);
      }
    }
    sort(pts.begin() + l, pts.begin() + r, cmp_y());
    return;
  }

  int m = (l + r) >> 1;
  type midx = pts[m].x;
  closest_pair(l, m);
  closest_pair(m, r);

  merge(pts.begin() + l, pts.begin() + m, pts.begin() + m, pts.
       begin() + r, stripe.begin(), cmp_y());
  copy(stripe.begin(), stripe.begin() + r - l, pts.begin() + l);

  int stripe_sz = 0;
  for (int i = l; i < r; ++i) {
    if (abs(pts[i].x - midx) < min_dist) {
      for (int j = stripe_sz - 1; j >= 0 && pts[i].y - stripe[j
           ].y < min_dist; --j)
        upd_ans(pts[i], stripe[j]);
      stripe[stripe_sz++] = pts[i];
    }
  }

  //3D (sort points by Z before starting)(cfloor in math/basics)
  //map opposite side
  map<pll, vector<int>> f;
  for(int i = m; i < r; i++){
    f[{cfloor(pts[i].x, min_dist), cfloor(pts[i].y, min_dist)}].
         push_back(i);
  }
  //find
  for(int i = l; i < m; i++){
    if((midz - pts[i].z) * (midz - pts[i].z) >= min_dist)
         continue;

    pll cur = {cfloor(pts[i].x, min_dist), cfloor(pts[i].y,
         min_dist)};
    for(int dx = -1; dx <= 1; dx++)
      for(int dy = -1; dy <= 1; dy++)
        for(auto p : f[{cur.st + dx, cur.nd + dy}])
          min_dist = min(min_dist, pts[i].dist2(pts[p]));
  }
}

int main(){
  //read and save in vector pts
  min_dist = LINF;
  stripe.resize(n);
  sort(pts.begin(), pts.end());
  closest_pair(0, n);
}
```

7.6 Half Plane Intersection

```
// Intersection of halfplanes - O(nlogn)
// Points are given in counterclockwise order
//
// by Agnez

typedef vector<point> polygon;

int cmp(ld x, ld y = 0, ld tol = EPS) {
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1; }

bool comp(point a, point b){
    if((cmp(a.x) > 0 || (cmp(a.x) == 0 && cmp(a.y) > 0) ) && (
         cmp(b.x) < 0 || (cmp(b.x) == 0 && cmp(b.y) < 0)))
         return 1;
    if((cmp(b.x) > 0 || (cmp(b.x) == 0 && cmp(b.y) > 0) ) && (
         cmp(a.x) < 0 || (cmp(a.x) == 0 && cmp(a.y) < 0)))
         return 0;
    ll R = a%b;
    if(R) return R > 0;
    return false;
}

namespace halfplane{
  struct L{
    point p,v;
    L(){}
    L(point P, point V):p(P),v(V){}
    bool operator<(const L &b)const{ return comp(v, b.v); }
  };
  vector<L> line;
  void addL(point a, point b){line.pb(L(a,b-a));}
  bool left(point &p, L &l){ return cmp(l.v % (p-l.p))>0; }
  bool left_equal(point &p, L &l){ return cmp(l.v % (p-l.p))>=0;
       }
  void init(){ line.clear(); }

  point pos(L &a, L &b){
    point x=a.p-b.p;
    ld t = (b.v % x)/(a.v % b.v);
    return a.p+a.v*t;
  }

  polygon intersect(){
    sort(line.begin(), line.end());
    deque<L> q; //linhas da intersecao
    deque<point> p; //pontos de intersecao entre elas
    q.push_back(line[0]);
    for(int i=1; i < (int) line.size(); i++){
      while(q.size()>1 && !left(p.back(), line[i]))
        q.pop_back(), p.pop_back();
      while(q.size()>1 && !left(p.front(), line[i]))
        q.pop_front(), p.pop_front();
      if(!cmp(q.back().v % line[i].v) && !left(q.back().p,line[i
           ]))
        q.back() = line[i];
      else if(cmp(q.back().v % line[i].v))
        q.push_back(line[i]), p.push_back(point());
      if(q.size()>1)
        p.back()=pos(q.back(),q[q.size()-2]);
    }
    while(q.size()>1 && !left(p.back(),q.front()))
      q.pop_back(), p.pop_back();
    if(q.size() <= 2) return polygon(); //Nao forma poligono (
         pode nao ter intersecao)
    if(!cmp(q.back().v % q.front().v)) return polygon(); //Lados
          paralelos -> area infinita
    point ult = pos(q.back(),q.front());

    bool ok = 1;
    for(int i=0; i < (int) line.size(); i++)
      if(!left_equal(ult,line[i])){ ok=0; break; }

    if(ok) p.push_back(ult); //Se formar um poligono fechado
    polygon ret;
    for(int i=0; i < (int) p.size(); i++)
      ret.pb(p[i]);
    return ret;
  }
};
```

7.7 Lines

```
//Suggestion: for line intersections check
     line_line_intersection and then use
     compute_line_intersection
//Distance(point - segment): Project point and calculate
     distance
//Segments Distance: brute distance (point - segment) for all
     border points

point project_point_line(point c, point a, point b) {
  ld r = dot(b - a,b - a);
  if (fabs(r) < EPS) return a;
  return a + (b - a)*dot(c - a, b - a)/dot(b - a, b - a);
}

point project_point_ray(point c, point a, point b) {
  ld r = dot(b - a, b - a);
  if (fabs(r) < EPS) return a;
  r = dot(c - a, b - a) / r;
  if (le(r, 0)) return a;
  return a + (b - a)*r;
}

point project_point_segment(point c, point a, point b) {
  ld r = dot(b - a, b - a);
  if (fabs(r) < EPS) return a;
  r = dot(c - a, b - a)/r;
  if (le(r, 0)) return a;
  if (ge(r, 1)) return b;
  return a + (b - a)*r;
}

ld distance_point_plane(ld x, ld y, ld z,
            ld a, ld b, ld c, ld d)
{
  return fabs(a*x + b*y + c*z - d)/sqrt(a*a + b*b + c*c);
}

bool lines_parallel(point a, point b, point c, point d) {
  return fabs(cross(b - a, d - c)) < EPS;
}

bool lines_collinear(point a, point b, point c, point d) {
  return lines_parallel(a, b, c, d)
    && fabs(cross(a-b, a-c)) < EPS
    && fabs(cross(c-d, c-a)) < EPS;
}

point lines_intersect(point p, point q, point a, point b) {
  point r = q - p, s = b - a, c(p%q, a%b);
  if (eq(r%s,0)) return point(LINF, LINF);
  return point(point(r.x, s.x) % c, point(r.y, s.y) % c) / (r%s)
       ;
}
```

```cpp
//be careful: test line_line_intersection before using this function
point compute_line_intersection(point a, point b, point c, point d) {
  b = b - a; d = c - d; c = c - a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

bool line_line_intersect(point a, point b, point c, point d) {
  if(!lines_parallel(a, b, c, d)) return true;
  if(lines_collinear(a, b, c, d)) return true;
  return false;
}

//rays in direction a -> b, c -> d
bool ray_ray_intersect(point a, point b, point c, point d){
  if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
    b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
  if (lines_collinear(a, b, c, d)) {
    if(ge(dot(b - a, d - c), 0)) return true;
    if(ge(dot(a - c, d - c), 0)) return true;
    return false;
  }
  if(!line_line_intersect(a, b, c, d)) return false;
  point inters = lines_intersect(a, b, c, d);
  if(ge(dot(inters - c, d - c), 0) && ge(dot(inters - a, b - a), 0)) return true;
  return false;
}

bool segment_segment_intersect(point a, point b, point c, point d) {
  if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
    b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
  int d1, d2, d3, d4;
  d1 = direction(a, b, c);
  d2 = direction(a, b, d);
  d3 = direction(c, d, a);
  d4 = direction(c, d, b);
  if (d1*d2 < 0 and d3*d4 < 0) return 1;
  return a.on_seg(c, d) or b.on_seg(c, d) or
    c.on_seg(a, b) or d.on_seg(a, b);
}

bool segment_line_intersect(point a, point b, point c, point d){
  if(!line_line_intersect(a, b, c, d)) return false;
  point inters = lines_intersect(a, b, c, d);
  if(inters.on_seg(a, b)) return true;
  return false;
}

//ray in direction c -> d
bool segment_ray_intersect(point a, point b, point c, point d){
  if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
    b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
  if (lines_collinear(a, b, c, d)) {
    if(c.on_seg(a, b)) return true;
    if(ge(dot(d - c, a - c), 0)) return true;
    return false;
  }
  if(!line_line_intersect(a, b, c, d)) return false;
  point inters = lines_intersect(a, b, c, d);
  if(!inters.on_seg(a, b)) return false;
  if(ge(dot(inters - c, d - c), 0)) return true;
  return false;
}

//ray in direction a -> b
bool ray_line_intersect(point a, point b, point c, point d){
  if (a.dist2(c) < EPS || a.dist2(d) < EPS ||
    b.dist2(c) < EPS || b.dist2(d) < EPS) return true;
  if (!line_line_intersect(a, b, c, d)) return false;
  point inters = lines_intersect(a, b, c, d);
  if(!line_line_intersect(a, b, c, d)) return false;
  if(ge(dot(inters - a, b - a), 0)) return true;
  return false;
}
```

7.8 Minkowski Sum

```cpp
#include "basics.cpp"
#include "polygons.cpp"
//ITA MINKOWSKI

typedef vector<point> polygon;

/*
 * Minkowski sum
   Distance between two polygons P and Q:
    Do Minkowski(P, Q)
    Ans = min(ans, dist((0, 0), edge))
 */

polygon minkowski(polygon & A, polygon & B) {
  polygon P; point v1, v2;
  sort_lex_hull(A), sort_lex_hull(B);
  int n1 = A.size(), n2 = B.size();
  P.push_back(A[0] + B[0]);
  for(int i = 0, j = 0; i < n1 || j < n2;) {
    v1 = A[(i + 1)%n1] - A[i%n1];
    v2 = B[(j + 1)%n2] - B[j%n2];
    if (j == n2 || cross(v1, v2) > EPS) {
      P.push_back(P.back() + v1); i++;
    }
    else if (i == n1 || cross(v1, v2) < -EPS) {
      P.push_back(P.back() + v2); j++;
    }
    else {
      P.push_back(P.back() + (v1 + v2));
      i++; j++;
    }
  }
  P.pop_back();
  sort_lex_hull(P);
  return P;
}
```

7.9 Nearest Neighbour

```cpp
// Closest Neighbor - O(n * log(n))
const ll N = 1e6+3, INF = 1e18;
ll n, cn[N], x[N], y[N]; // number of points, closes neighbor, x coordinates, y coordinates

ll sqr(ll i) { return i*i; }
ll dist(int i, int j) { return sqr(x[i]-x[j]) + sqr(y[i]-y[j]); }
ll dist(int i) { return i == cn[i] ? INF : dist(i, cn[i]); }

bool cpx(int i, int j) { return x[i] < x[j] or (x[i] == x[j] and y[i] < y[j]); }
bool cpy(int i, int j) { return y[i] < y[j] or (y[i] == y[j] and x[i] < x[j]); }

ll calc(int i, ll x0) {
  ll dlt = dist(i) - sqr(x[i]-x0);
  return dlt >= 0 ? ceil(sqrt(dlt)) : -1;
}

void updt(int i, int j, ll x0, ll &dlt) {
  if (dist(i) > dist(i, j)) cn[i] = j, dlt = calc(i, x0);
}

void cmp(vi &u, vi &v, ll x0) {
  for(int a=0, b=0; a<u.size(); ++a) {
    ll i = u[a], dlt = calc(i, x0);
    while(b < v.size() and y[i] > y[v[b]]) b++;
    for(int j = b-1; j >= 0     and y[i] - dlt <= y[v[j]]; j--)
      updt(i, v[j], x0, dlt);
    for(int j = b; j < v.size() and y[i] + dlt >= y[v[j]]; j++)
      updt(i, v[j], x0, dlt);
  }
}

void slv(vi &ix, vi &iy) {
  int n = ix.size();
  if (n == 1) { cn[ix[0]] = ix[0]; return; }

  int m = ix[n/2];

  vi ix1, ix2, iy1, iy2;
  for(int i=0; i<n; ++i) {
    if (cpx(ix[i], m)) ix1.push_back(ix[i]);
    else ix2.push_back(ix[i]);

    if (cpx(iy[i], m)) iy1.push_back(iy[i]);
    else iy2.push_back(iy[i]);
  }

  slv(ix1, iy1);
  slv(ix2, iy2);

  cmp(iy1, iy2, x[m]);
  cmp(iy2, iy1, x[m]);
}

void slv(int n) {
  vi ix, iy;
  ix.resize(n);
  iy.resize(n);
  for(int i=0; i<n; ++i) ix[i] = iy[i] = i;
  sort(ix.begin(), ix.end(), cpx);
  sort(iy.begin(), iy.end(), cpy);
  slv(ix, iy);
}
```

7.10 Polygons

```cpp
#include "basics.cpp"
#include "lines.cpp"

//Monotone chain O(nlog(n))
#define REMOVE_REDUNDANT
#ifdef REMOVE_REDUNDANT
bool between(const point &a, const point &b, const point &c) {
  return (fabs(area_2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0
      && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

//new change: <= 0 / >= 0 became < 0 / > 0 (yet to be tested)

void convex_hull(vector<point> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<point> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area_2(up[up.size()-2], up.back(), pts[i]) > 0) up.pop_back();
    while (dn.size() > 1 && area_2(dn[dn.size()-2], dn.back(), pts[i]) < 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

  #ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
  #endif
}

//avoid using long double for comparisons, change type and add division by 2
type compute_signed_area(const vector<point> &p) {
  type area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area;
}

point compute_centroid(vector<point> &p) {
  point c(0,0);
  ld scale = 3.0 * compute_signed_area(p);
  for (int i = 0; i < p.size(); i++){
      int j = (i+1) % p.size();
      c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
```

```
}

bool point_in_triangle(point a, point b, point c, point cur){
  ll s1 = abs(cross(b - a, c - a));
  ll s2 = abs(cross(a - cur, b - cur)) + abs(cross(b - cur, c -
       cur)) + abs(cross(c - cur, a - cur));
  return s1 == s2;
}

void sort_lex_hull(vector<point> &hull){
  if(compute_signed_area(hull) < 0) reverse(hull.begin(), hull.
       end());
  int n = hull.size();

  //Sort hull by x
  int pos = 0;
  for(int i = 1; i < n; i++) if(hull[i] <  hull[pos]) pos = i;
  rotate(hull.begin(), hull.begin() + pos, hull.end());
}

//determine if point is inside or on the boundary of a polygon (
     O(logn))
bool point_in_convex_polygon(vector<point> &hull, point cur){
  int n = hull.size();
  //Corner cases: point outside most left and most right wedges
  if(cur.dir(hull[0], hull[1]) != 0 && cur.dir(hull[0], hull[1])
        != hull[n - 1].dir(hull[0], hull[1]))
    return false;
  if(cur.dir(hull[0], hull[n - 1]) != 0 && cur.dir(hull[0], hull
       [n - 1]) != hull[1].dir(hull[0], hull[n - 1]))
    return false;

  //Binary search to find which wedges it is between
  int l = 1, r = n - 1;
  while(r - l > 1){
    int mid = (l + r)/2;
    if(cur.dir(hull[0], hull[mid]) <= 0)l = mid;
    else r = mid;
  }
  return point_in_triangle(hull[l], hull[l + 1], hull[0], cur);
}

//Shamos - Hoey for test polygon simple in O(nlog(n))
inline bool adj(int a, int b, int n) {return (b == (a + 1)%n or
     a == (b + 1)%n);}

struct edge{
  point ini, fim;
  edge(point ini = point(0,0), point fim = point(0,0)) : ini(ini
       ), fim(fim) {}
};

//< here means the edge on the top will be at the begin
bool operator < (const edge& a, const edge& b) {
  if (a.ini == b.ini) return direction(a.ini, a.fim, b.fim) < 0;
  if (a.ini.x < b.ini.x) return direction(a.ini, a.fim, b.ini) <
       0;
  return direction(a.ini, b.fim, b.ini) < 0;
}

bool is_simple_polygon(const vector<point> &pts){
  vector <pair<point, pii>> eve;
  vector <pair<edge, int>> edgs;
  set <pair<edge, int>> sweep;
  int n = (int)pts.size();
  for(int i = 0; i < n; i++){
    point l = min(pts[i], pts[(i + 1)%n]);
    point r = max(pts[i], pts[(i + 1)%n]);
    eve.pb({l, {0, i}});
    eve.pb({r, {1, i}});
    edgs.pb(make_pair(edge(l, r), i));
  }
  sort(eve.begin(), eve.end());
  for(auto e : eve){
    if(!e.nd.st){
      auto cur = sweep.lower_bound(edgs[e.nd.nd]);
      pair<edge, int> above, below;
      if(cur != sweep.end()){
        below = *cur;
        if(!adj(below.nd, e.nd.nd, n) and
             segment_segment_intersect(pts[below.nd], pts[(below
             .nd + 1)%n], pts[e.nd.nd], pts[(e.nd.nd + 1)%n]))
          return false;
      }
      if(cur != sweep.begin()){
        above = *(--cur);
        if(!adj(above.nd, e.nd.nd, n) and
             segment_segment_intersect(pts[above.nd], pts[(above
             .nd + 1)%n], pts[e.nd.nd], pts[(e.nd.nd + 1)%n]))
          return false;
      }
      sweep.insert(edgs[e.nd.nd]);
    }
    else{
      auto below = sweep.upper_bound(edgs[e.nd.nd]);
      auto cur = below, above = --cur;
      if(below != sweep.end() and above != sweep.begin()){
        --above;
        if(!adj(below->nd, above->nd, n) and
             segment_segment_intersect(pts[below->nd], pts[(
             below->nd + 1)%n], pts[above->nd], pts[(above->nd +
              1)%n]))
          return false;
      }
      sweep.erase(cur);
    }
  }
  return true;
}

// this code assumes that there are no 3 colinear points
int maximize_scalar_product(vector<point> &hull, point vec /*,
     int dir_flag*/) {
  /*
    For Minimize change: >= becomes <= and > becomes <
    For finding tangents, use same code passing direction flag
    dir_flag = -1 for right tangent
    dir_flag =  1 for left tangent
    >= or > becomes: == dir_flag
    < or <= becomes != dir_flag
    commentaries below for better clarification
  */
  int ans = 0;
  int n = hull.size();
  if(n < 20) {
    for(int i = 0; i < n; i++) {
      if(hull[i] * vec > hull[ans] * vec) {
        //hull[ans].dir(vec, hull[i]) == dir_flag
        ans = i;
      }
    }
  } else {
    if(hull[1] * vec > hull[ans] * vec) {
      //hull[ans].dir(vec, hull[i]) == dir_flag
      ans = 1;
    }
    for(int rep = 0; rep < 2; rep++) {
      int l = 2, r = n - 1;
      while(l != r) {
        int mid = (l + r + 1) / 2;
        bool flag = hull[mid] * vec >= hull[mid-1] * vec;
        //(hull[ans].dir(vec, hull[l]) == dir_flag
        if(rep == 0) { flag = flag && hull[mid] * vec >= hull[0]
              * vec; }
        //(hull[ans].dir(vec, hull[l]) == dir_flag
        else { flag = flag || hull[mid-1] * vec < hull[0] * vec;
             }
        //(hull[ans].dir(vec, hull[l]) != dir_flag
        if(flag) {
          l = mid;
        } else {
          r = mid - 1;
        }
      }
      if(hull[l] * vec > hull[ans] * vec) {
        //(hull[ans].dir(vec, hull[l]) == dir_flag
        ans = l;
      }
    }
  }
  return ans;
}
```

7.11 Delaunay Triangulation

```
/*
Complexity: O(nlogn)
Code by Bruno Maletta (UFMG): https://github.com/brunomaletta/
     Biblioteca

The definition of the Voronoi diagram immediately shows signs of
     applications.
*   Given a set S of n points and m query points p1,...,pm, we
      can answer for each query point, its nearest neighbor in S.
  This can be done in O((n+q)log(n+q)) offline by sweeping the
       Voronoi diagram and query points.
  Or it can be done online with persistent data structures.

*   For each Delaunay triangle, its circumcircle does not
      strictly contain any points in S. (In fact, you can also
      consider this the defining property of Delaunay
      triangulation)

*   The number of Delaunay edges is at most 3n - 6, so there is
      hope for an efficient construction.

*   Each point p belongs to S is adjacent to its nearest
      neighbor with a Delaunay edge.

*   The Delaunay triangulation maximizes the minimum angle in
      the triangles among all possible triangulations.

*   The Euclidean minimum spanning tree is a subset of Delaunay
      edges.

*/

#include "basics.cpp"

bool ccw(point a, point b, point c){ return area_2(a, b, c) > 0;
     }

typedef struct QuadEdge* Q;
struct QuadEdge {
  int id;
  point o;
  Q rot, nxt;
  bool used;

  QuadEdge(int id_ = -1, point o_ = point(INF, INF)) :
    id(id_), o(o_), rot(nullptr), nxt(nullptr), used(false) {}

  Q rev() const { return rot->rot; }
  Q next() const { return nxt; }
  Q prev() const { return rot->next()->rot; }
  point dest() const { return rev()->o; }
};

Q edge(point from, point to, int id_from, int id_to) {
  Q e1 = new QuadEdge(id_from, from);
  Q e2 = new QuadEdge(id_to, to);
  Q e3 = new QuadEdge;
  Q e4 = new QuadEdge;
  tie(e1->rot, e2->rot, e3->rot, e4->rot) = {e3, e4, e2, e1};
  tie(e1->nxt, e2->nxt, e3->nxt, e4->nxt) = {e1, e2, e4, e3};
  return e1;
}

void splice(Q a, Q b) {
  swap(a->nxt->rot->nxt, b->nxt->rot->nxt);
  swap(a->nxt, b->nxt);
}

void del_edge(Q& e, Q ne) { // delete e and assign e <- ne
  splice(e, e->prev());
  splice(e->rev(), e->rev()->prev());
  delete e->rev()->rot, delete e->rev();
  delete e->rot; delete e;
  e = ne;
}

Q conn(Q a, Q b) {
  Q e = edge(a->dest(), b->o, a->rev()->id, b->id);
  splice(e, a->rev()->prev());
  splice(e->rev(), b);
  return e;
}

bool in_c(point a, point b, point c, point p) { // p ta na
     circunf. (a, b, c) ?
  type p2 = p*p, A = a*a - p2, B = b*b - p2, C = c*c - p2;
  return area_2(p, a, b) * C + area_2(p, b, c) * A + area_2(p, c
       , a) * B > 0;
}

pair<Q, Q> build_tr(vector<point>& p, int l, int r) {
  if (r-l+1 <= 3) {
    Q a = edge(p[l], p[l+1], l, l+1), b = edge(p[l+1], p[r], l
         +1, r);
    if (r-l+1 == 2) return {a, a->rev()};
    splice(a->rev(), b);
    type ar = area_2(p[l], p[l+1], p[r]);
    Q c = ar ? conn(b, a) : 0;
    if (ar >= 0) return {a, b->rev()};
    return {c->rev(), c};
```

```
  }
  int m = (l+r)/2;
  auto [la, ra] = build_tr(p, l, m);
  auto [lb, rb] = build_tr(p, m+1, r);
  while (true) {
    if (ccw(lb->o, ra->o, ra->dest())) ra = ra->rev()->prev();
    else if (ccw(lb->o, ra->o, lb->dest())) lb = lb->rev()->next
        ();
    else break;
  }
  Q b = conn(lb->rev(), ra);
  auto valid = [&](Q e) { return ccw(e->dest(), b->dest(), b->o)
      ; };
  if (ra->o == la->o) la = b->rev();
  if (lb->o == rb->o) rb = b;
  while (true) {
    Q L = b->rev()->next();
    if (valid(L)) while (in_c(b->dest(), b->o, L->dest(), L->
        next()->dest()))
      del_edge(L, L->next());
    Q R = b->prev();
    if (valid(R)) while (in_c(b->dest(), b->o, R->dest(), R->
        prev()->dest()))
      del_edge(R, R->prev());
    if (!valid(L) and !valid(R)) break;
    if (!valid(L) or (valid(R) and in_c(L->dest(), L->o, R->o, R
        ->dest())))
      b = conn(R, b->rev());
    else b = conn(b->rev(), L->rev());
  }
  return {la, rb};
}

//NOTE: Before calculating Delaunay add a bound triangle: (-INF,
    -INF), (INF, INF), (0, INF)
vector<vector<int>> delaunay(vector<point> v) {
  int n = v.size();
  auto tmp = v;
  vector<int> idx(n);
  iota(idx.begin(), idx.end(), 0);
  sort(idx.begin(), idx.end(), [&](int l, int r) { return v[l] <
      v[r]; });
  for (int i = 0; i < n; i++) v[i] = tmp[idx[i]];
  assert(unique(v.begin(), v.end()) == v.end());
  vector<vector<int>> g(n);
  bool col = true;
  for (int i = 2; i < n; i++) if (area_2(v[i], v[i-1], v[i-2]))
      col = false;
  if (col) {
    for (int i = 1; i < n; i++){
      g[idx[i-1]].push_back(idx[i]), g[idx[i]].push_back(idx[i
          -1]);
    return g;
  }
  Q e = build_tr(v, 0, n-1).first;
  vector<Q> edg = {e};
  for (int i = 0; i < edg.size(); e = edg[i++]) {
    for (Q at = e; !at->used; at = at->next()) {
      at->used = true;
      g[idx[at->id]].push_back(idx[at->rev()->id]);
      edg.push_back(at->rev());
    }
  }
  return g;
}

vector<vector<point>> voronoi(const vector<point>& points, const
    vector<point>& delaunay){
  int n = delaunay.size();
  vector<vector<point>> voronoi(n, vector<point>());
  for(int i = 0; i < n; i++){
    for(int d = 0; d < delaunay[i].size(); d++){
      int j = delaunay[i][d], k = delaunay[i][(d + 1) % delaunay
          [i].size()];
      circle c = circumcircle(points[i], points[j], points[k]);
      voronoi[i].push_back(c.c);
      voronoi[j].push_back(c.c);
      voronoi[k].push_back(c.c);
    }
  }
}
```

8 Miscellaneous

8.1 Bitset

```
//Goes through the subsets of a set x :
int b = 0;
do {
// process subset b
} while (b=(b-x)&x);
```

8.2 builtin

```
__builtin_ctz(x) // trailing zeroes
__builtin_clz(x) // leading zeroes
__builtin_popcount(x) // # bits set
__builtin_ffs(x) // index(LSB) + 1 [0 if x==0]

// Add ll to the end for long long [__builtin_clzll(x)]
```

8.3 Date

```
struct Date {
  int d, m, y;
  static int mnt[], mntsum[];

  Date() : d(1), m(1), y(1) {}
  Date(int d, int m, int y) : d(d), m(m), y(y) {}
  Date(int days) : d(1), m(1), y(1) { advance(days); }

  bool bissexto() { return (y%4 == 0 and y%100) or (y%400 == 0);
       }

  int mdays() { return mnt[m] + (m == 2)*bissexto(); }
  int ydays() { return 365+bissexto(); }

  int msum()  { return mntsum[m-1] + (m > 2)*bissexto(); }
  int ysum()  { return 365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)
      /400; }

  int count() { return (d-1) + msum() + ysum(); }

  int day() {
    int x = y - (m<3);
    return (x + x/4 - x/100 + x/400 + mntsum[m-1] + d + 6)%7;
  }

  void advance(int days) {
    days += count();
    d = m = 1, y = 1 + days/366;
    days -= count();
    while(days >= ydays()) days -= ydays(), y++;
    while(days >= mdays()) days -= mdays(), m++;
    d += days;
  }
};

int Date::mnt[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
    30, 31};
int Date::mntsum[13] = {};
for(int i=1; i<13; ++i) Date::mntsum[i] = Date::mntsum[i-1] +
    Date::mnt[i];
```

8.4 Parentesis to Poslish (ITA)

```
#include <cstdio>
#include <map>
#include <stack>
using namespace std;

/*
 * Parenthetic to polish expression conversion
 */

inline bool isOp(char c) {
  return c=='+' || c=='-' || c=='*' || c=='/' || c=='^';
}

inline bool isCarac(char c) {
  return (c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' &&
      c<='9');
}

int paren2polish(char* paren, char* polish) {
  map<char, int> prec;
  prec['('] = 0;
  prec['+'] = prec['-'] = 1;
  prec['*'] = prec['/'] = 2;
  prec['^'] = 3;
  int len = 0;
  stack<char> op;
  for (int i = 0; paren[i]; i++) {
    if (isOp(paren[i])) {
      while (!op.empty() && prec[op.top()] >= prec[paren[i]]) {
        polish[len++] = op.top(); op.pop();
      }
      op.push(paren[i]);
    }
    else if (paren[i]=='(') op.push('(');
    else if (paren[i]==')') {
      for (; op.top()!='('; op.pop())
        polish[len++] = op.top();
      op.pop();
    }
    else if (isCarac(paren[i]))
      polish[len++] = paren[i];
  }
  for(; !op.empty(); op.pop())
    polish[len++] = op.top();
  polish[len] = 0;
  return len;
}

/*
 * TEST MATRIX
 */

int main() {
  int N, len;
  char polish[400], paren[400];
  scanf("%d", &N);
  for (int j=0; j<N; j++) {
    scanf(" %s", paren);
    paren2polish(paren, polish);
    printf("%s\n", polish);
  }
  return 0;
}
```

8.5 Parallel Binary Search

```
// Parallel Binary Search - O(nlog n * cost to update data
    structure + qlog n * cost for binary search condition)

struct Query { int i, ans; /*+ query related info*/ };
vector<Query> req;

void pbs(vector<Query>& qs, int l /* = min value*/, int r /* =
    max value*/) {
  if (qs.empty()) return;

  if (l == r) {
    for (auto& q : qs) req[q.i].ans = l;
    return;
  }

  int mid = (l + r) / 2;
  // mid = (l + r + 1) / 2 if different from simple upper/lower
      bound

  for (int i = l; i <= mid; i++) {
    // add value to data structure
  }

  vector<Query> vl, vr;
  for (auto& q : qs) {
    if (/* cond */) vl.push_back(q);
    else vr.push_back(q);
  }

  pbs(vr, mid + 1, r);

  for (int i = l; i <= mid; i++) {
    // remove value from data structure
  }

  pbs(vl, l, mid);
}
```

8.6 Python

```python
# reopen
import sys
sys.stdout = open('out','w')
sys.stdin  = open('in' ,'r')

//Dummy example
R = lambda: map(int, input().split())
n, k = R(),
v, t = [], [0]*n
for p, c, i in sorted(zip(R(), R(), range(n))):
        t[i] = sum(v)+c
        v += [c]
        v = sorted(v)[::-1]
        if len(v) > k:
                v.pop()
print(' '.join(map(str, t)))
```

8.7 Sqrt Decomposition

```cpp
// Square Root Decomposition (Mo's Algorithm) - O(n^(3/2))
const int N = 1e5+1, SQ = 500;
int n, m, v[N];

void add(int p) { /* add value to aggregated data structure */ }
void rem(int p) { /* remove value from aggregated data structure
        */ }

struct query { int i, l, r, ans; } qs[N];

bool c1(query a, query b) {
  if(a.l/SQ != b.l/SQ) return a.l < b.l;
  return a.l/SQ&1 ? a.r > b.r : a.r < b.r;
}

bool c2(query a, query b) { return a.i < b.i; }

/* inside main */
int l = 0, r = -1;
sort(qs, qs+m, c1);
for (int i = 0; i < m; ++i) {
  query &q = qs[i];
  while (r < q.r) add(v[++r]);
  while (r > q.r) rem(v[r--]);
  while (l < q.l) rem(v[l++]);
  while (l > q.l) add(v[--l]);

  q.ans = /* calculate answer */;
}

sort(qs, qs+m, c2); // sort to original order
```

8.8 Latitude Longitude (Stanford)

```cpp
/*
Converts from rectangular coordinates to latitude/longitude and
        vice
versa. Uses degrees (not radians).
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << " " << B.lat << " " << B.lon << endl;

  A = convert(B);
  cout << A.x << " " << A.y << " " << A.z << endl;
}
```

8.9 Week day

```cpp
int v[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
int day(int d, int m, int y) {
  y -= m<3;
  return (y + y/4 - y/100 + y/400 + v[m-1] + d)%7;
}
```

9 Math Extra

9.1 Combinatorial formulas

$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$
$\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$
$\sum_{k=0}^{n} k^4 = (6n^5+15n^4+10n^3-n)/30$
$\sum_{k=0}^{n} k^5 = (2n^6+6n^5+5n^4-n^2)/12$
$\sum_{k=0}^{n} x^k = (x^{n+1}-1)/(x-1)$
$\sum_{k=0}^{n} kx^k = (x-(n+1)x^{n+1}+nx^{n+2})/(x-1)^2$
$\binom{n}{k} = \frac{n!}{(n-k)!k!}$
$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$
$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$
$\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$
$\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k}$
$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$
$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$
$\sum_{k=1}^{n} k^2\binom{n}{k} = (n+n^2)2^{n-2}$
$\binom{m+n}{r} = \sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k}$
$\binom{n}{k} = \prod_{i=1}^{k}\frac{n-k+i}{i}$

9.2 Number theory identities

Lucas' Theorem: For non-negative integers m and n and a prime p,

$$\binom{m}{n} \equiv \prod_{i=0}^{k}\binom{m_i}{n_i} \pmod p,$$

where

$$m = m_k p^k + m_{k-1}p^{k-1} + \cdots + m_1 p + m_0$$

is the base p representation of m, and similarly for n.

9.3 Stirling Numbers of the second kind

Number of ways to partition a set of n numbers into k non-empty subsets.

$$\left\{ {n \atop k} \right\} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{(k-j)}\binom{k}{j}j^n$$

Recurrence relation:

$$\left\{ {0 \atop 0} \right\} = 1$$

$$\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 1$$

$$\left\{ {n+1 \atop k} \right\} = k\left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\}$$

9.4 Numerical integration

RK4: to integrate $\dot{y} = f(t, y)$ with $y_0 = y(t_0)$, compute

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\
k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\
k_4 &= f(t_n + h, y_n + hk_3) \\
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$